Verification of Imperative Programs using Symbolic Execution and Forward Reasoning in the *Theorema* system

Mãdãlina Eraşcu, Tudor Jebelean*

Research Institute for Symbolic Computation,
Johannes Kepler University, Linz, Austria
{merascu,tjebelea}@risc.uni-linz.ac.at

Extended abstract

We present our work in progress concerning the verification of imperative programs in the *Theorema* system. Given an imperative program P and its specification (input condition formula I_P and output condition formula O_P), we are interested in finding the answer to the question: *Is the program P correct with respect to its specification?* To answer this question we use a method which is based on forward reasoning [4, 3], symbolic execution [2, 5] and functional semantics [7].

The program P is represented as a term at meta-level; it contains formulae and terms from the theory Υ in predicate logic and constructs corresponding to the imperative language statements.

For writing imperative programs in *Theorema* system, we use some commands for the user interface role [6] (`Program, Pre, Post, FwdVCG`), which allow the definition of programs together with their specification, and imperative language constructs:

- abrupt termination statements (*return*)
- assignments - besides *simple constant* and *variable assignment, function* (including *recursive call*) *call* is also handled
- conditionals: *If with one* and *two branches*

Example
```
Program["MyFactorial", Fact[↓ n],
 Module[{fact}] ,
  If [n == 0,
   fact := 1,
```

* *The Theorema project is supported by FWF (Austrian National Science Foundation) SFB project F1302. 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.*

```
 fact := n*Fact[n - 1]];
 Return[fact]
],
Pre → n ≥ 0
Post → ∀_{m<n} (m | ¢out)
```

Although our programming language contains only the return statement, assignments and conditionals, it is still Turing complete because the presence of *conditionals* and *recursive call statement* assures the computability property.

A very important aspect towards the verification of programs is represented by the fact that the programs have to be well-formed. From our point of view, a program is well-formed if every branch has a *return statement* and every *variable* which is used in the program is *initialized*.

Definition 1. *A program P is represented as a tuple of statements.*

We express the well-formed property of the programs by defining the predicate $IsProgram$. In the following, we consider the symbol x standing for an input variable, $t \in \mathcal{T}$ a term from the set of terms, $v \in \mathcal{V}$ a variable from the set of variables, $V \subset \mathcal{V}$ a set of initialized variables and φ a formula. The function $Vars$ gives the set of initialized variables occurring in a term or in a formula.

Definition 2.

(2.1) $IsProgram[P] \Longleftrightarrow IsProgram[\{x\}, P]$

(2.2) $IsProgram[V, \langle \text{Return}[t] \rangle \smile P] \Longleftrightarrow Vars[t] \subseteq V$

(2.3) $IsProgram[V, \langle v := t \rangle \smile P] \Longleftrightarrow \bigwedge \left\{ \begin{array}{l} Vars[t] \subseteq V \\ IsProgram[V \cup \{v\}, P] \end{array} \right.$

(2.4) $IsProgram[V, \langle \text{If}[\varphi, P_T] \rangle \smile P] \Longleftrightarrow \bigwedge \left\{ \begin{array}{l} IsProgram[V, P_T \smile P] \\ IsProgram[V, P] \end{array} \right.$

(2.5) $IsProgram[V, \langle \text{If}[\varphi, P_T, P_F] \rangle \smile P] \Longleftrightarrow \bigwedge \left\{ \begin{array}{l} IsProgram[V, P_T \smile P] \\ IsProgram[V, P_F \smile P] \end{array} \right.$

The definition (2.1) of the predicate $IsProgram$ suggests that x is the only initialized variable. Next we describe the well-formed property of the program P for each kind of statement which is encountered.

Briefly describing the predicate $IsProgram$, we can *return* (2.2) a term t if the variables composing it are initialized. An *assignment* (2.3) is performed just with initialized variables. The result of an assignment is a new variable v which can be used further in the computations (more precisely in the computations which appear in the body of P).

The two definitions of the predicate $IsProgram$, corresponding to *conditionals* ((2.4) and (2.5)) are quite similar and express that P_T, P_F and P have to be also well-formed programs.

