ISSN 1512-1232

POSSIBILITY OF FUNCTIONAL PROGRAMS VERIFICATION THROUGH APPLICATION OF MODEL CHECKING

Natela Archvadze

Department of Computer Sciences Faculty of Exact and Natural Sciences
Ivane Javakhishvili Tbilisi State University, Tbilisi, GEORGIA
Natela.Archvadze@tsu.ge

Merab Pkhovelishvili

Department of Programming N.Muskhelishvili Computing Mathematic Institute
of Georgian Technical University, Tbilisi GEORGIA, merab5@list.ru

Abstract

The overall, ubiquitous computerization of all patterns of life makes the citizenry more and more vulnerable and dependent to the technology created by human. In this regards, the achievement of the required quality of the system software for the critical application is becoming the most significant scientific and technical challenge.

The verification of the software has the huge potential in solution of this problem. Verification considers the formal checking of formal requirements execution towards the software behavior, presented in the type of the formal model.

Recent breakthrough in the verification researches is connected to the Model Checking method that considers that the validity of the temporary logic formula, describing the requirements towards the software behavior, is checked on the basis of the program model.

This article hereby describes the Model Checking methodology application possibilities for the verification of the functional programs, namely for Lisp and Haskell Programs.

***Keywords**: Functional Programming Languages; Recursion Forms; Programs Verification*

I. INTRODUCTION: ACTUALITY OF THE VERIFICATION PROBLEM

Almost every day the information on software errors is disclosed and spread. Hereby we present several samples [1], having occurred in 2010: the accident in Mexico gulf could be probably caused due to software error [2]; company Apple recognized 4 errors regarding iPhone 4 connections quality; carrier rocket “Proton-M” with the satellite “Glonass-M” deviated from the assigned route due to the errors in mathematic software (main reason – incorrectly written formula in documentation pertaining to filling the acceleration block by the oxygen); Japanese probe “Akatsuki” could not depart to Venire orbit.

In average, 10 errors come on 1000 lines of the code within the modern software systems, 1-4 errors come on 1000 lines of high-quality software. The modern PU contains millions of codes. Already passed software programs are full of errors.

One of the stages of the verification is validation, the significance of which is confirmed by the fact that 80% of the costs attributable to the installed software falls at validation. Validation means the process of checking the conformity of the software with the consumer requirement. The software errors not detected during the validation stage lead to the huge material damage and human casualty.

During the recent decade, the complicatedness of the computer systems has reached the critical threshold: more and more sophisticated software is required, but the programing technology is unable to properly process them.

Verification is one of the methods aimed at software quality improvement (along with validation). Verification is the formal confirmation that the formal system meets the formally determined requirements.

Approximately 45 years ago, the verification direction appeared – it was deduction verification theorem-proving [3]. By it, only not-large systems correctness can be confirmed, and with significant efforts.

Example: in 2009, the Australian research center declared about the completion of confirmation of formal correctness of operating system kernel through Isabel system by NICTA. The operational system code included 7500 lines. 10000 theorems (200000 lines) have been formally confirmed, consuming thereof 4 years and 12 researchers. This was “extraordinary result”, achieved through the theorem-proving.

The complicatedness of confirmation of correctness through application of Theorem – Proving 30 times exceeds the complicatedness of the code; thus, through its application it is fairly easy to confirm the correctness of the program having several lines only.

II. MODEL CHECKING – DISCRETE DYNAMIC SYSTEMS VERIFICATION

Verification method – Model Checking [4,1] – is one of the most perspective and widely applied attitude, through which the programs automated functioning and correctness problems can be solved.

In 2007, Turing Prize was rewarded to three creators of Model Checking technique: Edmund M. Clarke (CMU), E. Allen Emerson (U Texas, Austin) and Joseph Sifakis (Verimag, France) for “their roles to transform the Model Checking method to the highly effective technology of verification, widely applied in PU processing and machinery tools industry”.

Model Checking operates with the formal models and checks some of their qualities. These qualities should be expressed by any of the logics. The ordinary logic is unable to express that dynamic.

A. Logic for PU requirements formalization Application of Modality

Propositional logic [5] of the statement is inadequate as far as the statements are static, not changeable across the time, added by the connuction non-commutation (A&B ≠ B&A). The statements within the classical logic are not formalized. Systems subject to verification develop across time; however, the ordinary logic is inadequate to express their qualities. Due to this, the tense is introduced into the logical statements.

