Combining Logic and Algebraic Techniques for Program Verification in *Theorema*

Laura Kovács, Nikolaj Popov, Tudor Jebelean
Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria,
Institute e–Austria, Timişoara, Romania
Email: {kovacs,popov,jebelean}@risc.uni-linz.ac.at

***Abstract*— We study and implement concrete methods for the verification of both imperative as well as functional programs in the frame of the *Theorema* system. The distinctive features of our approach consist in the automatic generation of loop invariants (by using combinatorial and algebraic techniques), and the generation of verification conditions as first–order logical formulae which do not refer to a specific model of computation.**

I. Introduction

Our research aims at obtaining useful theoretical insights and achievement of concrete practical progress by studying various methods for computer aided verification of both imperative and functional programs.

This work is performed in the frame of the *Theorema* system – `www.theorema.org`, a mathematical computer assistant which aims at supporting all phases of mathematical activity: construction and exploration of mathematical theories, definition of algorithms for problem solving, as well as experimentation and rigorous verification of them. *Theorema* provides both functional as well as imperative programming constructs. Moreover, the logical verification conditions which are produced by the methods presented here can be passed to the automatic provers of the system in order to be checked for validity. The system includes a collection of general as well as specific provers for various interesting domains (e. g. integers, sets, reals, tuples, etc.).

Imperative Programs. Our approach is based on the Floyd-Hoare-Dijkstra's inductive assertion method [11], [15], [8]. We focus in this paper on the automatic generation of the inductive (equality) assertions for loops in imperative programs, by using algorithmic combinatorics and computer algebra. Starting from the recurrence equations of the loop variables, we generate the closed forms by combinatorial techniques, and then we find the invariant properties as polynomial equations, by eliminating the loop index(es) using algebraic methods. These invariants are further used for the generation of verification conditions by the weakest precondition method.

It is well known that generation of loop invariants is in fact the challenging part of the Floyd-Hoare-Dijkstra method. Early attempts [9], [12], [18] treat cases where only few arithmetic operations (mainly additions) among program variables are involved. Significant advances have been achieved during the last decade by using computer algebra: [32], [1], [6], [5], [17], [26], [30], [28].

Some distinguishing features of our method are: the use of advanced symbolic summation techniques for finding the closed form of the loop variables (see below) and the use of Gröbner Bases for identifying the essential set of the invariant equality properties (by the Buchberger algorithm – see [2]).

Gröbner basis is also used in [26], [30], [28], but in this approach one must fix apriori the degree of the seeked polynomial. Our method does not need this, and in fact generates all the essential polynomial invariants. We use, in addition to algebraic computations, techniques from symbolic summation (Gosper algorithm [14], generating functions [31], [29], C-finite solving [10], algebraic dependencies [19]). Although the method is applicable only when certain restrictions on the loop syntax (see subsection II-C) are fulfilled, a large class of programs of practical interest can be handled.

The main contribution of this paper (w. r. t. imperative verification) is the identification and precise characterization of a certain class of loops *P–solvable loops*, for which our method will always find all existing polynomial invariants (that is, the generator of the corresponding ideal). We demonstrate the method on a concrete example involving loops with conditionals, and further interesting examples are presented in [22].

Functional Programs. Proving correctness of non-recursive procedural programs is broadly discussed and well understood in the literature, for instance by using Hoare Logic [15], [4]. However, there are relatively few approaches to recursive procedures (see e.g. [27] Chap. 2).

As usual, program correctness is transformed into a set of first-order predicate logic formulae by a Verification Condition Generator (VCG) – a device, which takes the program (its source text) and the specification (precondition and post-condition) and produces several verification conditions. As a distinctive feature of our method, these formulae do not refer to a theoretical model for program semantics or program execution, but only to the theory of the domain used in the program. This is very important for the automatic verification, because any additional theory present in the system will significantly increase the proving effort.

However, there is no "universal" VCG, due to the fact that proving program correctness is undecidable in general. On

978-0-7695-3071-0/07 $25.00 © 2007 IEEE
DOI 10.1109/ISoLA.2006.46

the other hand, in practice, proving program's correctness is possible in many particular cases and, therefore, many VCG-s have been developed for serving a big variety of situations. Our research is contributing exactly in this direction.

Our method is designed for simple recursive programs (multiple choice *if-then-else* with at most one recursive call on each branch) – which are the most used in practice. We furthermore define a specific class of such programs, the *coherent* ones, which have the property that each function call is applied to arguments obeying the respective input specification. Note that this is also a very natural constraint, and it already gives a set of verification conditions. For such coherent programs we are able to define a necessary and sufficient set of verification conditions, thus our condition generator is not only sound, but also *complete*. This distinctive feature of our method is very useful in practice for program debugging – as we also demonstrate by an example.

II. Procedural Programs

A. Theoretical Preliminaries

We start with a brief presentation of the necessary properties and techniques from polynomial algebra and algebraic combinatorics (for additional details see [7], [10], [19]).

We denote by $\mathbb{K}$ a field with characteristic zero (like e.g. $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$), and by $\bar{\mathbb{K}}$ we denote the algebraic closure of $\mathbb{K}$ (e.g. $\mathbb{C}$ is the algebraic closure of $\mathbb{R}$).

Polynomial assertions. Ideals:

Definition 2.1: Polynomial Assertions.
A *polynomial assertion* over $\mathbb{K}$ is a formula

$$p_1(x_1,\ldots,x_n)=0\wedge\ldots\wedge p_m(x_1,\ldots,x_n)=0,$$

where $p_i(x_1,\ldots,x_n),\ i=\overline{1,m}$ is a multi–variate polynomial in the program variables $x_1,\ldots x_m$, i.e. $p_i\in\mathbb{K}[x_1,\ldots x_m],\ i=\overline{1,m}$.

