PROGRAM VERIFICATION BY CALCULATING RELATIONS

Jacques Carette*, Ryszard Janicki* and Yun Zhai*†
Department of Computing and Software
McMaster University
Hamilton, Ontario, Canada L8S 4K1
{carette,janicki,zhaiy}@mcmaster.ca

Abstract *We show how properties of an interesting class of imperative programs can be verifed by means of relational modelling and symbolic computation.*

Keywords *Modelling, Proving Program Correctness, Symbolic Computation, Recurrence Relations*

1 Introduction

In late sixties and early seventies, a technique for verifications and analysis of computer programs based on a calculus of relations was proposed ([1, 2, 8, 15] and others). Despite many theoretical and methodological advantages (it rather emphasises *calculation* insted of *proving*), the technique has never become widely accepted because of the huge amount of symbolic computations that need to be performed for even relatively simple cases.

The situation has dramatically changed today, as we have very powerful tools supporting symbolic computation such as *Maple* [5] and *Mathematica* [18]. The problem is still non-trivial, as the most general cases can be proved to be undecidable, but for many less general cases an efficient solution seems to be feasible.

In this paper we show how to build a Maple [5] based tool [19] that can either automatically computer a closed form for simple programs with loops, or considerably simplify that task by computing *polynomial* invariants of such programs. Simple cases of recursion can also be treated. For straight-line programs, this reduces to a technique called *symbolic execution*. The main idea behind symbolic execution is to use symbolic expressions as input values and to simulate the execution of the program statements on this symbolic inputs. The formal specification of our system was done using *Maude* [14].

Symbolic execution has wide range of potential applications, however, it has been rarely used for proving properties of programs ([16] is one of few exceptions). This is because, in general, naïve symbolic execution can lead to exponential blow-ups.

Our symbolic analysis can be seen as a kind of compiler which can translate the input programs into a symbolic expression, and then can transform this expression into an output expression. From our point of view, recursion and looping are essentially equivalent, and so we will mainly restrict ourselves to loops as the source of our main difficulties [9]. The basic technique used in such cases is to find "loop invariants" proposed by C. A. R. Hoare in 1969 [6]. Unfortunately finding them is often problematic and research on how to find them in some automatic manner has only just begun [12, 16].

We will show that for many frequently occuring loops, finding invariants is not necessary as the symbolic expression for the output can be generated explicitly by solving the recurrence equations generated from the loop. Even if, due to structural complexity of a loop, finding loop invariants is necessary, the technique we have proposed might often help substantially.

2 Intuition and Motivation

The example below (from [2]) provides a motivation and illustrates well the main ideas. In principle we first translate a program into a *relational expression* and then we will try to obtain the program properties by analysing this relational expression. The full theory of those expressions can be found in [13].

Consider the well-known procedure factorial, written in a small subset of Maple [5]:

```
factorial:=proc(n::posint)
local i, fac;
   i:=1;
   fac:=1;
   while i < n do
   begin
      i:=i+1;
      fac:=fac*i;
   end;
   return fac
end proc;
```

Since n does not change its value in the above program we may consder it as a constant, so we may assume the above program has two integer variables i and *fac*. Define $D = \mathbb{Z} \times \mathbb{Z}$ and denote the elements of D as (i,fac). Each instruction can be modelled by a function $F_i : D \to D$, $i = 1, 2, 4, 5$, in the following manner:

"`i:=1`" corresponds to $F_1(i,\mathit{fac}) = (1,\mathit{fac})$,
"`fac:=1`" corresponds to $F_2(i,\mathit{fac}) = (i,1)$,
"`i:=i+1`" corresponds to $F_4(i,\mathit{fac}) = (i+1,\mathit{fac})$,

*Partially Supported by NSERC of Canada Grant.
†Supported by Ontario Graduate Scholarship.