The modalities (Tense Logic, A.Prior, Y1950) are applied, the operators of which are as follows: Fq – q will occur sometimes in future; Pq – q already occurred sometimes in past; Gq – q will always present in future, Hq – q has always been in past; pUq – q will occur in future, before that p is always executed; Xp - p will occur in the next moment. Through these operators, we can express the natural language sentences by such logic.

Even in 1977, A. Pnuel has introduced LTL discrete tense logic [6]. Tense logic described the sequence (yesterday, today, tomorrow…). The “Probable Universes” semantics has been applied herein. In each universe, each logical formula is either true or false, which is fair for any temporary formula as well.

ISSN 1512-1232

B. Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) is also determined here. In this formula, ϕ is:

- Atomic position (atomic predicate) p, q, ...,
- or by logical operators $\vee$, $\neg$ related formulas
- or by temporal operators U, related X (the past modality is not in LTL)

Grammar $\phi ::= p \mid \phi \vee \phi \mid \neg \phi \mid X\ \phi \mid \phi\ U\ \phi$

LTL basis = $\{\neg, \vee, U, X\}$

In LTL logic, only two modal operator - Until and NextTime – are added.

C. LTL and discrete systems specification

The sequence of "universes" within the LTL can be imagined as the never-ending sequence of the discrete systems positions. As for the dependence, it can be considered as availability, as the discrete, non-divided transitions of the systems. In such moment, atomic predicates describe the basic qualities of the process.

Any temporal formula is the calculation in future, the process dynamics: qUp is ended while calculation; Gr – is not ended.

D. Kripke's Structure

Kripke's structure, as fixed-length system with the transformation pointed positions as well as non-pointed positions, might be applied for the program modeling. Actually, the position is the program state; transition means the execution by the program operators, and the atomic positions means some statements we are interested in. For instance a>b, two specific processes stated in the critical interval, etc.

The general scheme of the formal verification through application of the model checking can be explained as follows: the Kripke's structure will be applied as the formal model through which we intend to verify; the requirements specifications are expressed on the temporal logic language. Model Checking is the program, the algorithm, enabling us to check the execution level of the quality expressed by the temporal logic across the Kripke's structure [7].

E. Kripke's Strucure, as the Program Model

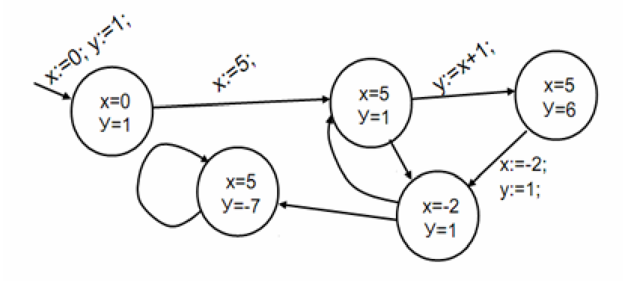
Program state means the vector of its variable values as well as notch, while transition means the change of the program variables by the operators or/and notches.

Let us discuss the McCarty's sample, where z is pre-determined:

```
begin
x:=0; y:=1;
```

```
while x+z<5 do
{ x:=5;
if z=1 then y:=x+1;
x:=-2; y:=1;
}
y:=x*y-5; x:=5;
end
```

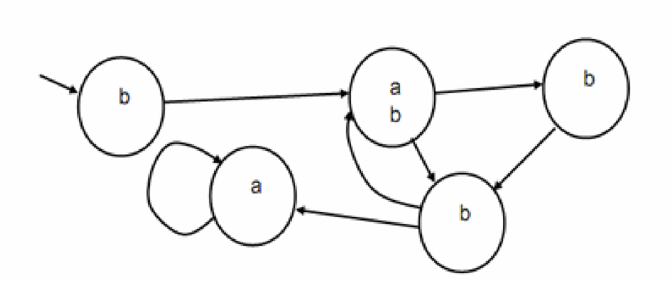
Let us develop the formal model to be applied for verification. Only two variables – x and y – determine the state, z is not defined in this block, thus it cannot determine the state. The state will be developed into the angels, and those operators that change the state will be positioned on sides:



Please note that we do not focus our attention on the operators' semantics within this scheme but on the states and transformation. The transformation to one state can be done from several states and vice versa, from one state transformation is possible to several directions. Return is possible to the angle from which the transition occurred, or directly, or by passing other states. The cycle operator's actions are reflected through such manner.

Let us suggest that the atomic positions we are interested in are as follows: a=x>y; b=|x+y|<3. This means that these are those formulas (in general, they might be temporal) according to which the verification should be implemented.

