Experimental Program Verification in the Theorema System

Tudor Jebelean, Laura Kovacs, Nikolaj Popov[1]

Research Institute for Symbolic Computation,
Johannes Kepler University, A-4040 Linz, Austria
Institute e-Austria Timişoara, Romania

Abstract. We describe practical experiments of program verification in the frame of the *Theorema* system (`www.theorema.org`). This includes both functional programs (using fixpoint theory), as well as imperative programs (using Hoare logic). By comparing different approaches we are trying to find general schemes which are useful for practical work. The *Theorema* system offers facilities for working with higher-order predicate logic formulae (including various general and domain-oriented provers) and also for defining and testing algorithms both in functional and in imperative styles. We generate verification conditions as natural-style predicate logic formulae, which can be then proven by *Theorema*, by issuing natural-style proofs which are human–readable.

Introduction

We describe the theoretical basis and practical experiments of program verification in the frame of the *Theorema* system (`www.theorema.org`) [5]. This work (in progress) is built on previous theoretical results and practical experiments regarding verification of functional programs, as well as of imperative programs [1, 19, 14]. We follow three main approaches:

- Hoare logic for imperative programs,
- fixpoint theory of functions for functional programs,
- forward verification for recursive programs.

The first approach (Section 1) consists in applying the model of Hoare logic and the method of weakest precondition. This is relatively straightforward, but we are able to show some interesting results concerning the automatic synthesis of loop invariants and of termination terms.

The second approach (Section 2) is based on the extraction (for certain classes of programs) of the purely logical conditions which are sufficient for the program correctness. These are inferred using Scott induction or induction on natural numbers in the fixpoint theory of functions and constitute a meta-theorem which is proven once for the whole class. The concrete verification conditions for each program are then provable without having to use the fixpoint theory.

[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.

The third approach (Section 3) consists in identifying some basic intuitive principles for the generation of the verification conditions, which are then formalised in a precise method. We prove that the correctness of the program is a logical consequence of these verification conditions, without using an additional theory of computation.

Currently we are comparing the results of these three approaches, in order to realize a practical verification engine in the frame of the *Theorema* system. The *Theorema* system is a computer mathematical assistant which is implemented on top of the computer algebra system *Mathematica* [30]. The systems allows the definition and the organization of mathematical theories (including the description of algorithms) in the language of higher–order predicate logic, and also offers the necessary environment for experimenting with algorithms (computing) and for investigating the properties of mathematical structures (proving, solving). Currently the computing and solving capabilities are imported from the underlying computer algebra system *Mathematica*, and are quite powerful, thus the most significant and interesting part of the *Theorema* system is the one providing proving capabilities. Proving is implemented in the system by way of various domain–specific provers (propositional, first–order predicate logic, limit domains, induction of natural numbers and over lists, proving over sets, proving equalities by Knuth–Bendix completion, etc.). The general approach is to implement inference rules, as well as strategies and techniques which are similar to the human style of proving. The proofs which are produced are explained in natural language and are quite human–readable, which means that even failed proofs are useful because they may give hints for finding errors or omissions in the respective theory. Although the main emphasis of Theorema is on fully automatic proving, the system also has interactive facilities for allowing the user to manually influence the behaviour of the provers.

The problem of producing the proofs of the verification conditions is not in the scope of this paper. We note, however, that the concrete proof problems issued from program verification examples are used as test cases for the provers of *Theorema* and for experimenting with the organization and management of the mathematical knowledge. Currently we are able to generate the verification conditions and to prove them automatically (in matter of seconds) in the *Theorema* system for many concrete programs. There are, however, more complex examples which take a long time to prove or which cannot be proven fully automatically, but only with user guidance, as there are problems which will require the improvement of the current provers or the design of new provers.

1 Imperative Programs

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, as well as a verification condition generator [12] based on Hoare–Logic and using the Weakest Precondition Strategy. This verification tool provides readable arguments for the correctness of programs, 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. The source code of a program contains a sequence of the following statements:

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

These optional arguments (of WHILE and FOR) are needed for the verification of the program. The Verification Condition Generator (VCG) takes an annotated program with pre– and postcondition (i.e. the program specification) and produces as output a verification condition containing a collection of formulas. The verification condition generator is 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* syntax. This process is relatively straightforward, thus we will concentrate in this paper on the relatively more interesting problem of finding loop invariants and termination terms.

