Practical Program Verification by Forward Symbolic Execution: Correctness and Examples

EXTENDED ABSTRACT

Mădălina Eraşcu and Tudor Jebelean

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{Madalina.Erascu,Tudor.Jebelean}@risc.uni-linz.ac.at

Abstract. We present the theoretical aspects and a prototype implementation in the *Theorema* system of a method for the verification of recursive imperative programs. The method is based on forward symbolic execution and functional semantics and generates first order verification conditions for the total correctness which use only the underlying theory of the program. All verification conditions are generated automatically by our prototype implementation in the frame of the *Theorema* system based on *Mathematica*.
The termination property is expressed as an induction principle depending on the structure of the program with respect to recursion. It turns out that part of the verification conditions (notably the termination condition) are crucial for the existence of the function defined by the program, without which the total correctness formula is trivial due to inconsistency of the assumptions.
The formal description of the method is the basis for the implementation and also for the proof of its correctness.

1 Introduction

We present a formal verification method for imperative programs based on symbolic execution [4, 1], forward reasoning [5, 3] and functional semantics [6]. The distinctive features of our approach are:

- All verification conditions are formulated in the theory of the objects which are manipulated by the program (the object theory – see below).
- The notions of program, semantics, verification condition, and termination are precisely formalized in predicate logic.
- Termination is treated in a purely logical way, namely as existence and uniqueness of the function implemented by the program.

Our approach is purely logical. We assume that the properties of the constants, functions and predicates which are used in the program are specified in an *object theory* Υ. (By a *theory* we understand a set of formulae in the language of predicate logic with equality.) For the purpose of reasoning about imperative programs we construct a certain *meta–theory* containing the properties of

the meta–predicate Π (which checks a program for syntactical correctness) and the meta–functions Σ (which defines the semantics of a program), Γ (which generates the verification conditions) and Θ (which generates one termination condition). The programming constructs (statements), the program itself, as well as the terms and the formulae from the object theory are *meta–terms* from the point of view of the meta-theory, and they behave like *quoted* (because the meta–theory does not contain any equalities between programming constructs, and also does not include the object theory).

The programming constructs are: *abrupt statement* (`Return`), *assignments* (allowing *recursive calls*) and *conditionals* (`If` with one and two branches). *Recursive calls* are indicated by the presence of the function symbol f, conventionally corresponding to the function realized by the program. A program is a list of statements, it takes as formal arguments a certain number of variables (denoted conventionally by $\overline{x}$) and it returns a single value (denoted conventionally by y). The meta–predicate Π checks whether a program is syntactically correct, and also that each branch terminates with `Return` and that each variable is initialized before its usage. (We call these variables *active*.)

The semantics of a program is defined via Σ as being an implicit definition *at object level* (that is, using only the signature of the object theory) of the function (conventionally denoted as f) which is implemented by the program. Practically, Σ works by forward symbolic execution on all branches of the program, using as *state* the current substitution for the active variables. Σ produces a conjunction of clauses – conditional definitions for $f[\overline{x}]$. Each clause depends on the accumulated [negated] conditions of the `If` statements leading to a certain `Return` statement, whose argument (symbolically evaluated) represents the corresponding value of $f[\overline{x}]$. Note that Σ effectively translates the original program into a *functional* program. From this point on, one could reason about the program using the Scott fixpoint theory ([5], pag. 86), however we prefer a purely logical approach.

The meta–function Γ produces the verification conditions as a conjunction of formulae at object level. The verification of the program is performed with respect to a given *specification*, composed of two predicates: the input condition $I_f[\overline{x}]$ and the output condition $O_f[\overline{x}, y]$, whose definitions are assumed to be present in the object–theory.

Moreover, some of the functions present in the object theory also have a specification – we call these *additional* functions. These functions will not be present in the verification conditions, but only their specifications. Typical examples of additional functions are those implemented by other programs.

The other functions from the object theory (we call them *basic*) have only input conditions, but no output conditions. These functions will occur in the verification conditions, thus the proof of such conditions will use the properties of the basic functions from the object theory. Typical examples of basic functions are the arithmetic operations in various number domains.

Γ operates similarly to Σ by using symbolic execution on all branches of the program, but in addition generates formulae using the following principles:

– coherence (safety): the arguments of every function (including the currently defined f) call must be satisfy the respective input condition;
– functional correctness: the return value on each branch must satisfy the output condition;

All the verification conditions are first order formulae at object level. This includes the termination condition, because the universally quantified predicate present in the induction principle is represented by a new constant.