Let us transform the McCarty's sample into the relevant Kripke's structure. For this purpose: a) variable with the false value (ex: x=0), and: b) variable which does not change the value (ex:x=5) will not be included in the following position. We will receive the following scheme of Kripke:



Hereby the Model Checking is already functioning, over which the following general scheme is applied for verification: from one side, we have the system, on another – system specifications. As far as only formally determined model can be formally confirmed, we need two models: 1. System formal model, same specifications, which will be presented by the Kripke's structures; 2. Requirement specifications essential for description of the system specifications, same formal language, which will be presented through the temporal logic formulas. According to the procedure by Model Checking, the conclusion will be developed: 1. Yes, system meets the specifications; or: 2. No, the system cannot meet the specifications as confirmed by the counter-sample, over which it is executed.

F. *Model Checking application samples*

Let us name the projects across which the Model Checking has been successfully applied:

- Cambridge ring protocol;
- IEEE Logical Link Control protocol, LLC 802.2 ;

- Part of XTP and TCP/IP protocol;
- Protocols of penetration to splint;
- Protocols for error control within the machinery;
- Cryptography protocols ;
- Ethernet Collision Avoidance Protocol;
- DeepSpace1 (NASA) – the critical errors gave been detected upon its passing;
- SLAM: Microsoft Model checking Driver Development Kit Driver for Windows.

III. MODEL CHECKING FOR FUNCTIONAL PROGRAMS

Let us set the task as follows: to apply the Model Checking for the functional programs verification, namely, initially for one of the classical functional language Lisp; afterwards - for commercial and widely spread language Haskell; and finally - for strong, multi-paradigm F# that works starting from Microsoft Visual Studio 2010.

The general problem of solution of this task is the fact that within the functional languages there is no state, i.e. there is no variable description, linking the memory cell to the variable, assigning operator, changing the value in the memory cell, etc.

The functional program is the recursive function, or combination of the functions. The program written according to the imperative paradigms, i.e. procedural language can be generally formulated as follows: we gave the initial state, operators' sequence which changes one state into another and we have the resulted state. According to this analogy, functional program can be formulated as follows: we have the initial argument, operators which change the argument and we have result, i.e. resulted value of the argument. The main idea is to investigate which arguments the function has, and which type of the actions are implemented for changing this argument to apply the formal methods.

The special positions within the functional programing languages are occupied by the lists. List is the abstract type of the data which presents the ordered multiplicity of the values, whereas several values can be met more than once. Hereinafter we will discuss only those functions the arguments of which are lists.

Thus, the sequence of the states of the calculation processes within the functional programing can be considered as the list.

A. *Recursion functions representation means*

For the repeated calculations, the recursion is applied within the functional programing instead of standard cycle operators. Recursion functions recall themselves, thus enabling the multiple implementations of the operations. For recursion, the large capacity stack is required; however, this can be avoided through application of tail area recursion. Hereby we discuss the samples of the recursion functions creating the program over the Lisp:

```
(defun MEMBER (item list)
  (cond ((null list) nil)
        ((eql(car list) item) list)
             (t (MEMBER item
                        (cdr list)))))

(defun negnums (l)
   (cond ((null l) nil)
         ((< (car l) 0)
               (cons (car l)
                       (negnums (cdr l))))
                   (t (negnums (cdr l)))))
 (defun reverse (x)
        (if (eq x nil) nil
        (if (atom x) x
            (append (reverse (cdr x))
        (cons (reverse (car x)) nil)) )))
```

Please note that the function determination style is similar. As a rule, the conditional (if) and branching (cond) operators are applied. We propose one condition for the operators that determined the actions to be implemented in case of emptying the list. The second condition includes the recursion access, which is done either on list tail-area, or list top, i.e. generally, as F0 function on the given argument. In [9-12] we describe the general form over which we can adopt the Lisp recursion functions, reflected in the tail area recursion:

```
(DEFINE FUN(F F0.L)
(COND((MEMBER NIL L) a)
     (T (g(f(M F L))
     (APPLY FUN(CONS  F
            (CONS F0(M F0 L)))))) ))
```

Let us hereby discuss the Haskell program samples:

```
  Length (L) = 0 when L == []
   Length (L) = 1 + Length (tail (L))
  Reverse_all (L) = L when atom (L)
  Reverse_all ([]) = []
  Reverse_all (H:T) = Append
  Reverse_all (T), Reverse_all (H))
  Append ([], L2) = L2
  Append (L1, L2) = prefix (head (L1), Append (tail (L1), L2))
```

The functions determined by the tail-area recursion across Haskell can be presented in the generalized form (template) as follows:

```
  f [ ] = g1 [ ]
  f (x : xs) = g2(g3 x)(g4(f( g5 xs )))
```

In this formula `g1, g2, g3, g4, g5` functions present the functions dependent on the task objectives: `g1` is the function for processing the empty list; `g2` is the function which unifies the results of processing the list top area and tail area; `g3` is the function that processes the list top area, `g4` is the function that processes the recursive recall for tail area of the non-empty lists; `g5` is the function which pre-processes the tail area of non-empty list before the recursion recall.