The *semantics* of a program is of the following form:

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

(We consider the simplified case with just one input variable, denoted by the symbol x. Although, our implementation handles also the case with more than one input variable.)
Each $i \in \{1, ..., n\}$ corresponds to a branch (path) of the program.
The symbol x stands for the input variable, x_0 for its symbolic value, $p_i[x]$ is a conjunction of first order logic formulae representing the path condition, $f[x]$ is the program function and $g_i[x]$ is the expression of $f[x]$. The expression of $g_i[x]$ is expressed using variable, constants and functions from the theory Υ.

Definition 3.

(3.1) $F[P] = \underset{x}{\forall}\big(F[\{x \to x_0\}, P]\{x_0 \to x\}\big)$

(3.2) $F[\sigma, \langle \text{Return}[t]\rangle \smile P] = (f[x_0] = t\sigma)$

(3.3) $F[\sigma, \langle v := t\rangle \smile P] = F[\sigma \circ \{v \to t\sigma\}, P]$

(3.4) $F[\sigma, \langle \text{If}[\varphi, P_T]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \varphi\sigma \Longrightarrow F[\sigma, P_T \smile P] \\ \neg\varphi\sigma \Longrightarrow F[\sigma, P] \end{array} \right.$

(3.5) $F[\sigma, \langle \text{If}[\varphi, P_T, P_F]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \varphi\sigma \Longrightarrow F[\sigma, P_T \smile P] \\ \neg\varphi\sigma \Longrightarrow F[\sigma, P_F \smile P] \end{array} \right.$

In a first step, the program semantics (3.1) is constructed by considering the set of replacements (with the shape $\{var \to expr\}$) for all the input variables of the program P. In the simplified case considered by us, the initial substitution is $\{x \to x_0\}$. After the whole program P is processed, we transform the constant value x_0 into an arbitrary x (by universaly quantifying the formula).

When a *return statement* is encountered (3.2), the program function has the value of the term t, value taken from the substitution σ. The *assignment statement* (3.3) updates the substitution with the variable v. When a *conditional* ((3.4) and (3.5)) is encountered, we have two different expressions for the program function: one when the formula $\varphi\sigma$ holds, one in the opposite case. The program function is not defined if P is the empty tuple or a tuple which does not contain the *return* statement.

Program verification using symbolic execution represents an approach in which the concrete values of the variables are replaced with symbolic values. Thus, our programs are executed on a certain class of inputs (the symbolic value can represent any arbitrary value from the class of inputs).

We apply symbolic execution in order to obtain the verification conditions for imperative programs. A symbolic run, from our point of view, uses the elements: *state, assumptions (path condition)* and *verification conditions.*

A state is a substitution specifying the values of all the initialized variables. We also call the state the *substitutions set* (σ). The values occurring in this set are symbolic.

The *assumptions* are a set of conditions which the inputs must satisfy in order to reach the respective branch. These conditions are obtained by analyzing the algorithm body. Some additional assumptions are obtained from the correctness property of the functions which already belong to the theory and are used in the algorithm. The correctness property means that the output condition is

satisfied if the input is satisfied. As a remark, the assumptions are expressed using symbolic values (constant values in this aim) and they are logical formulae.

The *verification conditions set* (Φ) is represented by the logical formulae which are generated at the end of a branch. The end of a branch is reached when a *return statement* is encountered. Additionally, some supplementary verification conditions are generated, those required by the input and output conditions of the functions used in the program.

Each symbolic execution begins with the following configuration:

- the initial state is formed with substitutions only for the input variables
- the initial set of assumptions contains only the formula standing for the input condition, expressed using the symbolic values for input variables
- the verification conditions set is empty

Let G be the function generating the verification conditions, I_h, O_h - predicates describing the precondition (input condition), respectively the postcondition (output condition) of a function h.

Definition 4.