1.1 Generation of Loop Invariants

Verification of correctness of loops needs additional information, so-called annotations (invariants and termination terms). In most verification systems, these annotations are given by the user. It is generally agreed [6] 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. In this paper we present our work–in–progress technique for automated invariant generation by combinatorial and algebraic methods, which extends the work presented in [14]. We assume that the variables take values in the fixed field of real numbers $\mathbb{R}$ and that all the statements in the body of a loop are either assignments of the form $x := p$ (p is a polynomial), or WHILE loops.

1.2 Solving First Order Recurrences

Analysing the code of a loop, we identify recurrence equations for those variables which are modified during the execution of the loop (called *critical* variables).

Using these, an explicit expression is found for the value of each critical variable as a function of the index of loop iteration. By eliminating the loop index from the system of equations, one obtains invariant relations among the critical variables, which have to be embedded in the invariant of the loop.

For illustration, consider the "Division" program of two natural numbers:

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

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

In the above program, input parameters are specified by $\downarrow$, and output parameters by $\uparrow$. 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 the index of the loop. For each recursive equation we use the Gosper-Zeilberger algorithm (see e.g. [13],[7]). Namely, we use the Paule-Schorn implementation in *Mathematica* [26] which is already embedded in the *Theorema* system, in order to produce a closed–form for the expressions of quo_k and rem_k.

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

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 is extracted based on the following principle: at the termination of the loop (i.e. when the condition of the loop(Φ) is falsified), the so–far generated formulas together with the negation of the loop–condition and the assertion that is still needed to be generated have to imply the postcondition of the loop. Thus, by some heuristics and logical manipulation of formulae, the additional assertion is generated, and finally 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 termination term must be positive [8], we transform the given loop-condition using specific heuristics (algebraic manipulations) until we obtain a term T such that $T \geq 0 \ \Leftrightarrow\ \Phi$. In the example above, the termination term will be:

$$rem - y.$$

In the case of a 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:

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

In the above example, we worked with recursive equations whose behaviour does not depend on other equations. For the case when a critical variable of a recursive statement is influenced also by some other recursive statements, our method is still applicable. In this situation, first we work with that equation (critical variable) which does not depend on other equations, generate the closed–form of it, and then substitute this expression in the other recursive equations. Proceeding in a similar manner for the other recursive equations, we will solve again first–order recursive equations by the Gosper–algorithm.

Also, when the obtained recurrence is not Gosper–summable, but it is of the form:

$$x_n = t * x_{n-1}^c,$$

where t is a term that does not depend on x, and $c \in \mathbb{N}$, the closed form of the detected recurrence can be solved by our recurrence solving package. The presented strategy works only if we do not have mutual recurrence(s) in the loop body.

1.3 Mutual Recurrences

We use the technique of *generating functions* from combinatorics [29, 25], and we demonstrate it here by an example.

$$
\begin{aligned}
&Specification[\text{"}Fibonacci\text{"},\ FibonacciProcSpec[\downarrow n, \uparrow F],\\
&\quad Pre \to (n \geq 0),\\
&\quad Post \to (F = FibExp[n])]
\end{aligned}
$$

$$
\begin{aligned}
&Program[\text{"}Fibonacci\text{"},\ FibonacciProc[\downarrow n, \uparrow F],\\
&\quad Module[\{H,\ i\},\\
&\qquad i\ :=\ n;\\
&\qquad F\ :=\ 1;\\
&\qquad H\ :=\ 1;\\
&\qquad WHILE[i\ >\ 1,\\
&\qquad\quad H\ :=\ H\ +\ F;\\
&\qquad\quad F\ :=\ H\ -\ F;\\
&\qquad\quad i\ :=\ i\ -\ 1]]]
\end{aligned}
$$

Note: $FibExp[n]$ denotes the term: $F_n \ = \ \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}}$, where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi}$ is its conjugate.
From the loop–body, we can set up the recurrence:

$$H_n \ = \ H_{n-1} \ + \ F_{n-1} \ + \ [n = 1], (n \in Z),\ H_0 = 0$$

$$F_n \ = \ H_n \ - \ F_{n-1}\ (n \in Z),\ F_0 = 0$$

where the value of $[n = 1]$ is 1 (i.e. H_1) when $n = 1$, and 0 when $n > 1$.