The meta-function Θ generates one termination condition: a certain induction principle corresponding to the structure of the recursive calls, which in fact ensures the existence and uniqueness of the function f defined implicitly by Σ.

1.1 Related Work

Our approach follows the principles of the symbolic execution approach introduced in [4], but gives formal definitions in a meta-theory for the meta-level functions and predicate which characterize the object computation.

These definitions give the possibility of reflective view of the system by describing how the data (the state, the program, the verification conditions) are manipulated and by introducing a causal connection between the data and the status of computation (a certain statement of the program determines a certain computation of $\Sigma[P]$ and Γ to be performed).

We mention that our approach keeps the verification process very simple: the verification conditions generated are first order logic formulae and the proofs of correctness is kept at object-level without introducing a model of computation.

Approaches for solving the correctness of symbolic executed programs exists due to [5, 8, 2]; for the imperative programs containing *assignments*, *conditionals* and `while` *loops bounded* on the number of times they are executed, the proof of correctness is given by analyzing the verification conditions on each branch of the program. For the programs containing loops with unbounded number of iterations, the branches of the program are infinite and have to be traversed inductively in order to prove/disprove their correctness. In the inductive traversal of the tree, additional assertions have to be provided, called *inductive assertions*. But the inductive assertions method applies to partial correctness proofs ([5]), while our approach concentrates in proving the total correctness of programs.

2 The Formal Meta–Theory

2.1 Program Syntax

The meta–level predicate Π checks whether a program is syntactically correct and additionally that every variable (except the input variables) is used only after being assigned, and that each branch contains `Return`.

As *states* of the execution we use substitutions σ (set of replacements of the form $\{var \to expr\}$). We sometimes write $\{\overline{var} \to \overline{expr}\}$ instead of

$\{var_1 \to expr_1, var_2 \to expr_2, ...\}$. We denote by $\bar{x}$ the sequence of input variables. The meta–function *Vars* returns the variables which occur in a term.

The formulae composing the meta–definitions below are to be understood as universally quantified over the meta–variables of various types as described in the sequel: We denote by $t \in \mathcal{T}$ a term from the set of object level terms, by $v \in \mathcal{V}$ a variable from the set of variables, by $V \subset \mathcal{V}$ the set of active variables and by φ an object level formula. P_T, P_F, P are tuples of statements representing parts of programs: P_T is the tuple of statements executed if φ is evaluated to $True$, P_F in the case of $False$ evaluation of φ, while P represents the rest of the program to be executed. The tuple P_F can be also empty, case which corresponds to the *one branch* `If` *statement*.

Definition 1

1. $\Pi[P] \Longleftrightarrow \Pi[\{\bar{x}\}, P]$
2. $\Pi[V, \langle \texttt{Return}[t] \rangle \smile P] \Longleftrightarrow Vars[t] \subseteq V$
3. $\Pi[V, \langle v := t \rangle \smile P] \Longleftrightarrow \bigwedge \begin{cases} Vars[t] \subseteq V \\ \Pi[V \cup \{v\}, P] \end{cases}$
4. $\Pi[V, \langle \texttt{If}[\varphi, P_T, P_F] \rangle \smile P] \Longleftrightarrow \bigwedge \begin{cases} Vars[\varphi] \subseteq V \\ \Pi[V, P_T \smile P] \\ \Pi[V, P_F \smile P] \end{cases}$
5. $\Pi[V, P] \Longleftrightarrow \mathbb{F}$, in all other cases

2.2 Program Semantics

The meta–level function Σ creates an object–level formula containing a new function constant f. We consider this formula as being the *semantics* of the program P in the following sense: the function implemented by the program satisfies $\Sigma[P]$.

The formula $\Sigma[P]$ has the shape:

$$\underset{\bar{x}:I_f}{\forall} \bigwedge \left\{ p_i[\bar{x}] \Rightarrow (f[\bar{x}] = g_i[\bar{x}]) \right\}_{i=1}^{n}$$

(We denoted by „$\bar{x} : I_f$" in the condition "$\bar{x}$ satisfies I_f".) Here f is a new (second order) symbol – a name for the function defined by the program. In the case of recursive programs, f may occur in some p_i's and g_i's.