Definition 2.2: Ideals.
Let R be a ring. A set $\mathbf{A}\subseteq R$ is called an *ideal* in R iff for all $a_1,a_2,a\in\mathbf{A},p\in R$ we have $a_1+a_2\in\mathbf{A}$ and $pa\in\mathbf{A}$. We write $\mathbf{A}\trianglelefteq R$ to denote that $\mathbf{A}$ is an ideal in R.

For $p_1,\ldots,p_r\in R$ we denote by $\langle p_1,\ldots,p_r\rangle$ the smallest ideal containing $p_1,\ldots,p_r$. If $\mathbf{A}=\langle p_1,\ldots,p_r\rangle$, we say that $\mathbf{A}$ is *generated* by $p_1,\ldots,p_r$ and that $p_1,\ldots,p_r$ is a *basis* of $\mathbf{A}$.

C–finite sequences:

Definition 2.3: Linear and C-finite recurrences.

1) A *linear recurrence* in $\mathbb{K}$ is a recurrence of the form

$$\begin{aligned}f(n+r)=\ & a_0(n)f(n)+a_1(n)f(n+1)+\ldots+\\ & a_{r-1}(n)f(n+r-1)+a(n),\quad (n\geq 1)\end{aligned}$$

The value *r* is called the *order* of the recurrence.
The recurrence is called *homogeneous* if $a(n)=0$ and *inhomogeneous* otherwise.

2) A homogeneous linear recurrence is called *C–finite* if $a_1(n),\ldots,a_{r-1}(n)$ are constants, i.e. independent of n.

A crucial and elementary fact about C–finite sequences is that they always admit a closed form solution (i.e. a solution that is expressed, without recursion, as a function of the recurrence index) [10].

Proposition 2.1: Closed Form of C–finite Sequences.
The closed form of a C–finite sequence $f(n)$ is:

$$f(n)=p_1(n)\phi_1^n+\cdots+p_s(n)\phi_s^n,\qquad (1)$$

where $\phi_1,\ldots,\phi_s\in\bar{\mathbb{K}}$ are the roots of the characteristic polynomial [10] of the C–finite sequence $f(n)$.

(The proof of this proposition can be found in [10], [22].)

Remark 2.1: For solving C-finite recurrences in our invariant generation algorithm, we use the `SumCracker` package [19] implemented in *Mathematica* by the Combinatorics group of RISC-Linz.

Algebraic Dependencies:

Definition 2.4: Algebraic Dependencies.
Let $f_1(n),\ldots,f_s(n)$ be sequences in $\mathbb{K}$.
An *algebraic dependency* (or *algebraic relation*) [19] of these sequences is a polynomial p such that

$$p(f_1(n),\ldots,f_1(n+r_1),\ldots\ldots,f_s(n),\ldots,f_s(n+r_s))=0,\quad (2)$$

where $r_1,\ldots,r_s$ are the orders of $f_1(n),\ldots,f_s(n)$, respectively.

A complete study of finding algebraic dependencies among sequences $\theta_1^n,\ldots,\theta_s^n$ (i.e. exponentials), with $\theta_1,\ldots,\theta_s\in\bar{\mathbb{K}}$, can be found in [19]. In this paper, let us state only the properties on which our algorithm relies on:

Remark 2.2: In the case when $\theta_1,\ldots,\theta_s$ are algebraic numbers, then, if they are related algebraically, one can determine automatically all their algebraic dependencies [19]. For obtaining automatically the needed algebraic dependencies, we use the `Dependencies` package [19] implemented in *Mathematica* by the RISC combinatorics group.

Remark 2.3: In the case of our algorithm, at the current stage, we restrict our work to rational exponential sequences, i.e. $\theta_1,\ldots,\theta_s\in\bar{\mathbb{Q}}$. Thus, using Remark 2.2, we are able to determine automatically the existing algebraic dependencies among $\theta_1,\ldots,\theta_s$.

Definition 2.5: Annihilating Ideal.[19]
For any sequences $f_1(n),\ldots,f_m(n)$ over $\mathbb{K}$, we call the ideal of their algebraic dependencies the *annihilating ideal* (or *annihilator*) of these sequences.

B. Imperative Program Verification

Programming Language: We have developed and integrated in the *Theorema* system an environment for imperative program verification. The user interface has four main (intuitive) constructors[21], [16]:

1) `Specification`: Pre- and postcondition of the program, written in first-order predicate logic;
2) `Program`: Programs are considered as procedures, with input, transient and output values;
3) `VCG`: A predicate transformer, for generating verification conditions in the language of first-order predicate logic;
4) `Execute`: Testing the program, with concrete input values.

Our programming language supports: assignments, blocks, conditionals, `For` and `While` loops, procedure calls.

Verification Environment: The verification of programs consists of two main parts, namely:
Generation of verification conditions: In order to verify imperative programs, we automatized the Floyd–Hoare–Dijkstra method, namely the weakest precondition strategy [11], [8], and it is implemented as `VCG`. In more detail, the `VCG` takes a program and its specification, and working recursively bottom–up on the program syntax, using a predicate transformer, the program is "eliminated", and a collection of purely logical, universally quantified proof situations remain (in *Theorema* syntax). These proof obligations are the so–called *verification conditions*, that one has to prove in order to insure the correctness of the considered program. The automated generation of loop invariants is performed in this phase;
Proving verifications conditions: The automatically obtained verification conditions are fed into the available provers of the *Theorema* system. In most cases the `PCS` prover of *Theorema* is applied [3], that uses quantifier elimination, and produces human–readable proofs of the verification conditions (i.e. program correctness is proved), or failures in case the program is not correct. The proving part of the verification process is not in the scope of the current paper. However, it is worth to be mentioned, that in many examples the generated verification conditions were proven automatically, by calling `PCS`.