We seek a closed form for (F_n) and (H_n), using an extension of the Mallinger's Mathematica package GeneratingFunctions [16], which was developed in the Combinatoric Group of RISC. Then, by applying the *generating functions* technique, we obtain the harmonic forms:

$$
\begin{aligned}
H(z) \ &= \ \sum_n H_n z^n \ = \ \sum_n H_{n-1} z^n \ + \ \sum_n F_{n-1} z^n \ + \ \sum_n [n = 1] z^n \ (n \in Z)\\
&= \ zH(z) + zF(z) + z\\
F(z) \ &= \ \sum_n H_n z^n \ - \sum_n F_{n-1} z^n \ (n \in Z)\\
&= \ H(z) \ - \ zF(z)
\end{aligned}
$$

Solving this system in the unknowns F and H, we obtain the generating functions:

$$F(z) \ = \ \frac{z}{1 - z - z^2}, \quad H(z) \ = \ \frac{z(1+z)}{1 - z - z^2}$$

and by combinatorial techniques their closed form:

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

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: $i_k = n - (k-1)$. Now we eliminate k from the three equations, obtaining:

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

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

1.4 Further Work

We have been able to automatically generate verification conditions (including invariants and termination terms) for several interesting programs, like: computation of the integer square root, sum of integers, square calculation, binary powering, etc. These experiments show the power and also the limitations of the methods we have implemented.

One further development which we are now studying is the generation of invariants for nested WHILE loops. The main idea is to characterise the behaviour of the WHILE loops using recurrence equations. Namely, we first generate the closed form for the critical variables of the innermost loop. These can be used in order to express the critical variables of the outer loop, and finally we try to obtain the invariant relations by elimination of the loop index variables.

Furthermore we 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 [2, 3], namely to generate the Gröbner bases of the already obtained closed–forms (this approach has been investigated in [11, 27, 17]).

2 Functional Programs

While proving [partial] correctness of non-recursive procedural programs is quite well understood, for instance by using Hoare Logic [8, 15], there are relatively few approaches to recursive procedures (see e.g. [21] Chap. 2).

We discuss here a practical approach, based on a certain theory and including implementation, to automatic generation of verification conditions for functional recursive programs. The implementation is part of the *Theorema* system, and complements the research performed in the *Theorema* group on verification and synthesis of functional algorithms based on logic principles [1, 4, 24].

We consider the correctness problem expressed as follows: *given* the program (by its source text) which computes the function F and given its specification by a precondition on the input $I_F[x]$ and a postcondition on the input and the output $O_F[x, y]$, *generate* the verification conditions which are [minimally] sufficient for the program to satisfy the specification.

For simplifying the presentation, we consider here the "homogeneous" case: all functions and predicates are interpreted over the same domain. Proving the verification conditions will require the *specific theory* relevant to this domain and to the auxiliary functions and predicates which occur in the program.

The functional program of F can be interpreted as a set of predicate logic formulae, and the correctness of the program can be expressed as:

$$(\forall x \ : \ I_F[x]) O_F[x, F[x]], \tag{1}$$

which we will call the *correctness formula* of F. Under certain additional assumptions, for program correctness it is sufficient that the correctness formula is a logical consequence of the formulae corresponding to the definition of the function (and the specific theory which describes the properties of the domain[s] and the auxiliary functions involved). This approach was previously used by other authors and is also experimented in the Theorema system [1]. However, the proof of (1) may be difficult, because the prover has to find the appropriate induction principle and has to find out how to use the properties of the auxiliary functions present in the program.