and `"fac:=fac*i"` maps to $F_5(i,\mathit{fac}) = (i, \mathit{fac}\cdot i)$.
The test `"i<n"` can be modelled by two partial identity functions, $I_3, \bar{I}_3 : D \to D$, where I_3 models `"i<n"`, and $\bar{I}_3$ models its complement, i.e. `"i≥n"`. More precisely,
`"i<n"` corresponds to $I_3(i,\mathit{fac})$, and
`"i≥n"` corresponds to $\bar{I}_3(i,\mathit{fac})$, where

$$I_3(i,\mathit{fac}) = \begin{cases} (i,\mathit{fac}) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases}$$

$$\bar{I}_3(i,\mathit{fac}) = \begin{cases} (i,\mathit{fac}) & \text{if } i \geq n \\ \perp & \text{if } i < n \end{cases}$$

It is a well known fact that non-recursive programs can be modelled adequately with Kleene Algebras of Relations with Tests (see [13]) by using the following scheme. Let R, R_1, R_2 be relations (each function is a relation!) that model the program statements `S`, `S1`, `S2`, respectively. Let `T` be a test modelled by I_T and $\bar{I}_T$, and let the symbols "$\circ$" and "$*$" denote the (forward) composition of relations, and transitive and reflexive closure of relations (Kleene star), respectively. Then :
`"S1;S2"` is modelled by $R_1 \circ R_2$,
`"if T then S1 else S2"` is modelled by
$(I_T \circ R_1) \cup (\bar{I}_T \circ R_2)$,
and `"while T do S"` by $(I_T \circ R)^* \circ \bar{I}_T$.
Using this scheme one can easily model the above program by writing the following relational expression:

$$F = F_1 \circ F_2 \circ (I_3 \circ F_4 \circ F_5)^* \circ \bar{I}_3$$

Calculating "$\circ$" is easy, but calculating "$*$" is not. Let $G = I_3 \circ F_4 \circ F_5$. Then we have:

$$\begin{aligned} G(i,\mathit{fac}) &= (I_3 \circ F_4 \circ F_5)(i,\mathit{fac}) \\ &= F_5(F_4(I_3(i,\mathit{fac}))) \\ &= \begin{cases} (i+1, \mathit{fac}\cdot(i+1)) & \text{if } i < n \\ \perp & \text{if } i \geq n \end{cases} \end{aligned}$$

Similarly :

$$\begin{aligned} G^2(i,\mathit{fac}) &= G(G(i,\mathit{fac})) \\ &= \begin{cases} (i+2, \mathit{fac}\cdot(i+1)\cdot(i+2)) & \text{if } i+1 < n \\ \perp & \text{if } i+1 \geq n \end{cases} \end{aligned}$$

Hence :

$$G^k(i,\mathit{fac}) = \begin{cases} (i+k, \mathit{fac}\cdot(i+1)\cdot(i+2)\dots(i+k)) \\ \qquad\qquad \text{if } i+k-1 < n \\ \perp \qquad \text{if } i+k-1 \geq n \end{cases}$$