C. A Method for Polynomial Invariant Generation

An important task in imperative program verification is to automatically infer loop invariants and termination terms (or ranking functions).

In this paper we focus on automated loop invariant generation. Our method combines computer algebra and algorithmic combinatorics with formal methods and computational logic, in such a way that the end of the invariant generation process valid polynomial assertions (if the imperative loop has such assertions) are obtained. For doing so, we introduce the notion of a *P-solvable loop*. The main result of this paper is the theoretical and practical presentation of the fact that if an imperative program contains P–solvable loops and if the loop has polynomial invariant properties, our algorithm will find (all of) them.

In this paper we extend an earlier conference paper [23] by allowing C-finite assignment statement in the loop body. Thus, the considered imperative loops contain conditional statements, with the property that among the assignment statements there are some C-finite assignments (and/or Gosper-summable recurrences, geometric series). This work relies also on results from [16], where treatment of loop with only (Gosper-summable) assignment statements was considered.

P-Solvable Imperative Loops:

Definition 2.6: P-Solvable Imperative Loops.
An imperative loop is called *P-solvable* iff

- its assignment statements are either Gosper-summable [14], [16], geometric series (i.e. special cases of C-finite) or C-finite recurrences;
- its recursively changed variables $x_1, \dots, x_m$ have their closed forms of the following nature:

$$\begin{cases} x_1(n) = & p_{1,1}(n)\theta_{1,1}^n + \cdots + p_{1,s_1}(n)\theta_{1,s_1}^n \\ x_2(n) = & p_{2,1}(n)\theta_{2,1}^n + \cdots + p_{2,s_2}(n)\theta_{2,s_2}^n \\ \vdots & \\ x_m(n) = & p_{m,1}(n)\theta_{m,1}^n + \cdots + p_{m,s_m}(n)\theta_{m,s_m}^n \end{cases}, \quad (3)$$

where:

1) n is the loop counter;
2) $x_i(n)$ $(1 \le i \le m)$ represents the value of x_i at iteration n;
3) $p_{1,1}, \dots, p_{1,s_1}, \dots\dots, p_{m,1}, \dots, p_{m,s_m} \in \bar{\mathbb{K}}[n]$;
4) $\theta_{1,1}, \dots, \theta_{1,s_1}, \dots\dots, \theta_{m,1}, \dots, \theta_{m,s_m} \in \bar{\mathbb{K}}$;
5) there exist algebraic dependencies among $\theta_{1,1}^n, \dots, \theta_{1,s_1}^n, \dots\dots, \theta_{m,1}^n, \dots, \theta_{m,s_m}^n$.

Theoretical Background of the Proposed Approach: In this section we show the theoretical background of our algorithm which, in the case when a P–solvable loops admits polynomial invariants, finds *all* these polynomial assertions. We give only a brief theoretical illustration of our algorithm, more details can be found in [22].

Algorithm: Finding polynomial invariants of P-solvable Loop
We consider a P-solvable loop, having $x_1, \dots, x_m$ as its recursively changed loop variables . The main steps of our algorithm are as follows:

1) Solving recurrences:
    a) Extract from the loop body the recurrence equations of $x_1, \dots, x_m$;
    b) By recurrence solving (Gosper, geometric series or C-finite), we obtain the system of closed forms of $x_1, \dots, x_m$ of the presented in eq. (3).
    W.l.o.g. we can assume that, in the closed forms of $x_1, \dots, x_n$: $s_1 = \dots = s_m$ and $\theta_{1,1} = \dots = \theta_{m,1}, \dots \dots, \theta_{1,s_1} = \dots = \theta_{m,s_m}$. (Otherwise, we can add the missing θ_{j,s_j} to the respective closed form as $0 \cdot \theta_{j,s_j}$, and by renaming we have in each closed form the same exponential terms.)
    Thus the closed forms of the loop variables are:

$$\begin{cases} x_1(n) = & p_{1,1}(n)\theta_1^n + \cdots + p_{1,s}(n)\theta_s^n \\ x_2(n) = & p_{2,1}(n)\theta_1^n + \cdots + p_{2,s}(n)\theta_s^n \\ \vdots & \\ x_m(n) = & p_{m,1}(n)\theta_1^n + \cdots + p_{m,s}(n)\theta_s^n \end{cases}, \quad (4)$$

    c) Since there are algebraic dependencies among θ_j $(j = \overline{1,s})$, according to def.(2.6) we have the polynomial equations: $t_k(\theta_1, \dots, \theta_s) = 0$ $(k > 1)$. We introduce the notations: $y_0 = n, y_1 = \theta_1^n, \dots, y_s = \theta_s^n$. Thus by rewriting (and reordering) eq.(4), we have:

$$\begin{cases} x_1(n) = & q_1(y_0, y_1, \dots, y_s) \\ x_2(n) = & q_2(y_0, y_1, \dots, y_s) \\ \vdots & \\ x_m(n) = & q_m(y_0, y_1, \dots, y_s) \end{cases}, \quad (5)$$

where $q_i \in \bar{\mathbb{K}}[y_0, y_1, \dots, y_s]$.