The method presented in this section generates several verification conditions, which are easier to prove. In particular, only the termination condition needs an inductive proof, and this termination condition is "reusable", because it basically expresses an induction principle which may be useful for several programs. This is important for automatic verification embedded in a practical verification system, because it leads to early detection of bugs (when proofs of simpler verification conditions fail).

Moreover, the verification conditions are provable in the frame of predicate logic, without using any theoretical model for program semantics or program execution, but only using the theories relevant to the predicates and functions present in the program text. This is again important for the automatic verification, because any additional theory present in the system will significantly increase the proving effort.

We start by developing a set of rules for generating verification conditions, for programs having a particular structure. The rules for partial correctness are developed using Scott induction and the fixpoint theory of programs, however the verification conditions themselves do not refer to this theory, they only state facts about the predicates and functions present in the program text. In particular, the termination condition consists in a property of a certain simplified version of the original program.

We approach the correctness problem by splitting it into two parts: *partial correctness* (prove that the program satisfies the specification provided it terminates), and *termination* (prove that the program always terminates). Proving *partial correctness* may be achieved by Scott induction [9, 28, 18, 22] – a detailed description of the method for a certain class of functional programs is presented in [19].

2.1 Simple Recursive Programs.

Consider the definition (or program):

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, F[R[x]]], \tag{2}$$

where Q is a predicate[2] and S, C, R are auxiliary functions (S is a "simple" function, C is a "combinator" function, and R is a "reduction" function). We assume that S, C, R are already given together with their input and output conditions $I_S[x]$, $O_S[x,y]$, $I_C[x,y]$, $O_C[x,y,z]$, $I_R[x]$, $O_R[x,y]$ and assumed to satisfy their correctness formulae (as in (1)). Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

We consider that the definition (2) is the abbreviation of the conjunction of the logical formulae:

$$(\forall x)\ (Q[x] \Rightarrow (F[x] = S[x])), \tag{3}$$

$$(\forall x)\ ((\neg Q[x]) \Rightarrow (F[x] = C[x, F[R[x]]])). \tag{4}$$

We are working in first–order predicate logic and in the context of a certain *local theory*, that is a theory containing the definition of F (above) and all the necessary properties of the predicates and functions which are used in the program, as well as the correctness formulae for the subroutines S, C, R.

Using Scott induction in the fixpoint theory of functions, one obtains the following verification conditions for the *partial correctness* of the program of F:

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]])$$

$$(\forall x, y : I_F[x])\ (\neg Q[x] \implies I_F[R[x]])$$

$$(\forall x, y : I_F[x])\ ((\neg Q[x] \wedge O_F[R[x], y]) \implies O_F[x, C[x, y]])$$

(In fact, the conditions can be further decomposed using the preconditions and the postconditions of the auxiliary functions, see [10].)

First, we need to ensure the *termination* of the calls to the auxiliary functions. Let I_S, I_C and I_R be the preconditions of S, C, R respectively. The verification conditions are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x])$$

---

[2] In practice Q is also implemented by a program, and it may also have an input condition, but we do not want to complicate the present discussion by also including this aspect, which has a special flavour.

$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_R[x])$$

$$(\forall x, y : I_F[x])\ ((\neg Q[x] \wedge O_F[R[x], y]) \implies O_C[x, y]).$$

The condition for the *termination* of the program can be expressed using a simplified version of the initial function:

$$F'[x] = \textbf{ If } Q[x] \textbf{ then } 0 \textbf{ else } F'[R[x]]$$

which only depends on Q and R. Namely, the verification condition is

$$(\forall x : I_F[x])\ F'[x] = 0,$$

which must be proven based on the logical formulae corresponding to the definition of F' (and the local theory relevant to the program). The condition follows from the equivalence of the termination properties of F and F', which can be proven in the fixpoint theory of functions e. g. by using induction on the number of recursion steps (see [23]).

This method can be easily extended to programs using **Case** (**If–then–else** with several cases), as it is illustrated in the example below.