Since G^* is *not* a function, we need to express G^k in terms of the relation calculus:
$(i,\mathit{fac})G^k(i',\mathit{fac}') \iff i' = i+k \;\wedge$
$\quad \mathit{fac}' = \mathit{fac}\cdot(i+1)\cdot\ldots\cdot(i+k) \wedge i+k-1 < n.$
We have $G^* = \bigcup_{i=0}^{\infty} G^i$, hence:
$(i,\mathit{fac})G^*(i',\mathit{fac}') \iff \exists k \geq 0, (i,\mathit{fac})G^k(i',\mathit{fac}')$
$\iff \exists k, 0 \leq k < n-i+1 \;\wedge\; i' = i+k \;\wedge$
$\qquad \mathit{fac}' = \mathit{fac}\cdot(i+1)\cdot\ldots\cdot(i+k).$
We may now make some simplification:
$(F_1 \circ F_2)(i,\mathit{fac}) = F_2(F_1(i,\mathit{fac})) = (1,1).$
This means:
$(i,\mathit{fac})F_1 \circ F_2 \circ G^*(i',\mathit{fac}') \iff (1,1)G^*(i',\mathit{fac}')$
$\iff \exists k, 0 \leq k < n \wedge i' = k+1 \wedge \mathit{fac}' = (k+1)!$
Let us calculate : $(1,1)G^* \circ \bar{I}_3(i',\mathit{fac}')$.
The partial function $\bar{I}_3$ in the relational representation looks as follows:
$(i,\mathit{fac})\bar{I}_3(i',\mathit{fac}') \iff i \geq n \wedge i = i' \wedge \mathit{fac} = \mathit{fac}'$. From the definition of " $\circ$ " we have :
$(1,1)G^* \circ \bar{I}_3(i',\mathit{fac}') \iff \exists i, \mathit{fac}, (1,1)G^*(i,\mathit{fac}) \;\wedge$
$(i,\mathit{fac})\bar{I}_3(i',\mathit{fac}') \iff (\exists k, 0 \leq k < n \wedge i = k+1 \wedge \mathit{fac} = (k+1)!) \wedge (i \geq n \wedge i = i' \wedge \mathit{fac} = \mathit{fac}')$.

Note that $i = k+1 \wedge i \geq n \Rightarrow k+1 \geq n \iff k \geq n-1$, and $0 \leq k < n \wedge k \geq n-1 \Rightarrow k = n-1 \iff n = k+1$.
So now, we do not have a general $\exists k$, but a very specific k=n-1, which means $G^* \circ \bar{I}_3$ is a function again, and the statement $(\exists k, 0 \leq k < n \wedge i = k+1 \wedge \mathit{fac} = (k+1)!) \wedge (i \geq n \wedge i = i' \wedge \mathit{fac} = \mathit{fac}')$ is reduced to:

$$i' = k+1 = n-1+1 = n \wedge \mathit{fac}' = n!$$

In this way we have proved that $F(i,\mathit{fac}) = (n, n!)$, so the program is correct. To make this technique feasible for bigger, more realistic programs, we need a tool that would be able to do all those symbolic calculations. Our tool [19] will take the text of the program `factorial` as an input and will return the text "$n!$" as output. In the next sections we will show how it can be done with some help from *Maple* [5]. Our system [19] can also deal with some kind of limited recursion as well.

3 Loops and Recurrence Equations

A formula that expresses the meaning of a loop can be explicitly derived (in some cases) by solving appropriate recurrence equations.

Consider our program `factorial`. Every time the loop is executed, the value of i is incremented by one and the value of *fac* is incremented i times. We may express this change in a form of recurrences. For this example, the recurrence relation is the following $i(k+1) = i(k) + 1, \quad \mathit{fac}(k+1) = \mathit{fac}(k)\cdot i(k+1)$, where $i(k+1)$ and $\mathit{fac}(k+1)$, for $k \geq 0$ are the values of i and *fac* at the end of iteration $k+1$. In this sense, k represents *time*, which is the important new concept in this representation. Because *time* is explicitly reified in the recurrence, this allows many techniques from symbolic computation to apply. Before entering the loop, the value of i is 1 and the value of *fac* is 1, so we initialize the recurrence by $i(0) = 1$ and $\mathit{fac}(0) = 1$. Hence the following recurrence equations describe the meaning of our loop.

$$\begin{gathered} i(0) = 1; \qquad \mathit{fac}(0) = 1; \\ i(k+1) = i(k)+1; \quad \mathit{fac}(k+1) = \mathit{fac}(k)\cdot i(k+1); \end{gathered}$$

These recursive function can not be solved by Maple [5] directly since they are *non-linear* recurrences. However, we can clearly see that $i(k+1)$ is independent of $\mathit{fac}(k)$, but $\mathit{fac}(k+1)$ is not independent of $i(k+1)$, i.e. we may not be able to solve the system directly, but we might be able to solve it incrementally. From a simple data-flow analysis, this order can be determined. If we solve for $i(k)$ first (including initial conditions), we get $i(k) = k+1$. Replacing $i(k)$ by $k+1$, in the equation for *fac*, we get $\mathit{fac}(k+1) = \mathit{fac}(k) \cdot (k+2)$, which is also solveable. Putting the solution together, we get

$$i(k) = k+1 \tag{3.1}$$
$$\mathit{fac}(k) = (k+1)! \tag{3.2}$$

Note that the above solution is still in terms of *time*; however, regardless of whether the loop terminates or not, we have found the *core meaning* of the loop!