2) Polynomial invariant generation.
Consider $\mathbf{A}$ to be the generator of the annihilating ideal [7], [19] of $y_0, \dots, y_s$, i.e. $\mathbf{A} \trianglelefteq \bar{\mathbb{K}}[y_0, \dots, y_s]$. (In other words, $\mathbf{A}$ is the ideal of $t_k(\theta_1, \dots, \theta_s)$ $(k > 1)$.)
Consider
$I = \langle x_1 - q_1(y_0, \dots, y_s), \dots, x_m - q_m(y_0, \dots, y_s)\rangle + \mathbf{A}$.
Thus $I \trianglelefteq \bar{\mathbb{K}}[x_1, \dots, x_m, y_0, \dots, y_s]$.
Consider $J = I \cap \bar{\mathbb{K}}[x_1, \dots, x_m]$. Thus $J \trianglelefteq \bar{\mathbb{K}}[x_1, \dots, x_m]$, hence J is the annihilating ideal of $x_1, \dots, x_m$, i.e. it is the (generator of the) ideal of all polynomial identities (i.e. algebraic dependencies) among $x_1, \dots, x_m$.

Thus, if a P–solvable loop admits polynomial assertions, at the end of our algorithm we obtain all polynomial identities among the loop variables.

The Implementation of the Algorithm for Invariant Generation: We consider a slight generalization of the algorithm presented above, namely an invariant generation algorithm of P–solvable loops with conditionals.

For better understanding, we illustrate our algorithm on the below example:

Example 2.1: Program for Computing Square Roots, by K. Zuse

```
      Specification["SqrtZuse", SqrtZuse[↓ $a$, ↓ $err$, ↑ $q$],
      Pre → $(a \geq 1) \wedge (err > 0)$,
      Post → $(q^2 \leq a) \wedge (a < q^2 + err)$)]
      Program["SqrtZuse", SqrtZuse[↓ $a$, ↓ $err$, ↑ $q$],
      Module[{$r, p$}, $r := a - 1$;  $q := 1$;  $p := 1/2$;
(A)   While[$(2 * p * r \geq err)$],
      If[$2 * r - 2 * q * p \geq 0$
        Then $r := 2 * r - 2 * q - p$;  $q := q + p$;  $p := p/2$,
        Else $r := 2 * r$;  $p := p/2$]]]]
```

The invariant generation (based on the algorithm from the previous section) is performed as follows:

Step 0: Transformation of loops with conditionals, i.e. outer loops, into nested loops with assignments only, i.e. inner loops (see Remark. 2.4).

Remark 2.4: Transformation Rule

```
        {$I$}
        While[$b$,
        While[$b \wedge b1', c1; c2; c4$];
        While[$b \wedge \neg b1', c1; c3; c4$]]
        {$I \wedge \neg b$}
------------------------------------------------------------
{$I$}  While[$b, c1$; If[$b1$ Then $c2$ Else $c3$]; $c4$]  {$I \wedge \neg b$}
```
,

where:

- all loops have the same invariant I;
- $b1'$ represents condition $b1$ the modified by the assignment statement $c1$.

Performing this transformation on example (2.1), we obtain the following system of nested while loops (each inner while loop has only assignments):

```
(A)    While[$(2 * p * r \geq err)$,
(B)      While[$(2 * p * r \geq err) \wedge (2 * r - 2 * q * p \geq 0)$,
              $r := 2 * r - 2 * q - p$;  $q := q + p$;  $p := p/2$];
(C)      While[$(2 * p * r \geq err) \wedge \neg(2 * r - 2 * q * p \geq 0)$
              $r := 2 * r$;  $p := p/2$]]
```

Step 1: Solving recurrences for the inner loops (containing assignments only):

Step 1.(a): Extracting system of recurrences for each inner loop.

After statement simplification, we obtain the following recurrence systems for the while loops (B), (C):

While loop (B):
$i = \overline{0, \mathbf{I}}$

$$\begin{cases} p_{i+1} = & p_i/2 \\ q_{i+1} = & q_i + p_i \\ r_{i+1} = & 2 * r_i - 2 * q_i - p_i \end{cases}$$

While loop (C):
$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$

$$\begin{cases} p_{j'+1} = & p_{j'}/2 \\ q_{j'+1} = & q_{j'} \\ r_{j'+1} = & 2 * r_{j'} \end{cases}$$

where $\mathbf{I}, \mathbf{J}$ represent the unknown bounds of the iteration number of each loop counter i, j, respectively.

Step 1.(b): Solving system of recurrences for each inner loop (by Gosper algorithm, handling geometric series or C-finite solving using eq.(1).)

While loop (B):
$i = \overline{0, \mathbf{I}}$

$$\begin{cases} p_i & \underset{geom.series}{=} & \frac{1}{2^i} * p_0 \\ q_i & \underset{Gosper}{=} & q_0 + 2 * p_0 - \frac{1}{2^{i-1}} * p_0 \\ r_i & \underset{C-finite}{=} & 2^i * (r_0 - 2 * q_0 - 2 * p_0) - \\ & & \frac{1}{2^{i-1}} * p_0 + 2 * q_0 + 4 * p_0 \end{cases}$$

While loop (C):
$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$

$$\begin{cases} p_{j'} & \underset{geom.series}{=} & \frac{1}{2^j} * p_{\mathbf{I}} \\ q_{j'} & = & q_{\mathbf{I}} \\ r_{j'} & \underset{geom.series}{=} & 2^j * r_{\mathbf{I}} \end{cases}$$

For solving C-finite recurrences we use the `SumCracker` package [19] (see Remark 2.1).

Step 1.(c): Introduction of extra variables with their algebraic dependencies.

While loop (B): We denote $x_i = 2^{-i}, y_i = 2^{-i}$. Thus, by Remark 2.3, and using the `Dependencies` package (see Remark 2.2), we can determine all algebraic dependencies among x_i, y_i, if they are related. In this case, we obtain the algebraic dependency: $x_i * y_i - 1 = 0$.