Each of the n paths of the program has associated a object–level formula $p_i[x]$ – the accumulated `If`–conditions on that path, and the object–level term $g_i[x]$ – the symbolic expression of the return value obtained by composing all the assignments (symbolic execution). Note that $p_i[x]$ and $g_i[x]$ do not contain other free variables than x.

Definition 2

1. $\Sigma[P] = \underset{\bar{x}}{\forall}\big(\Sigma[\{\bar{x} \to \bar{x_0}\}, P]_{\{\bar{x_0} \to \bar{x}\}}\big)$

2. $\Sigma[\sigma, \langle \texttt{Return}[t]\rangle \smile P] = (f[\bar{x_0}] = t\sigma)$

3. $\Sigma[\sigma, \langle v := t\rangle \smile P] = \Sigma[\sigma \circ \{v \to t\sigma\}, P]$

4. $\Sigma[\sigma, \langle \texttt{If}[\varphi, P_T, P_F]\rangle \smile P] = \bigwedge \begin{cases} \varphi\sigma \Longrightarrow \Sigma[\sigma, P_T \smile P] \\ \neg\varphi\sigma \Longrightarrow \Sigma[\sigma, P_F \smile P] \end{cases}$

Remark 1. The way Σ handles the `If` statement insures: $\underset{\bar{x}}{\forall}\bigvee\limits_i p_i[\bar{x}] = I_f[\bar{x}]$ (all branches are covered) and $\underset{i\neq j}{\forall} \neg(\underset{\bar{x}}{\forall} p_i[\bar{x}] \wedge p_j[\bar{x}])$ (branches are mutually disjoint).

2.3 Partial Correctness

As we mentioned before, the meta–level function Γ generates two kinds of verification conditions: *coherence (safety)* conditions and *functional* conditions.

In this section we present the first two groups of conditions, which together insure *partial correctness*. The termination condition is subject to next section.

The *coherence* conditions have the shape $\Phi \Rightarrow I_h[t_1, t_2, \ldots]$, where I_h is the input condition of h, and $t_1, t_2, \ldots$ are the symbolic values of the function call. The formula Φ accumulates the conditions on the current branch, namely:

- the conditions from `If` statements;
- the input conditions and the output conditions for the previous function calls.

We consider γ, $\overline{\gamma}$ - a variable or a constant and respectively a sequence of variable and/or constants from the theory Υ, basic functions h, additional functions g, arbitrary function u. The symbol y is a new constant name. An expression like $e_{\tau \leftarrow w}$ denotes that τ is replaced by w in e, where w is a new variable name.

Definition 3

1. $\Gamma[P] = \underset{\bar{x}}{\forall}\big(\Gamma[\{\bar{x} \to \bar{x}_0\},\ I_f[\bar{x}_0],\ P]_{\{\bar{x}_0 \to \bar{x}\}}\big)$
2. $\Gamma[\sigma,\ \Phi,\ \langle \mathtt{Return}[\gamma]\rangle \backsim P] = \Big(\Phi \Rightarrow O_f[\bar{x}_0, \gamma\sigma]\Big)$
3. $\Gamma[\sigma,\ \Phi,\ \langle \mathtt{Return}[t_{\tau \leftarrow u[\overline{\gamma}]}]\rangle \backsim P] = \Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ \mathtt{Return}[t_{\tau \leftarrow w}]\rangle \backsim P]$
4. $\Gamma[\sigma,\ \Phi,\ \langle v := \gamma\rangle \backsim P] = \Gamma[\sigma \circ \{v \to \gamma\sigma\},\ \Phi,\ P]$
5. $\Gamma[\sigma,\ \Phi,\ \langle v := h[\overline{\gamma}]\rangle \backsim P] =$
$$\bigwedge \begin{cases} \Phi \Rightarrow I_h[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\},\ \Phi \wedge I_h[\overline{\gamma}\sigma],\ P] \end{cases}$$
6. $\Gamma[\sigma,\ \Phi,\ \langle v := g[\overline{\gamma}]\rangle \backsim P] =$
$$\bigwedge \begin{cases} \Phi \Rightarrow I_g[\overline{\gamma}\sigma] \\ \Gamma[\sigma \circ \{v \to c\},\ \Phi \wedge I_g[\overline{\gamma}\sigma] \wedge O_g[\overline{\gamma}\sigma, c],\ P] \end{cases}$$
7. $\Gamma[\sigma,\ \Phi,\ \langle v := t_{\tau \leftarrow u[\overline{\gamma}]}\rangle \backsim P] = \Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ v := t_{\tau \leftarrow w}\rangle \backsim P]$
8. $\Gamma[\sigma,\ \Phi,\ \langle \mathtt{If}[\varphi_{\tau \leftarrow u[\overline{\gamma}]},\ P_T,\ P_F]\rangle \backsim P] =$
$$\Gamma[\sigma,\ \Phi,\ \langle w := u[\overline{\gamma}],\ \mathtt{If}[\varphi_{\tau \leftarrow w},\ P_T,\ P_F]\rangle \backsim P]$$
9. $\Gamma[\sigma,\ \Phi,\ \langle \mathtt{If}[\varphi,\ P_T,\ P_F]\rangle \backsim P] = \bigwedge \begin{cases} \Gamma[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \backsim P] \\ \Gamma[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \backsim P] \end{cases}$