This technique can be applied to any loop if the set of appropriate recurrence equations that the loop defines is essentially triangular, with polynomial solutions for the non-linearities (or it can be transformed into such a system).

4 Loop Termination

To determine the value of recurrence variables after the loop, we need the recurrence condition which *symbolically* determines the number of iterations for the recurrence. In our example, the recurrence variables are i and *fac*, and the recurrence condition is given by $i(k) < n$, which, together with equation (3.1) gives us the following formula for the loop termination (see [17]):

$$\min_{k\geq 0}\{k \mid i(k) \geq n\} = \min_{k\geq 0}\{k \mid k+1 \geq n\} = n-1$$

In order to compute the value of *fac* at the loop exit, we have to substitute $n-1$ for k in (3.2). So, we get "$n!$".

Note that it is only necessary to obtain a closed-form solution to the recurrences involving for those variables which actually *occur* in the stopping condition to determine loop termination. This can frequently be much simpler, as is the case for all *for* loops!

5 Input Language and Relation Generator

The language for input programs (like `factorial`) was chosen as a subset of Maple [5], and it is a combination of the following statements

1. `var:=expr`
2. `if T then S1 else S2`
3. `while T do statement end do`
4. `for i from i1 to i2 by i3 do S end do`
5. *Recursive Function Calls*

In our system, we have two main modules: the *Relation Generator* and *Relation Solver*, both written in Maple [5]. This is made especially easy since Maple has some very powerful reflection capabilities through its `ToInert` function, which gives an accurate AST representation for any Maple program (or expression). The first one generates a series of appropriate relations (recurrence, state transition, etc.) for the given input program, while the second one solves the relations and produces an output expresion, either in an explicit form, or, if an explicit form cannot be found, then implicit forms (like invariants) are returned.

The *Relation Generator* is a total function – it translates the given input program into a sequence of appropriate relations. Table 1 shows the relationship between parts of program and names of the relations used. Table 2 shows what is generated for our program `factorial`.

Program Statements	Relations Generated
`var:=expr`	`StateTransition`
`if-then-else`	`Piecewise`
`while C do`	`Fixedpoint([C],...)`
`for-from-to-by-do`	`Fixedpoint(For(),...)`
Recursion	`RecursionCall`

Table 1. Rules for program transformation

Input Program	Relations
`factorial:=proc(n)`	
`i:=1`	`F1:StateTransition(i,1)`
`fac:=1`	`F2:StateTransition(fac,1)`

`while i < n do`
`    i:=i+1`
`    fac:=fac*i`
`end do`
	`F3:FixedPoint([i < n],`
	`[StateTransition(i,i+1),`
	`StateTransition(fac,fac*i)]`
	`)`

`fac`	`F4:fac`
`end proc;`	

Table 2. Translation of `factorial` into the set of appropriate relations

6 Solving Relations: Overview

The method for solving relations depends on the kind of function that generates them. The technique described in this and almost all remaining chapters is a refinement and generalisation of many results from [3, 12, 17]. Of course, if code does not contain either loops nor recursion, from a symbolic point of view such straight-line code is completely trivial, and we can simply compute the result. The only drawback is that such an answer can be exponentially larger than the input program.

For the case where we have either a `while` or `for` loop whose body is straight-line code, we generate a system of recurrence equations, which we try to solve in closed-form, using whatever triangular structure we may find. Using similar ideas, we can also generate systems of recurrences for programs containing recursion.