2.2 Example

For illustrating the method we give here an example of a *binary powering* program annotated with its specification. Let be the program:

$$\begin{aligned} P[x, n] = \ & \textbf{If } n = 0 \textbf{ then } 1 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P[x * x, n/2] \\ & \textbf{else } x * P[x * x, (n-1)/2]. \end{aligned}$$

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n.$$

The (automatically generated) verification conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \implies 1 = x^n) \tag{5}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \mathrm{Even}[n] \implies n/2 \in \mathbb{N}) \tag{6}$$

$$(\forall x, n, m : n \in \mathbb{N})\ (n > 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \implies m = x^n) \tag{7}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \neg\mathrm{Even}[n] \implies (n-1)/2 \in \mathbb{N}) \tag{8}$$

$$(\forall x, n, m : n \in \mathbb{N})\ (n > 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2} \implies x * m = x^n) \tag{9}$$

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \implies \mathbb{T}) \tag{10}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \mathrm{Even}[n] \implies \mathrm{Even}[n]) \tag{11}$$

$$(\forall x, n : n \in \mathbb{N})\ (n > 0 \wedge \neg\mathrm{Even}[n] \implies \mathrm{Odd}[n]) \tag{12}$$

$$(\forall x, n, m : n \in \mathbb{N})\ (n > 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \implies \mathbb{T}) \tag{13}$$

$$(\forall x, n, m : n \in \mathbb{N})\ (n > 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2} \implies \mathbb{T}) \tag{14}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = 0, \tag{15}$$

where

$$\begin{aligned} P'[x,n] = \ & \textbf{If } n = 0 \textbf{ then } 0 \\ & \textbf{elseif } \mathrm{Even}[n] \textbf{ then } P'[x * x, n/2] \\ & \textbf{else } P'[x * x, (n-1)/2]. \end{aligned}$$

One sees that the formulae (6), (9) and (10) are trivial to prove. The formulae (1) to (5), (7) and (8) are relatively easy to prove. The only formula which needs an induction proof (on the second argument n) is (11).

Currently we are working on extending the method to more general program schemes, such that they include multiple recursive calls, mutual recursive functions, etc.

3 Forward Verification

Examining the verification conditions generated by the methods presented in the previous sections, one may notice that they correspond to certain common–sense assertions. (e. g. that the input of an auxiliary function should satisfy its input conditions).

In this section we present an extension of the approach introduced in [10], which consists in generating the verification conditions using such "intuitive" principles, by following in a *forward* manner the steps of the (virtual) execution of the program. We formalise these principles in a precise method and we then prove that the correctness formula is a logical consequence of the generated verification conditions. Remarkably, the proof of this fact does not need an additional model for the notion of computation – it is sufficient to interpret the program text as a set of logical formulae in the frame of predicate logic with equality.

We will explain the method by demonstrating it on a simple example. Consider the definition (2), with the same assumptions about the auxiliary functions as in the previous section.

3.1 Partial Correctness

The verification conditions are generated using the following principles:

- check the input conditions when calling subroutines;
- accumulate all reasonable assumptions (coming from the input condition of the main function, from **if–then–else** conditions and from the correctness formulae of the subroutines),
- try to finally obtain the correctness property for the output of F.

Since any input to F, in case it satisfies Q, is passed to S, the first verification condition is:

$$(\forall x : I_F[x])\ ((I_F[x] \wedge Q[x]) \Rightarrow I_S[x]). \tag{16}$$

That is, starting from assumptions $I_F[x]$ and $Q[x]$, one has to prove $I_S[x]$. Note that $O_S[x, S[x]]$ is a logical consequence of $I_S[x]$ and the correctness formula of S (which is assumed).

Finally on this program branch, one has to prove the "correctness" of the result $F[x]$:

$$(\forall x : I_F[x])\ ((Q[x] \wedge I_S[x] \wedge O_S[x, S[x]]) \Rightarrow O_F[x, S[x]]). \tag{17}$$

Similarly, for the other case:

$$(\forall x : I_F[x])\ (\neg Q[x] \Rightarrow I_R[x]), \tag{18}$$

and then one also has $O_R[x, R[x]]$. Next "execution" step would be to use $R[x]$ as input to F, thus one needs:

$$(\forall x : I_F[x])\ ((\neg Q[x] \wedge I_R[x] \wedge O_R[x, R[x]]) \Rightarrow I_F[R[x]]). \tag{19}$$

Note that each verification condition accumulates (in the antecedents of the implication) the postcedents proved in the previous conditions, as well as the facts which can be obtained by using the correctness of the subroutines. Thus we will not list from now on the full verification conditions, but only mention the new antecedents and the next postcedent.