While loop (C): Working in the same manner, we introduce the extra variables $u_{j'} = 2^j, v_{j'} = 2^{-j}$, and their algebraic dependency $u_{j'} * v_{j'} - 1 = 0$.
Thus, the final system of closed forms, describing the behavior

of loop-variables in the while loops (B) and (C) is:

While loop (B):
$i = \overline{0, \mathbf{I}}$

$$\begin{cases} p_i & = & p_0 * y_i \\ q_i & = & q_0 + 2 * p_0 - 2 * p_0 * y_i \\ r_i & = & x_i * (r_0 - 2 * q_0 - 2 * p_0) - \\ & & 2 * p_0 * y_i + 2 * q_0 + 4 * p_0 \\ x_i * y_i - 1 & = & 0 \end{cases} \quad (6)$$

While loop (C):
$j = \overline{0, \mathbf{J}}, j' = j + \mathbf{I}$

$$\begin{cases} p_{j'} & = & p_{\mathbf{I}} * v_{j'} \\ q_{j'} & = & q_{\mathbf{I}} \\ r_{j'} & = & r_{\mathbf{I}} * u_{j'} \\ u_{j'} * v_{j'} - 1 & = & 0 \end{cases}$$

Since the systems from eq.(6), i.e. of while loops (B) and (C), satisfy the definition of a P-solvable loop, we are able to determine polynomial invariants for the while loop (A), if they exist.

Step 1.(d): Merging connected systems of recurrences (inner loops contained in outer loop).

While loop (A): By substituting the values of variables of the while loop (B) into the expressions of variables of the while loop (C) (from eq.(6)), we obtain the system (using initial values and $x_{j'} = x_{\mathbf{I}}$, $y_{j'} = y_{\mathbf{I}}$, as well as ignoring the index J by the convention that the variables indexed by J denote the final values after an iteration of the while loop (A)):

$$\begin{cases} p & = & \frac{1}{2} * y * v \\ q & = & 2 - y \\ r & = & ((a - 4) * x - y + 4) * u \\ x * y - 1 & = & 0 \\ u * v - 1 & = & 0 \end{cases} \quad (7)$$

Step 2: Polynomial Invariant Generation.
By eliminating the loop-bound variables I, J and the extra variables u, v, x, y from eq.(7), and performing Gröbner basis computation in order to get the essential set of polynomial invariants, the automatically obtained polynomial invariant for the while loop (A) is:

$$a - 2 * p * r = q^2.$$

Step 3: The final invariant property for the outer loops.
In addition to the automatically generated invariant properties, there are some other invariant properties that cannot be yet obtained automatically by our algorithm (e.g. inequalities, modulo expressions, etc.). In the current version of our verification framework for imperative programs these properties are still considered to be given by the user.

Thus, in the case of example 2.1, the user–asserted invariant properties are:

$$(err \geq 0) \wedge (p \geq 0) \wedge (r \geq 0).$$

Hence, the final invariant property used for the verification of example 2.1 is:

$$(a - 2 * p * r = q^2) \wedge (err \geq 0) \wedge (p \geq 0) \wedge (r \geq 0).$$

Using this invariant the VCG produces a universally quantified lemma with 3 proof obligations in order to prove partial correctness of the program. The proofs of these verification conditions were done by applying the PCS prover of the *Theorema* system [3].

D. Discussion

We have tested our algorithm on several examples of programs that needed polynomial invariants in order to be verified [22]. In all cases, our algorithm succeeded with the generation of polynomial invariants.

Note that the generated invariant equalities do not depend on the pre- and/or postconditions of the program. However, currently we investigate the possibility for invariant inequality generation by using quantifier elimination together with manipulation of pre- and postconditions. Moreover, current research suggests that such invariants are very useful for performing loop optimization [13] and constant propagation [25].

III. Functional Programs

We consider the correctness problem expressed as follows: *given* the program which computes the function F in a domain D 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 VC_1, ... VC_n which are sufficient for the program to satisfy the specification. The function F satisfies the specification, if: F terminates on any input x satisfying I_F, and, for each such input, the condition $O_F[x, F[x]]$ holds. This is also called "total correctness" of the program.

$$(\forall x : I_F[x])\ O_F[x, F[x]], \quad (8)$$

Any VCG should come together with its *Soundness* statement, that is: for a given program F, defined on a domain D, with a specification I_F and O_F if the verification conditions VC_1, ... VC_n hold in the theory of D, then the program F satisfies its specification I_F and O_F.

Moreover, we are also interested in the following question: What if some of the verification conditions do not hold? May we conclude that the program is not correct? In fact, the program may still be correct. However, if the VCG is complete, then one can be sure that the program is not correct. A VCG is complete, if whenever the program satisfies its specification, the produced verification conditions hold.

The notion of *Completeness* of a VCG is important for the following two reasons: theoretically, it is the dual of *Soundness* and practically, it helps debugging. Any counterexample for the failing verification condition would carry over to a counterexample for the program and the specification, and thus give a hint on "what is wrong". Indeed, most books about program verification present methods for verifying correct programs.

However, in practical situations, it is the failure which occurs more often until the program and the specification are completely debugged.

A. Coherent Programs

In this subsection we state the principles we use for writing coherent programs with the aim of building up a non-contradictory system of verified programs. Although, these principals are not our invention (similar ideas appear in [20]), we state them here because we want to emphasize on and later formalize them.

Building up correct programs: Firstly, we want to ensure that our system of coherent programs would contain only correct (verified) programs. This we achieve, by:

– start from basic (trustful) functions e.g. addition, multiplication, etc.;

– define each new function in terms of already known (defined previously) functions by giving its source text, the specification (input and output predicates) and prove their total correctness with respect to the specification.