When loops contain branches (i.e. `if-then-else`), the resulting system of reccurences essentially **never** falls within a class of solvable recurrence.

Relations	Recurrence and Initial Condition
`StateTransition(i,1)`	*Initial Condition:* $i(0) = 1$
`StateTransition(fac,1)`	*Inital Condition:* $fac(0) = 1$
`FixedPoint(i< n,`	*Loop Termination:* $t = min\{k \geq 0 \mid i(k) \geq n\}$
`StateTransition(i,i+1),`	*Recurrence:* $i(k+1) = i(k) + 1$
`StateTransition(fac,fac*i), )`	*Recurrence:* $fac(k+1) = fac(k) \cdot i(k+1)$
`return fac`	$fac(t)$

Table 3. Recurrence Equation and Initial Conditions for `factorial`.

At present, we immediately shift to generating implicit results, in the form of polynomial invariants [12].

7 Generating Recurrences: `while`

If the input program is a simple `while` loop, without `if-then-else` statements inside the `while` loop, the core relation we generate will be `FixedPoint`. Table 3 shows the results for our `factorial` program. In this case our system [19] will produce the output formula "n!".

7.1 Generating Recurrence Relations

In this case all we have inside the loop are assignment statements which are represented by `StateTransition` relations. These relations might however be *mixed*, in other words a variable at time $k + 1$ might occur on both the left and right hand sides. This occurs in our `factorial` code, where *fac* depends on $i(k+1)$ rather than $i(k)$. However, a simple program transformation related to *Static Single Assignment (SSA)* form [20] takes care of this issue.

7.2 Generating Initial Conditions

The initial conditions are easily determined: they are the values of each of the loop variables (i.e. those which change) right before the loop starts. These can be determined by unwinding the stack of `StateTransition` calls preceding the loop. This is always possible, though might again generate very large answers.

7.3 Stopping Conditions

If we want to find the actual stopping condition for a loop, we need to solve (symbolically) the recurrence equations (with known initial functions) just generated. Suppose the solution is:

$$v_1(k) = F_1(k), v_2(k) = F_2(k), \ldots, v_m(k) = F_m(k).$$

Now, we can decide the loop stop condition on the basis of the condition C in `FixedPoint([C],...)`. In general, this condition reads

$$\min_{k \geq 0}\{k \mid \neg C\}$$

R	$<$	$\leq$	$>$	$\geq$	$=$	$\neq$
$\neg R$	$\geq$	$>$	$\leq$	$<$	$\neq$	$=$

Table 4. Boolean symbols and their opposite values

where C depends on the state variables at time k. In general, this is a diophantine equation, and thus well-known to be unsolvable. But in many practical situations, the actual equations are simple. We draw attention to three such cases.

- *Case 1* $C = v_i \; R \; c$, where $v_1, \ldots, v_m$ are the recurrence variables, c is constant with respect to the v_i's, and R is a relational operator from Table 4. The converse of R can easily be computed explicitly, also shown in Table 4. Assuming that the expression for $v_i = F_i(k)$ is simple enough (in terms of k), this can be solved in closed form.
- *Case 2* $C = v_i \; R \; \phi(v_j)$, where $v_1, \ldots, v_m$ are the recurrence variables, and some of them occur in the expression ϕ, with R as before.

$$\begin{aligned} z &= \min_{k \geq 0}\{k | \neg(v_i(k) \; R \; \phi(v_j(k))\} \\ &= \min_{k \geq 0}\{k | F_i(k) - \phi(v_j(k)) \; S \; 0\} \end{aligned}$$

where $S = \neg R$.
- *Case 3* C is a conjunction of terms which satisfy *Case 1* or *Case 2*. Then we can simply take the minimum of all the conjuncts.

8 Solving with `for loops`

Since a `for` loop is a special case of a `while` loop, this case is very similar to the previous. Generating recurrences is exactly the same. Assume the following pattern for the `for`: loop

```
for i from s to e by step do S.
```

8.1 Generating Initial Conditions

Initial functions are generated from the stack of `StateTransitions` preceding the loop for all variables, with the addition of $i(0) = s$.