The next verification condition is created using the same principles on F as for any other subroutine. That is, we add to the antecedents both $I_F[R[x]]$ (postcedent of the previous verification condition), but also $O_F[R[x], F[R[x]]]$. The new postcedent corresponds to the next "execution" step, i. e. states that the input condition of C is satisfied:

$$(\forall x : I_F[x])\ ((\ldots \wedge I_F[R[x]] \wedge O_F[R[x], F[R[x]]])\ \Rightarrow I_C[x, F[R[x]]]), \tag{20}$$

and then we may assume $O_C[x, F[R[x]], C[x, F[R[x]]]]$, and finally we must prove the "correctness" of the output:

$$(\forall x : I_F[x])\ ((\ldots \wedge I_C[x, F[R[x]]] \wedge O_C[\ldots]) \Rightarrow O_F[x, C[x, F[R[x]]]]). \tag{21}$$

We may say that these 6 verification conditions ensure *partial correctness*. Namely, the following two formulae, which we will call *partial correctness formulae* are logical consequences of the verification conditions and the *local knowledge* (definition of F, properties of additional functions and predicates, including the correctness formulae for S, C, and R):

$$(\forall x : I_F[x])\ (Q[x] \Rightarrow O_F[x, F[x]]), \tag{22}$$

$$(\forall x : I_F[x])\ (\neg Q[x] \wedge (I_F[R[x]] \Rightarrow O_F[R[x], F[R[x]]])\ \Rightarrow\ O_F[x, F[x]]). \tag{23}$$

3.2 Total Correctness

Termination. Intuitively, the program defined by (2) terminates if any element x satisfying I_F also has the following property: x satisfies Q, or, by repeated applications of R, it is reduced to an element satisfying Q. We may formalise this property as a predicate P which fulfils: $(Q[x] \vee P[R[x]]) \Rightarrow P[x])$ for any x. Let us choose as *termination condition* for the program of F the following formula:

$$((\forall x : I_F[x]) \ ((Q[x] \vee P[R[x]]) \Rightarrow P[x])) \ \Rightarrow (\forall x : I_F[x]) P[x].$$

Here P is a new constant symbol, thus this formula holds for any P (in second order logic). Hence, the termination condition is in fact an induction principle, namely that induction principle which is appropriate for the particular program which is verified. Typically, the proof of this formula will require an induction over the domain of F (possibly specified in I_F).

Total Correctness. If we replace $P[x]$ by $I_F[x] \Rightarrow O[x, F[x]]$ then we obtain (after some simple equivalent transformations of some inner formulae) an implication whose antecedents are the two partial correctness conditions (22) and (23) and whose postcedent is the correctness formula (1). Thus, *the correctness formula is a logical consequence of the partial correctness conditions and the termination condition*, of course by using the local knowledge. This is quite remarkable, because it shows that the verification conditions ensure the total correctness of the program, and for proving this we do not need any additional model for defining the notion of computation, termination, etc., but everything is done in the frame of the local theory.

3.3 General Principle

A tuple with $m - n + 1$ elements $\langle a_n, a_{n+1}, \ldots, a_{m-1}, a_m \rangle$ will be denoted by $\langle a_i \rangle_{i=n}^{m}$ (the empty tuple $\langle\rangle$ when $n > m$). We use a similar indexed notation for denoting finite sets, finite conjunctions and finite disjunctions (empty conjunction is true, empty disjunction is false). We will also consider that the combinator functions C are applied to an element and a tuple: $C[x, \langle \ldots \rangle]$.

We define *cumulative conditions* $\widehat{Q}_i[x]$ of $\{Q_i[x]\}_{i=1,\ldots}$ as

$$Q_i[x] \wedge \bigwedge_{k=1}^{i-1} \overline{Q_k[x]},$$

the *correctness predicate* of a function G as $K_G[x]$ expressed by $I_G[x] \Rightarrow O_G[x, G[x]]$, and the *cumulated correctness condition* of a function G as $H_G[x]$ expressed by $I_G[x] \wedge O_G[x, G[x]]$.