This simple inductively defined principle would guarantee that no wrong program may enter our system. The next we want to ensure is the easy exchange (mobility) of our program implementations. This principle is usually referred as:

Modularity: Once we define the new function and prove its correctness, we "forbid" using any knowledge concerning the concrete function definition. The only knowledge we may use is the specification. This gives the possibility of easy replacement of existing functions. For example we have a powering function P, with the following program definition (implementation):

$$P[x,n] = \textbf{ If } n=0 \textbf{ then } 1 \textbf{ else } P[x,n-1]*x$$

The specification of P is:
The domain $\mathbb{D}=\mathbb{R}^2$, precondition $I_P[x,n] \iff n\in\mathbb{N}$ and a postcondition $O_P[x,n,P[x,n]] \iff P[x,n]=x^n$.

Additionally, we have proven the correctness of P. Later, after using the powering function P for defining other functions, we decide to replace its definition (implementation) by another one, however, keeping the same specification. In this situation, the only thing we should do (besides preserving the name) is to prove that the new definition (implementation) of P meets the old specification.

Furthermore, we need to ensure that when defying a new program, all the calls made to the existing (already defined) programs obey the input restrictions of that programs – we call this:

Appropriate values for the auxiliary functions. The following example will give an intuition on what we are doing. Let the program for computing F be:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } H[x] \textbf{ else } G[x],$$

with the specification of F (I_F and O_F) and specifications of the auxiliary functions H (I_H and O_H), G (I_G and O_G). The two verification conditions, ensuring that the calls to the auxiliary functions have appropriate values are:

$$(\forall x : I_F[x])\ (Q[x] \implies I_H[x])$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_G[x]).$$

B. Simple Recursive Programs

As we already mentioned, there is no universal VCG. Thus, in our research, we concentrate on constructing a VCG which is appropriate only for a certain kind of recursive programs – Simple Recursive Programs. They are the most used in practice and at the same time the most elementary (from mathematical point of view) ones. Simple recursive programs are those F, which may be defined as:

$$F[x] = \textbf{ If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x,F[R[x]]], \qquad (9)$$

where Q is a predicate[1] and S,C,R are auxiliary functions ($S[x]$ is a "simple" function, $C[x,y]$ is a "combinator" function, and $R[x]$ is a "reduction" function). We assume that the functions S, C, and R satisfy their specifications given by $I_S[x]$, $O_S[x,y]$, $I_C[x,y]$, $O_C[x,y,z]$, $I_R[x]$, $O_R[x,y]$. Note that functions with multiple arguments also fall into this scheme, because the arguments x,y,z could be vectors (tuples).

Type (or domain) information does not appear explicitly in this formulation, however it may be included in the input conditions.

Note that the "programming language" used here contains only the construct **If–then–else** in addition to the language of first order predicate logic.

One may also use some additional restrictions on the shape of the definitions of Q, S, C, and R (e. g. that they do not contain quantifiers) in order to make the program "easy" to execute. However, this depends on the complexity of the "interpreter" ("compiler") and does not influence the actual generation of the verification conditions. In general, the auxiliary functions may be already defined in the underlying theory, or by other programs (that includes logical terms).

Considering Coherent Simple Recursive programs, we give here the appropriate definition:

Definition 3.1: Let S, C, and R be functions which satisfy their specifications. Then the program (9) is coherent if the following conditions hold:

$$(\forall x : I_F[x])\ (Q[x] \implies I_S[x]) \qquad (10)$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_F[R[x]]) \qquad (11)$$
$$(\forall x : I_F[x])\ (\neg Q[x] \implies I_R[x]) \qquad (12)$$
$$(\forall x : I_F[x])$$
$$(\neg Q[x] \wedge O_F[R[x],F[R[x]]] \implies I_C[x,F[R[x]]]). \qquad (13)$$

Theorem 3.1: Let S, C, and R be functions which satisfy their specifications. Let also the program (9) be coherent.

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

Then, (9) satisfies the specification given by I_F and O_F if and only if the following verification conditions hold:

$$(\forall x : I_F[x])\ (Q[x] \implies O_F[x, S[x]]) \tag{14}$$

$$(\forall x : I_F[x]) \\ (\neg Q[x] \wedge O_F[R[x], F[R[x]]] \implies O_F[x, C[x, F[R[x]]]]) \tag{15}$$

$$(\forall x : I_F[x])\ (F'[x] = 0) \tag{16}$$

where:

$$F'[x] = \ \textbf{If } Q[x] \textbf{ then } 0 \textbf{ else } F'[R[x]] \tag{17}$$

Before going to the detailed proof, we note that this theorem, in fact, gives two statements, namely:

– *Soundness*: If (14), (15) and (16) hold, then the program (9) is correct, and

– *Completeness*: If (9) is correct, then (14), (15) and (16) hold.

The proof of the *Soundness* statement is split into two parts:

– prove partial correctness of (9) using Scott induction in the fixpoint theory of programs [24];

– prove termination using induction on the number of recursive calls.

Formally, the *Soundness* statement of the theorem is an immediate consequence of a similar theorem stated in [16], where the necessary conditions for the program correctness are (14), (15), (16), (10), (11), (12) and (13). *Proof:* (*Completeness*) Assume (9) is correct. We start now with proving (14) and (15) simultaneously. Take arbitrary but fixed x and assume $I_F[x]$. We consider firstly the case $Q[x]$, then by the definition of F, we have $F[x] = S[x]$, and by using the correctness formula (8) of F, we conclude (14) holds. The validity of (15) is trivial, because we have $Q[x]$.

In the second case, we assume $\neg Q[x]$. Now, the validity of (14) is trivial. For proving (15), it suffices to show $O_F[x, C[x, F[R[x]]]]$. By the definition of F, we obtain $F[x] = C[x, F[R[x]]]$. Since, F is assumed to be correct, that is, its correctness formula hold, by having $I_F[x]$, we obtain $O_F[x, F[x]]$, and hence $O_F[x, C[x, F[R[x]]]]$, which completes the proof of (15).