8.2 Number of Loop Iterations

We have to consider two cases:
- *Case 1* The variable i, i.e. loop counter, *is not modified by* S. In this case the number of iterations z can be solved explicitly and uniformly for all cases, and is given by

$$z = \left\lfloor \frac{e - s + 1}{step} \right\rfloor$$

- *Case 2* The variable i, i.e. loop counter, *is modified by* S. In this case we have to transform the loop `for` into an equivalent `while` loop and proceed as in Chapter 7.3.

Input program	Relations
`chebyshev:=proc(n)`	
`u0:=1`	`F1:StateTransition(u0,1)`
`u1:=x`	`F2:StateTransition(u1,x)`

`for i from 2 to n-1`
`do u0:=u1`
`u1:=-u0+2*x*u1`
`end do`
	`F3:Fixedpoint(For([i,2,n-1,1]),`
	`[StateTransition(u0,u1),`
	`StateTransition(u1,-u0+2*x*u1)]`
	`)`

`return -u0+2*x*u1`	`F4:  -u0+2*x*u1`
`end proc`	

Table 5. Translation of `chebyshev` into the set of appropriate relations

Relations	Recursive and Initial Functions
`StateTransition(u0,1)`	*Initial Function*: $u0(1) = 1$
`StateTransition(u1,x)`	*Initial Function*: $u1(1) = 1$

`FixedPoint(`
`For([i,2,n-1,1]),`
`StateTransition(u0,u1),`
	Loop Termination:
	$t = \left\lfloor \frac{n-1-2+1}{1} \right\rfloor = n-2$

	Recursive Function: $u0(k+1) = u1(k)$

`StateTransition`
`(u1, -u0+2*x*u1),`
`])`
	Recursive Function:
	$u1(k+1) =$
	$-u0(k+1) + 2 \cdot x \cdot u1(k)$

`-u0+2*x*u1`	$-u0(t) + 2 \cdot x \cdot u1(t)$

Table 6. Recursive and Initial Functions for `chebyshev`

9 Solving Relations involving recursion

If a recursive function is correctly defined, it defines both recurrence functions and initial conditions in quite natural way. However, we can not always solve (symbolically) the recurrence equations thus generated (for instance we cannot do it for Ackerman function).

Note that it is important here to assume that we have a *meaningful* program, as otherwise a recursively defined function might come equipped with naturally defined initial conditions.

10 The case of branches in loops

When we have `if-then-else` inside a `while`, we usually are not able to generate explicit symbolic output. We can often generate implicit output, or invariants, in a way similar to that desribed in [12, 16]. More details can be found in [19].

11 Example 2: Chebyshev Polynomials

The program `factorial` involves `while` loop. We show a simple example with `for` loop.

```
chebyshev:=proc(n)
local i, u0, u1, t;
    u0 := 1;
    u1 := x;
    for i from 2 to n-1 do
        u0 := u1;
        u1 := -u0+2*x*u1;
    end do;
   return -u0+2*x*u1
end proc:
```

Translation of the program into set of appropriate relations in given in Table 5, whilst recursive functions, initial functions, and loop termination condition is in Table 6.

Our system [19] has produced the following output for the above program `chebyshev`:

$$-x(-1+2x)^{(n-3)} + 2x^2(-1+2x)^{(n-2)}.$$

12 Related work

Symbolic execution has been studied since seventies, however with different goals than ours. King [11] in 1976 has developed EFFIGY, a symbolic execution system with a fixed number of integers.

Kemmerer and Eckmann [10] have presented an approach to symbolic execution based on the concept of path expressions and path conditions.

DISSECT [7] and SELECT [4] are another symbolic execution systems that use the path conditions concept. DISSECT can be used to symbolically execute some simple FORTRAN programs. The main purpose of SELECT [4] is to complement mechanical program verification and debug programs.