The order of the above clauses of Γ has a semantic meaning. Namely, we use this as an abbreviation for additional conditions which should be added to clauses of the definition in order to specify that, for instance, the equality (3.9) is applied only if no subterm of φ is of the form $u[\overline{\gamma}]$ – as specified in the clause (3.8).

2.4 Termination

We want to generate verification conditions which ensure that a program is correct with respect to a specification composed of two object–level formulae: the input condition $I_f[x]$ and the output condition $O_f[x, y]$. Apparently, the correctness could be expressed as: "The formula $\underset{x}{\forall} I_f[x] \Rightarrow O_f[x, P[x]]$ is a logical consequence of the theory Υ augmented with $\Sigma[P]$ and with the verification conditions." However, this always holds in the case that $\Sigma[P]$ is contradictory to Υ, which may happen when the program is recursive. Therefore, *it is crucial that the existence (and possibly the uniqueness) of an f satisfying $\Sigma[P]$ is a logical consequence of the object theory augmented with the verification conditions.* More concretely, before using $\Sigma[P]$ as an assumption, one should prove $\underset{f}{\exists}\Sigma[P]$. The later is ensured by the termination condition which is expressed as an induction scheme developed from the structure of the recursion.

The meta–function Θ generates the termination condition (for simplicity of presentation we assume that `Return`, assignments, and `If` conditions do not contain composite terms – the elimination of these by introducing new assignments can be done as in the definition of Γ).

Definition 4

1. $\Theta[P] = \big(\underset{\bar{x}:I_f}{\forall} \Theta[\{\bar{x} \to \bar{x_0}\},\ \mathbb{T},\ P]_{\{\bar{x}_0 \leftarrow \bar{x}\}}\big) \implies \underset{\bar{x}:I_f}{\forall} \pi[\bar{x}]$
2. $\Theta[\sigma,\ \Phi,\ \langle \texttt{Return}[\gamma]\rangle \smile P] = \big(\Phi \Rightarrow \pi[\bar{x_0}]\big)$
3. $\Theta[\sigma,\ \Phi,\ \langle v := \gamma\rangle \smile P] = \Theta[\sigma \circ \{v \to \gamma\sigma\},\ \Phi,\ P]$
4. $\Theta[\sigma,\ \Phi,\ \langle v := h[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\},\ \Phi,\ P]$
5. $\Theta[\sigma,\ \Phi,\ \langle v := f[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to y\},\ \Phi \wedge O_f[\overline{\gamma}\sigma, y] \wedge \pi[\overline{\gamma}\sigma],\ P]$
6. $\Theta[\sigma,\ \Phi,\ \langle v := g[\overline{\gamma}]\rangle \smile P] = \Theta[\sigma \circ \{v \to y\},\ \Phi \wedge O_g[\overline{\gamma}\sigma, y],\ P]$
7. $\Theta[\sigma,\ \Phi,\ \langle \texttt{If}[\varphi,\ P_T,\ P_F]\rangle \smile P] = \bigwedge \begin{cases} \Theta[\sigma,\ \Phi \wedge \varphi\sigma,\ P_T \smile P] \\ \Theta[\sigma,\ \Phi \wedge \neg\varphi\sigma,\ P_F \smile P] \end{cases}$

One single formula is generated, which uses a new constant symbol π standing for an arbitrary predicate. The function Θ operates similarly to Γ by inspecting all possible branches and collecting the respective `If` conditions (in Φ). Moreover it collects the characterizations by output conditions of the values produced by calls to additional functions (Definition 4.6), including for the currently defined function f. However, in the case of f (Definition 4.5) one also collects the condition $\pi[\overline{\gamma}\sigma]$ – that is the arbitrary predicate applied to the current symbolic values of the arguments of the recursive call to f. On each branch, the collected conditions are used as premise of $\pi[\bar{x}_0]$, and then the conjunction of all these clauses (after reverting to free variables $\bar{x}$) is universally quantified over the input condition and is used as a premise in the final formula.