The program has been implemented (Microsoft Visual Studio 2010, C#), which, at the entry, is granted by the text file – the determination text for the given function across Haskell. The program will then compare this text to the Haskell tail area recursion function template, and as a result will bring back the specific values of g1, g2, g3, g4 and g5 functions.

For instance, let us hereby consider the function Length, which calculates the length of the argument – list. For this function, g1 is constant, which brings back the value 0; function g2 is the operation „+"; function g3 is constant, that brings back value "1", and function g4 and g5 are the similar functions, i.e. they bring back without changes the transferred parameters. Therefore, the description of function `length` can be formulated as follows:

```
g1 _ = 0
g2 a b = a + b
g3 _ = 1
```

```
    g4 x = x
    g5 x = x
    length [] = gl []
    length (x:xs) = g2 (g3 x) (g4 (length (g5 xs))
```

Let us hereby consider the second function `last: [a] -> a,` which brings back the last element from the list. It can be described as follows:

```
    last []    =  error "Prelude.last:                      "
last [x]    =  x
last (_:xs)=  last xs
```

For this function, `g1, g2, g3, g4` and `g5` functions are determined as follows:

```
    gl _ = error
    g2 a b = b
    g3 x = x
    g4 x = x
    g5 x = x
```

Therefore, the description of function `last` can be formulated as follows:

```
    f [] = gl []
    f (x:xs) = g2 (g3 x) (g4 (f (g5 xs)))
```

These types of the recursive functions are applied for tasks other than verification as well. One of such tasks is the automated creation of the program "significant" part according to the data structure.

IV. CONCLUSION

Achieving excellence in program software still remains the topical scientific and technical problem. The verification of the software plays significant role in solving this problem. The verification through application of Model Checking methodology has been considered hereby, according to which the verification means the formal checking of the software actions formal requirements, represented in the type of the formal model. Hereby we also showed that the functional programs verification can be implemented through application of this very methodology if the lists are applied instead of positions which do not exist across functional languages. More specifically, we have applied the tail area recursive functions for verification.

REFERENCES

1. Ю.Г. Карпов. Model Checking. Верификация параллельных и распределенных программных систем. 2010. БХВ-Петербург. С.552
2. Don Shafer, Phillip Laplante. The BP Oil Spill: Could Software be a Culprit? IEEE IT Pro September/October 2010, IEEE, 2010.
3. John Harrison. Theorem Proving for Verification. Intel Corporation, CAV 2008.
4. Edmund M. Clarke Jr., Orna Grumberg , Doron A. Peled. Model Checking .1999, pp.314.
5. Hurley, Patrick (2007). A Concise Introduction to Logic 10th edition. Wadsworth Publishing. p. 392.
6. Amir Pnueli. The temporal logic of programm. Proc. of 18th Anny.Symp. on Foundation of Computer Science, 1977.
7. S.A.Kripke. Semantic conderation on modal logic. Acta Philosofica Fennica, v.16, 1963.

8. V.Berj. Metodi rekursivnix funkcii. (In Russian). Moscow,”Mashonostroenie”, 1983.
9. Archvadze N., Pkhovelishvili M., Shetsiruli L., Nizharadze. Recursion forms and their verification by using the undictive methods. Computing and Computational Intelligence. Proceeding of the 3nd European Computing Conference (ECC’09), Tbilisi, 2009, pp.357-361.
10. Archvadze N., Pkhovelishvili M., Shetsiruli L Problems of Verification of Functional Programs. Bulletin of the Georgian Academy Sciences. Bulleten of the Georgian National Academy of Sciences. vol.3. no.3, 2009. pp 16-19.
11. N. Archvadze, M. Pkhovelishvili, L. Shetsiruli, M. Nizharadze. Program Recursive Forms and Programming Automatization for Functional Languages. WSEAS TRANSACTIONS on COMPUTERS. Volume 8, 2009. ISSN: 1109-2750. pp. 1256-1265.
12. N. Archvadze, M. Pkhovelishvili. Several issues of program verificatio. PCI’2010 The Third International Conference “Problems of Cybernetics and Informatics”, Baky, Azerbaijan. Volume I. pp.71-74.

Article received: 2013-11-15