Rodríguez-Carbonell and Kapur [16] have recently developed some interesting techniques for automatic finding loop invariants.

13 Conclusions and Future Work

We have described a symbolic execution system that can be used to analyse properties of programs. The system performs a symbolic execution of the input program and get either an explicit or implicit symbolic output. The system [19] can handle assignment statetements, `if-then-else` statements, `for-do` statements, and `while-do` statements, the latter with some restrictions.

Despite the restrictions, it can be used for a huge variety of programs, including programs like Binomial Coefficients, Bessel Functions, Greater Common Divisor, etc. [19].

For the future work, we would like to loosen the restriction for `while-do` loop we have now. We also would like to be able to produce explicit symbolic solutions in some cases where now we can only produce invariants.

References

[1] H. Bekić, Definable operations in general algebras and the theory of automata and flowcharts, Unpublished Manuscript, IBM Laboratory, Vienna 1969.

[2] A. Blikle, An anlysis of programs by algebraic means, In A. Mazurkiewicz, Z Pawlak (eds), *Mathematical Foundation of Computer Science*, Banach Center Publications, Vol. 2, pp. 167–213, Polish Scientific Publishers, Warsaw 1977.

[3] T. E. Cheatham, J. A. Townley, Symbolic Evaluation of Programs: A look at Loop Analysis, *Proc. of ACM Symposium on Symbolic and Algebraic Computation*, 1976, pp. 90-96.

[4] B. Elspas, R. S. Boyer, K. N. Levitt, SELECT-A formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices, 10(6)*, pages 234–245, June 1975.

[5] M. B. Monagan and K. O. Geddes and K. M. Heal and G. Labahn and S. M. Vorkoetter, *Maple Programming Guide*, Springer Verlag, 1998.

[6] C. A. R. Hoare, An Axiomatic Basis of Computer Programming, *Comm. of ACM* 12 (1969), 576-580.

[7] W. E. Howden. Symbolic Testing and the DISSECT Symbolic Evaluation System. *IEEE Trans. on Software Engineering SE-3, 4*, pages 266–278, July 1977.

[8] R. Janicki, Analysis of Coroutines by Means of Vectors of Coroutines, *Fundamenta Informaticae*, 2, 2 (1979), 289-316.

[9] A. Kaldewaij, *Programming. The Derivation of Algorithms*, Prentice-Hall 1990.

[10] R. A. Kemmerer, S. T. Eckmann. UNISEX: A UNix-based symbolic Executor for Pascal. *Softw. Pratt. Exper. 15,5*, pages 439–457, May 1985.

[11] J. C. King, Symbolic Execution and program testing. *Communications of the ACM*, pages 385–394, July 1976.

[12] L. I. Kovács, T. Jebelean, Automated Generation of Loop Invariants by Recurrence Solving in *Theorema*, *Proc. of SNASC'04* (Symbolic and Numeric Algorithms for Scientific Computing).

[13] D. Kozen, A completeness theorem for Kleene algebras and the algebra of regular events, *Information and Computation* 110 (1994), 366-390.

[14] *The Maude System*, http://maude.cs.uiuc.edu

[15] A. Mazurkiewicz, Proving algorithms by tail function, *Information and Control*, 18 (1971) 793-798.

[16] E. Rodríguez-Carbonell, D. Kapur, *Program Verification Using Automatic Generation of Invariants*, Proc. of ICTAC'05, *Lecture Notes in Computer Science* 3407, Springer 2005, pp. 325-340.

[17] B. Scholz, T. Fahringer, *Advanced Symbolic Analysis for Compilers*. Springer-Berlin, 2003.

[18] S. Wolfram, *The Mathematica Book*, Cambridge University Press, 1999.

[19] System website or Yun Thesis, hidden for reviewing process.

[20] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman and F. K. Zadeck, *Efficiently computing static single assignment form and the control dependence graph*, ACM Trans. Program. Lang. Syst., Vol 13, Number 4, 1991, ACM Press, pp. 451–490.