Example: Primitive recursive functions: If $\Sigma[P]$ is

$$q[x] \Rightarrow f[x] = s[x], \quad \neg q[x] \Rightarrow f[x] = c[x, f[r[x]]],$$

then the termination condition is:

$$\big(\underset{x:I_f}{\forall} (q[x] \Rightarrow \pi[x]) \wedge ((\neg q[x] \wedge \pi[r[x]]) \Rightarrow \pi[x])\big) \ \Rightarrow \ \underset{x:I_f}{\forall} \pi[x].$$

Note that in this formula π a new symbol standing for an arbitrary predicate. As in illustration, when $I_f[x]$ is "x natural", $q[x]$ is $x = 0$ and $r[x]$ is decrement, then this is the usual induction over natural numbers.

The termination condition is always an implication between two formulae quantified over all x satisfying I_f, with the right-hand side being as above. The left-hand side of the termination condition is a (quantified) conjunction of implications having $f[x]$ on the right-hand side, each implication corresponding to one branch of the program. Namely, if the program branch is represented in $\Sigma[P]$ by $p_i[x] \Rightarrow (f[x] = g_i[x])$, then the corresponding implication in the termination condition is: $(p_i[x] \wedge R_i[x]) \Rightarrow \pi[x]$, where $R_i[x]$ is the conjunction of a number of atoms of the form $\pi[t]$, one for each occurrence of $f[t]$ in $g_i[x]$.

The rule above is applicable only for programs which do not contain nested recursions (that is, terms of the form $f[\ldots, f[\ldots], \ldots]$), and also no occurrences of f in the `If` conditions. Otherwise, f would occur in the termination condition, which we do not want to allow. In this case one can eliminate the un-wanted occurrences of f by using new (universally quantified) variables pre-conditioned by the output condition O_f – as in the function Γ above.

We conjecture that (for terminating programs) the formula $\underset{f}{\exists!} \underset{x:I_f}{\forall} \Sigma[P]$ is a logical consequence of the the above termination condition. To prove the existence of f appears to be a challenge, however we were able to develop the proof for the general case of primitive recursive functions, as well as for some particular examples with nested occurrences of P (the McCarthy 91 function, matching of expressions).

The uniqueness is a straightforward consequence of the termination condition, by considering $\pi[x]$ as $f_1[x] = f_2[x]$, where f_1 and f_2 are two functions. The property π holds for $q[x]$ and by using the termination condition we prove that holds also in the case $\neg q[x]$. Therefore π holds for all x.

Furthermore, if we assume the existence of f, then, by setting $\pi[x]$ to $I_f[x] \Rightarrow O_f[x, f[x]]$, one obtains in a straightforward way a proof of the total correctness formula $\underset{x}{\forall} I_P[x] \Rightarrow O_P[x, P[x]]$ from the other verification conditions.

3 Implementation and Example

Our prototype implementation (`FwdVCG`) is based on the theoretical aspects presented in the previous sections. It is built on top of the computer algebra system *Mathematica* and uses *Theorema* building blocks and procedural language for writing imperative programs.
We illustrate with the following code fragment:
Example:

```
Program["GCD", GCD[↓ a, ↓ b]],
 Module[{}],
  If[a = 0,
   Return[b]];
  If[b ! = 0,
   If[a > b,
    a := GCD[a - b, b],
    a := GCD[a, b - a]]];
 Return[a]],
Pre → a ≥ 0 ∧ b ≥ 0,
Post→ ∃k1 ∃k2 ((a = k1 ∗ y) ∧(a = k2 ∗ y))
```