Let us consider programs which can be expressed as:

$$F[x] = \left\langle C_i[x, \langle F[R_{ij}[x]] \rangle_{j=1}^{m_i}] \ \text{ if } Q_i[x] \right\rangle_{i=1,n}.$$

We assume that the necessary properties (including correctness formulae) of the auxiliary functions $\{Q_i, C_i, \{R_{ij}\}_{j=1}^{m_i}\}_{i=1}^{n}$ are present in the local knowledge.

The logical formulae corresponding to the definition of F are (for $i = 1, n$):

$$\widehat{Q}_i[x] \;\Rightarrow\; F[x] = C_i[x, \langle F[R_{ij}[x]]\rangle_{j=1}^{m_i}]$$

We list now the general form of the verification conditions, which are all universally quantified by $(\forall x : I_F[x])$ (omitted for simplicity). The verification conditions expressing that the inputs to R_{ij} must be appropriate are (for $i = 1, n$ and $j = 1, m_i$):

$$\widehat{Q}_i[x] \;\Rightarrow I_{R_{ij}}[x].$$

Then one may assume that the outputs of R_{ij} are correct and one has to show that the inputs to F are appropriate (for $i = 1, n$ and $j = 1, m_i$):

$$(\widehat{Q}_i[x] \wedge H_{R_{ij}}[x]) \Rightarrow I_F[R_{ij}[x]],$$

(where H is the cumulated correctness condition defined earlier). Next one assumes that the outputs of F are correct and one has to show that the inputs to C_i are appropriate (for $i = 1, n$):

$$\left(\widehat{Q}_i[x] \wedge \bigwedge_{j=1}^{m_i} (H_{R_{ij}}[x] \wedge H_F[R_{ij}[x]])\right) \Rightarrow I_{C_i}[x, \langle F[R_{ij}[x]]\rangle_{j=1}^{m_i}],$$

Finally, one assumes that the outputs of C_i are correct and one has to prove that the outputs of F are correct (for $i = 1, n$):

$$\left(\widehat{Q}_i[x] \wedge \bigwedge_{j=1}^{m_i} (H_{R_{ij}}[x] \wedge H_F[R_{ij}[x]]) \wedge H_{C_i}[x, \langle F[R_{ij}[x]]\rangle_{j=1}^{m_i}]\right)$$
$$\Rightarrow O_F[x, C[x, \langle F[R_{ij}[x]]\rangle_{j=1}^{m_i}]].$$

The partial correctness formulae ($i = 1, n$) have the form

$$\left(\widehat{Q}_i[x] \wedge \bigwedge_{j=1}^{m_i} K_F[R_{ij}[x]]\right) \;\Rightarrow O_F[x, C_i[x, \langle F[R_{ij}[x]]\rangle_{j=1}^{m_i}]]$$

and they are logical consequences of the local theory and of the verification conditions for partial correctness.

The termination condition (with explicit quantifiers) is:

$$\left((\forall x : I_F[x]) \left(\bigvee_{i=1}^{n} \widehat{Q}_i[x] \wedge \bigwedge_{j=1}^{m_i} P[R_{ij}[x]]\right) \Rightarrow P[x]\right) \;\Rightarrow\; ((\forall x : I_F[x]) P[x])$$

It is relatively straightforward to prove that the correctness formula of F is a logical consequence of the termination condition and the partial correctness formulae, together with the local theory.

This general method is implemented as an automatic code analyser and verification condition generator for functional programs (and program schemata). The verification conditions of the concrete example presented in this section have been generated automatically using this implementation.

Similar principles can be applied to imperative programs, and this is work in progress. Imperative programs contain loops, and they can be treated either as recursive calls, either by adding special variables as loop counters.

Conclusions and Further Work

Using several complementary approaches to the generation of verification conditions, we obtain more insight into the issue of practical algorithm verification in the context of automatic reasoning. Moreover, we obtain interesting proof problems for the [semi–]automatic provers of the *Theorema* system, which allow us to detect the possible weaknesses of the current provers and to design and implement new proving methods.