(4.1) $G[P] = \underset{x}{\forall}\big(G[\{x \to x_0\}, \{I_P[x_0]\}, P]\{x_0 \to x\}\big)$

(4.2) $G[\sigma, \Phi, \langle \text{Return}[t]\rangle \smile P] = \Big(\Phi \Rightarrow O_P[x_0, t\sigma]\Big)$

(4.3) $G[\sigma, \Phi, \langle v := \gamma\rangle \smile P] = G[\sigma \circ \{v \to \gamma\sigma\}, \Phi, P]$

(4.4) $G[\sigma, \Phi, \langle v := h[\overline{\gamma}]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} \Phi \Rightarrow I_h[\overline{\gamma}\sigma] \\ G[\sigma \circ \{v \to h[\overline{\gamma}\sigma]\}, \Phi \cup \{I_h[\overline{\gamma}\sigma], O_h[\overline{\gamma}\sigma, h[\overline{\gamma}\sigma]]\}, P] \end{array} \right.$

(4.5) $G[\sigma, \Phi, \langle \text{If}[\varphi, P_T]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} G[\sigma, \Phi \cup \{\varphi\sigma\}, P_T \smile P] \\ G[\sigma, \Phi \cup \{\neg\varphi\sigma\}, P] \end{array} \right.$

(4.6) $G[\sigma, \Phi, \langle \text{If}[\varphi, P_T, P_F]\rangle \smile P] = \bigwedge \left\{ \begin{array}{l} G[\sigma, \Phi \cup \{\varphi\sigma\}, P_T \smile P] \\ G[\sigma, \Phi \cup \{\neg\varphi\sigma\}, P_F \smile P] \end{array} \right.$

The verification conditions for the program P are generated as follows. We start from the program P and, at the first step (4.1), we create the substitution $\sigma_0 = \{x \to x_0\}$ and the path condition $\{I_P[x_0]\}$. In the next steps each statement of the program P is processed. The function G has now three arguments: the state σ, the set of assumptions generated before the statement s is processed and the part of the program $\langle s\rangle \smile P$. When a *return statement* is encountered (4.2), the execution of the program stops and a verification condition is generated: the path condition must imply the postcondition of the program P. The formula representing the postcondition depends on the input variable and the term returned via the *return statement*.

We have two definitions corresponding to *term assignment* ((4.3) and (4.4)): in the case of a simple assignment (4.3) no additional verification conditions should be generated. When a function call (4.4) appears on the right-hand side of the assignment, we have to generate an intermediary verification condition - the existing assumptions must imply the input condition of the function. Next,

we update the substitution σ, the set of accumulated conditions Φ and the tuple P of statements. We add to the set Φ the assumptions represented by the precondition (input condition) formula of the function h (whose values for the arguments are taken from the σ) and the postcondition (output condition) formula of the function h (with updated values for the arguments). If these two conditions hold then we perform the assignment, without violating any pre-, postconditions. In this case the substitution σ is updated. We denote by $\gamma \in \mathcal{T}$ a term, by $\overline{\gamma} \in \mathcal{T}$ a sequence of terms and by $h \in \mathcal{F}$ a function with the arrity n.

The call of the *conditionals* ((4.5) and (4.6)) changes the set Φ and the program P. In the case of *If with one branch*, the set of assumptions is updated, by adding the condition $\varphi\sigma$ to the existing set of assumptions Φ if the execution of the program follows the $True$ branch or $\neg\varphi\sigma$ if the execution follows the $False$ branch. In the first case the $P_T \smile P$ branch is processed, in the other case only the P branch.

In the case of *If with two branches*, the only thing different from the previous expression of G is that the tuple $P_T \smile P$, respectively $P_F \smile P$ have to be processed in the cases when $\varphi\sigma$, respectively $\neg\varphi\sigma$, holds.

Following these theoretical foundations, we used the computer algebra system Mathematica [8] for developing a tool which provides the verification conditions for the imperative programs in an automatic manner. We will check the validity of these verification conditions (first order logic formulae) using an algebraic-logic simplifier, special syntax constructs for writing them (like *Definition, Lemma*) and we will prove them (by applying various simplifiers, solvers and provers from the *Theorema* library)[1].

References

[1] 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** (2006), no. 4, 470–504 (english).

[2] P. David Coward, *Symbolic Execution Systems - a review*, (1988).

[3] G. Dromey, *Program derivation: the development of programs from specifications*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[4] K. Sieber J. Loeckx, *The Foundations of Program Verification*, Wiley - Teubner, 1984.

[5] J. King, *Symbolic Execution and Program Testing*, (1976).

[6] M. Kirchner, *Program Verification with the Mathematical Software System Theorema*, Tech. Report 99-16, July 1999.

[7] J. Stroy, *The Scott-Strachey approach to programming language theory*, MIT Press, 1977.

[8] S. Wolfram, *The Mathematica Book*, Wolfram Media, 2003.