Now, we show that the simplified version F' of the initial function F terminates. Moreover, F' terminates if F terminates. In the course of the proof, one may notice that proving $F'[x] = 0$ is the same as proving that F' terminates. The precise proof goes as follows: Take arbitrary but fixed x and assume $I_F[x]$. Since $F[x]$ terminates, we denote $F(x) = a$, for some constant a. We first show that there must exist a number n such that after n steps of recursive calls, the predicate Q will be satisfied, that is

$$F(x) = a \implies (\exists n \in \mathbb{N})(Q[R^n[x]]), \tag{18}$$

where $R^0[x] = x$ and $R^{n+1}[x] = R[R^n[x]]$. We prove this statement by contradiction, i.e. assume:

$$F(x) = a \wedge (\forall n \in \mathbb{N})(\neg Q[R^n[x]]).$$

Henceforth, by $\downarrow$ we denote the predicate expressing termination (we want to prove $F'[x] \downarrow$) and by Ω the nowhere defined function and by $\perp$ the nonterminating term – $\forall x \Omega[x] = \perp$.

Let $f_0, f_1, ...f_m...$ be the finite approximations of F obtained as

$$f_0[x] = \Omega[x]$$
$$f_{m+1}[x] = \ \textbf{If } Q[x] \textbf{ then } S[x] \textbf{ else } C[x, f_m[R[x]]],$$

then the function computed by (9) is defined as

$$F' = \bigcup_m f_m$$

which is the least fixpoint of (9).

Since we have $F(x) = a$, there must exist a finite approximation f_m, such that $f_m[x] = a$. If $m = 0$ then $f_0[x] = a$ which contradicts the definition of $f_0 = \Omega$, hence $m > 0$.

From $(\forall n \in \mathbb{N})(\neg Q[R^n[x]])$ and in particular $\neg Q[x]$ by the definition of f_m we obtain $f_m[x] = C[x, f_{m-1}[R[x]]]$, thus $f_{m-1}[R[x]] \downarrow$.

By repeating the same kind of reasoning m times, we obtain that $f_{m-m}[R^m[x]] = f_0[R^m[x]]$ and by its definition $f_0[R^m[x]] = \perp$ which contradicts $f_m[x] = a$, and so, we have proven (18).

The proof of the termination of F' ($F'[x] = 0$) will be completed by proving the following statement:

$$(\exists n \in \mathbb{N})(Q[R^n[x]] \ \Rightarrow F'[x] \downarrow). \tag{19}$$

Assume $Q[R^n[x]$ for some n and, without loss of generality, assume $(\forall k < n)(\neg Q[R^k[x]])$.

– Case 1: $n = 0$. By the definition of F', $F'[x] = 0$ and thus $F'[x] \downarrow$.

– Case 2: $n > 0$. By unfolding the definition of F', we obtain $F'[x] = F'[R[x]] = F'[R^2[x]]...$, and finally, we obtain $F'[x] = F'[R^n[x]]$. From here, by the definition of F' and $Q[R^n[x]$ we obtain $F'[x] = 0$. This completes the proof. ■

In fact, the method presented here works analogously on the more general class of programs containing **Case** (**If–then–else** with several cases), as illustrated in the example below.

C. Example and Discussion

We consider *binary powering*:

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

We consider this program in the context of the theory of real numbers, and in the following formulae, all variables are implicitly assumed to be real. Additional type information (e. g. $n \in \mathbb{N}$) may be explicitly included in some formulae.

The specification is:

$$(\forall x, n : n \in \mathbb{N})\ P[x, n] = x^n. \tag{20}$$

The (automatically generated) conditions for **coherence** are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0 \ \Rightarrow \mathbb{T}) \tag{21}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \mathrm{Even}[n] \ \Rightarrow \mathrm{Even}[n]) \tag{22}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \wedge \neg\mathrm{Even}[n] \ \Rightarrow \mathrm{Odd}[n]) \tag{23}$$

$$(\forall x, n, m : n \in \mathbb{N})(n \neq 0 \wedge \mathrm{Even}[n] \wedge m = (x * x)^{n/2} \Rightarrow \mathbb{T}) \tag{24}$$

$$(\forall x, n, m : n \in \mathbb{N}) \\ (n \neq 0 \wedge \neg\mathrm{Even}[n] \wedge m = (x * x)^{(n-1)/2} \ \Rightarrow \mathbb{T}) \tag{25}$$

One sees that the formulae (21), (24) and (25) are trivially valid. The formulae (22) and (23) are easy consequences of the elementary theory of reals and naturals.

For the further check of **correctness** the generated conditions are:

$$(\forall x, n : n \in \mathbb{N})\ (n = 0\ \Rightarrow 0 = x^n) \tag{26}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \mathrm{Even}[n]\ \Rightarrow n/2 \in \mathbb{N}) \tag{27}$$

$$\begin{array}{l}(\forall x, n, m : n \in \mathbb{N})\\ (n \neq 0 \land \mathrm{Even}[n] \land m = (x * x)^{n/2}\ \Rightarrow m = x^n)\end{array} \tag{28}$$

$$(\forall x, n : n \in \mathbb{N})\ (n \neq 0 \land \neg\mathrm{Even}[n]\ \Rightarrow (n-1)/2 \in \mathbb{N}) \tag{29}$$

$$\begin{array}{l}(\forall x, n, m : n \in \mathbb{N})\\ (n \neq 0 \land \neg\mathrm{Even}[n] \land m = (x * x)^{(n-1)/2}\ \Rightarrow x * m = x^n)\end{array} \tag{30}$$

$$(\forall x, n : n \in \mathbb{N})\ P'[x, n] = 0, \tag{31}$$

where

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

The proofs of these verification conditions are straightforward, however (26) reduces to:

$$0 = 1$$

(because we consider a theory where $0^0 = 1$).