The automatically generated verification conditions for the previous example are the following:

$$(a \geq 0 \wedge b \geq 0) \wedge (a = 0) \Rightarrow \underset{k1}{\exists} \underset{k2}{\exists} ((a = k1 * b) \wedge (b = k2 * b)) \tag{1}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \Rightarrow a \geq b \tag{2}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \Rightarrow a - b \geq 0 \wedge b \geq 0 \tag{3}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a > b \wedge a \geq b \wedge (a - b \geq 0 \wedge b \geq 0) \wedge \underset{k1\,k2}{\exists\,\exists}((a - b = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1})) \Rightarrow \underset{k1\,k2}{\exists\,\exists}((a = k1 * \mathrm{y1}) \wedge (b = k2 * \mathrm{y1})) \tag{4}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \Rightarrow a \geq b \tag{5}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \Rightarrow a \geq 0 \wedge b - a \geq 0 \tag{6}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge a \geq b \wedge (a \geq 0 \wedge b - a \geq 0) \wedge \underset{k1\,k2}{\exists\,\exists}((a = k1 * \mathrm{y2}) \wedge (b - a = k2 * \mathrm{y2})) \Rightarrow \underset{k1\,k2}{\exists\,\exists}((a = k1 * \mathrm{y2}) \wedge (b = k2 * \mathrm{y2})) \tag{7}$$

$$(a \geq 0 \wedge b \geq 0) \wedge a \neq 0 \wedge \neg(b \neq 0) \Rightarrow \underset{k1k2}{\exists\,\exists}((a = k1 * a) \wedge (b = k2 * a)) \tag{8}$$

Remark 2. 1. Each verification condition is universally quantified; the bound variables are: $a, b, y1, y2$;
2. The function GCD can not occur in the verification conditions thus their occurrences are replaced by the variables $y1$ and $y2$ (e.g. (4), (7));
3. The input condition of the basic function ,,-"has to be fulfilled thus verification conditions are generated at this aim (e.g. (2), (5)).
4. The program output has the expressions: b (in (1)), $y1$ (in (4)), $y2$ (in (7)), a (in (8)).

The termination condition is:

$$\Big(\underset{\substack{a,b\\ a\geq 0, b\geq 0}}{\forall} \bigwedge \begin{cases} a = 0 \Rightarrow \pi[a, b] \\ (a \neq 0 \wedge b \neq 0 \wedge a > b \wedge \pi[a - b, b]) \Rightarrow \pi[a, b] \\ (a \neq 0 \wedge b \neq 0 \wedge a \not> b \wedge \pi[a, b - a]) \Rightarrow \pi[a, b] \\ (a \neq 0 \wedge b = 0) \Rightarrow \pi[a, b] \end{cases}\Big) \Longrightarrow \Big(\underset{\substack{a,b\\ a\geq 0, b\geq 0}}{\forall} \pi[a, b]\Big)$$

4 Conclusion and Future Work

The method presented in this paper combines forward symbolic execution and functional semantics for generating the verification conditions necessary for checking the imperative program correctness. We implemented it in the *Theorema* system and we tested it on programs written in our mini-programming language.

For checking the validity of the verification conditions we built a prototype of a simplifier that reduces the verification conditions to (systems of) equalities and inequalities. The simplified formulae were obtained using first order logic inference rules, truth constants and simple algebraic simplifications.

As future work we want to apply and automate advanced algebraic and combinatorial algorithms and methods in order to check their validity. A promising approach seems to be the Fourier-Motzkin Elimination method ([7]), especially for systems of inequalities with small number of constraints and variables because of the time complexity reasons.

References

1. D. Coward, *Symbolic execution systems – a review*, Softw. Eng. J. **3** (1988), no. 6, 229–239.
2. L. Deutsch, *An interactive program verifier*, Ph.D. thesis, University of California - Berkley, USA, 1973.
3. G. Dromey, *Program derivation: the development of programs from specifications*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.
4. J. King, *A new approach to program testing*, Proceedings of the international conference on Reliable software (New York, NY, USA), ACM Press, 1975, pp. 228–233.
5. J. Loeckx, K. Sieber, and R. Stansifer, *The foundations of program verification*, John Wiley & Sons, Inc., New York, NY, USA, 1984.
6. J. McCarthy, *A Basis for a Mathematical Theory of Computation*, Computer Programming and Formal Systems (P. Braffort and D. Hirschberg, eds.), North-Holland, Amsterdam, 1963, pp. 33–70.
7. A. Schrijver, *Theory of linear and integer programming*, John Wiley & Sons, Inc., New York, NY, USA, 1986.
8. R. Topor, *Interactive Program Verification using Virtual Programs*, Ph.D. thesis, University of Edinburgh, Scotland, 1975.