Further work includes the extension of these methods to more complex programs, as well as a more systematic comparison and cross–fertilisation of the different approaches to the generation of verification conditions both for functional and for imperative programs.

References

1. A. Craciun; B. Buchberger. Functional program verification with theorema. In *CAVIS-03 (Computer Aided Verification of Information Systems)*, Institute e-Austria Timisoara, February 2003.
2. B. Buchberger. *Ein algorithmisches Kriterium für die Lösbarkeit 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.
3. 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.
4. B. Buchberger. Verified algorithm development by lazy thinking. In *IMS 2003 (International Mathematica Symposium)*, Imperial College, London, July 2003.
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. R. W. Gosper. *Decision procedures for indefinite hypergeometric summation. Journal of Symbolic Computation*, 75:40–42, 1978.
8. C. A. R. Hoare. *An axiomatic basis for computer programming. Comm. ACM*, 12, 1969.

9. D. Scott J. W. de Bakker. A theory of programs. In *IBM Seminar*, 1969. Vienna, Austria, 1969.
10. T. Jebelean. Forward verification of recursive programs. In *Computer Aided Verification of Information Systems (CAVIS-04)*, Institute e-Austria Timisoara, February 2004. Technical report 04-01, Institute e-Austria Timisoara (`www.ieat.ro`).
11. 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.
12. M. Kirchner. Program verification with the mathematical software system Theorema. Technical Report 99-16, RISC-Linz, Austria, 1999. PhD Thesis.
13. D. E. Knuth. *The Art of Computer Programming, volume 2 / Seminumerical Algorithms.* Addison-Wesley, 2nd edition, 1969.
14. L. Kovács. *Program Verification using Hoare Logic.* In *Computer Aided Verification of Information Systemsm Romanian-Austrian Workshop*, 2003. Timisoara, Romania, February 2003.
15. B. Buchberger; F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German).* Springer, 2nd edition, 1981.
16. C. Mallinger. Algorithmic manipulations and transformations of univariate holonomic functions and sequences. Master's thesis, RISC, J. Kepler University, Linz, August 1996.
17. 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.
18. Z. Manna. *Mathematical Theory of Computation.* McGraw-Hill Inc., 1974.
19. T. Jebelean N. Popov. A practical approach to verification of recursive programs in theorema. In *Proceedings of the 5th International Workshop on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2003)*, 2003. Timisoara, Romania, 1–4 October 2003.
20. R. L. Graham; D. E. Knuth; O. Patashnik. *Concrete Mathematics, 2nd ed.* Addison-Wesley Publishing Company, 1989. pg. 306-330.
21. M. C. Paull. *Algorithm Design. A recursion transformation framework.* Wiley, 1987.
22. N. Popov. Operators in Recursion Theory. Technical Report 03-06, RISC-Linz, Austria, 2003.
23. N. Popov. Verification of Simple Recursive Programs: Sufficient Conditions. Technical Report 04-06, RISC-Linz, Austria, 2004.
24. T. Jebelean; L. Kovacs; N. Popov. Verification of imperative programs in Theorema. In *1st South-East European Workshop in Formal Methods (SEEFM03)*, 2003. Thessaloniki, Greece, 20 November 2003.
25. 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.
26. P. Paule; M. Schorn. *A Mathematica version of Zeilberger's algorithm for proving binomial coefficient identities. Journal of Symbolic Computation*, 20(5-6):673–698, 1995.
27. M. Müller-Olm; H. Seidl. *Polynomial Constants are Decidable.* In *Static Analysis Symposium (SAS 2002), vol.2477 of LNCS*, 2002. pp. 4-19.
28. J. Loeckx; K. Sieber. *The Foundations of Program Verification.* Teubner, second edition, 1987.

29. R. P. Stanley. Differentiably finite power series. *European Journal of Combinatorics*, 1:175–188, 1980.
30. S. Wolfram. *The Mathematica Book. Version 5.0.* Wolfram Media, 2003.