Therefore, according to the *completeness* of the method, we conclude that the program P does not satisfy its specification. Moreover, the failed proof gives a hint for "debuging": we need to change the return value in the case $n = 0$ to 1.

The new program will have the same verification conditions, except that (26) is changed to

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

which is now provable.

IV. Conclusions

The theoretical basis and the concrete implementation of our automatic system for program verification is a result of an experimental and practical approach to the problem of program correctness, both procedural and functional ones. Although the examples presented here appear to be relatively simple, they already demonstrate the usefulness of our approach in the general case. We aim at extending these experiments to industrial-scale examples, which are in fact not more complex from the mathematical point of view. Furthermore we aim at improving the education of future software engineers by exposing them to successful examples of using formal methods (and in particular automated reasoning) for the verification and the debugging of concrete programs.

References

[1] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful Techniques for the Automatic Generation of Invariants. In *Proc. of SAS 1996*, volume 1102 of *LNCS*, pages 323–335, 1996.

[2] B. Buchberger. *Introduction to Gröbner Bases*, volume 251 of *London Mathematical Society Lecture Notes: "Gröbner Bases and Applications"*, pages 3–31. Cambridge University Press, 1998.

[3] B. Buchberger, A. Craciun, T. Jebelean, L. Kovacs, T. Kutsia, K. Nakagawa, F. Piroi, N. Popov, J. Robu, M. Rosenkranz, and W. Windsteiger. Theorema: Towards Computer-Aided Mathematical Theory Exploration. *Journal of Applied Logic*, 4(4):470–504, 2006.

[4] B. Buchberger and F. Lichtenberger. *Mathematics for Computer Science I - The Method of Mathematics (in German)*. Springer, 2nd edition, 1981.

[5] M. A. Colon, S. Sankaranarayanan, and H. B. Sipma. Linear Invariant Generation Using Non-Linear Constraint Solving. In *Proc. of CAV 2003*, volume 2725 of *LNCS*, pages 420–432, 2003.

[6] P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Proc. of VMCAI'05*, pages 1–24, 2005.

[7] D. Cox, J. Little, and D. O'Shea. *Ideal, Varieties, and Algorithms. An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Springer, 2nd edition, 1998.

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[9] B. Elspas, M. W. Green, K. N. Lewitt, and R. J. Waldinger. Research in interactive program—proving techniques. Technical report, Stanford Research Institute, May 1972.

[10] G. Everest, A. van der Poorten, I. Shparlinski, and T. Ward. *Recurrence Sequences*, volume 104 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2003.

[11] R. W. Floyd. Assigning Meanings to Programs. In *Proc. Symphosia in Applied Mathematics 19*, pages 19–37, 1967.

[12] S. M. German and B. Wegbreit. A Synthesizer of Inductive Assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.

[13] B. Goldberg, L. Zuck, and C. Barrett. Into the Loops. In *Proc. of COCV 2004*, 2004.

[14] R. W. Gosper. Decision Procedures for Indefinite Hypergeometric Summation. *Journal of Symbolic Computation*, 75:40–42, 1978.

[15] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM*, 12, 1969.

[16] T. Jebelean, L. Kovács, and N. Popov. Experimental Program Verification in the Theorema System. *Int. Journal on Software Tools for Technology Transfer (STTT)*, 2006. To appear.

[17] D. Kapur. Automatically Genearting Loop Invariants using Quantifier Elimination. In *Proc. of ACA*, 2004.

[18] M. Karr. Affine Relationships Amomg Variables of Programs. *Acta Informatica*, 6:133–151, 1976.

[19] M. Kauers. *Algorithms for Nonlinear Higher Order Difference Equations*. PhD thesis, RISC-Linz, Johannes Kepler University Linz, Austria, 2005.

[20] M. Kaufmann and J. S. Moore. An Industrial Strength Theorem Prover for a Logic Based on Common Lisp. *Software Engineering*, 23(4):203–213, 1997.

[21] M. Kirchner. Program Verification with the Mathematical Software System Theorema. Technical Report 99-16, RISC-Linz, Austria, 1999.

[22] L. Kovacs. Finding Polynomial Invariants for Imperative Loops in the Theorema System. Technical Report 06-03, RISC-Linz, Austria, 2006.

[23] L. Kovacs and T. Jebelean. An Algorithm for Automated Generation of Invariants for Loops with Conditionals. *IEEE Computer Society*, pages 245–249, 2005. *Proc. of SYNASC'05*.

[24] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Inc., 1974.

[25] M. Müller-Olm, M. Petter, and H. Seidl. Interprocedurally Analyzing Polynomial Identities. In *Proc. of STACS 2006*, 2006.

[26] M. Müller-Olm and H. Seidl. Polynomial Constants are Decidable. In *Proc. of SAS 2002*, volume 2477 of *LNCS*, 2002. pp. 4-19.

[27] M. C. Paull. *Algorithm Design. A Recursion Transformation Framework*. Wiley, 1987.

[28] E. Rodriguez-Carbonell and D. Kapur. Automatic Generation of Polynomial Loop Invariants: Algebraic Foundations. In *Proc. of ISSAC 04*, 2004.

[29] 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.

[30] S. Sankaranaryanan, B. S. Henry, and Z. Manna. Non-Linear Loop Invariant Generation using Gröbner Bases. In *Proc. of POPL 2004*, Venice, Italy, 2004.

[31] R. P. Stanley. Differentiably Finite Power Series. *European Journal of Combinatorics*, 1:175–188, 1980.

[32] A. Tiwari, H. Ruess, H. Saidi, and N. Shankar. A Technique for Invariant Generation. In *Proc. of TACAS*, volume 2031 of *LNCS*, pages 113–127, 2001.