TESTING OF PROGRAM CORRECTNESS
IN FORMAL THEORY

Ivana Berkovic
University of Novi Sad, Technical Faculty “Mihajlo Pupin”, Zrenjanin, Serbia
berkovic@tf.zr.ac.yu

Branko Markoski
University of Novi Sad, Technical Faculty “Mihajlo Pupin”, Zrenjanin, Serbia
markoni@uns.ns.ac.yu

Jovan Setrajcic
University of Novi Sad, Faculty of Sciences, Novi Sad, Serbia
bora@if.ns.ac.yu

Vladimir Brtka
University of Novi Sad, Technical Faculty “Mihajlo Pupin”, Zrenjanin, Serbia
vbrtka@tf.zr.ac.yu

Dalibor Dobrilovic
University of Novi Sad, Technical Faculty “Mihajlo Pupin”, Zrenjanin, Serbia
ddobrilo@tf.zr.ac.yu

ABSTRACT
Within software’s life cycle, program testing is very important, since quality of specification demand, design and application must be proven. All definitions related to program testing are based on the same tendency and that is to give answer to the question: does the program behave in the requested way? One of oldest and best-known methods used in constructive testing of smaller programs is the symbolic program execution. One of ways to prove whether given program is written correctly is to execute it symbolically. Ramified program may be translated into declarative shape, i.e. into a clause sequence, and this translation may be automated. Method comprises of transformation part and resolution part. This work gives the description of the general frame for the investigation of the problem regarding program correctness, using the method of resolution invalidation. It is shown how the rules of program logic can be used in the automatic resolution procedure. The examples of the realization on the LP prolog language are given (without limitation to Horn’s clauses and without final failure). The process of Pascal program execution in the LP system demonstrator is shown.

Keywords: program correctness, resolution, test information, testing programs

1. INTRODUCTION
The program testing is defined as a process of program execution and comparison of observed behaviour to behaviour requested. The primary goal of testing is to find software flaws [1], and secondary goal is to improve self-confidence in testers (persons performing tests) in case when test finds no errors. Conflict between these two goals is visible when a testing process finds no error. In absence of other information, this may mean that the software is either of very high or very poor quality.

Program testing is, in principle, complicated
process that must be executed as systematically as possible in order to provide adequate reliability and quality certificate.

Within software lifespan, program testing is one of most important activities since fulfillment of specification requirements, design and application must be checked out. According to Mantos [2], big software producers spend about 40% of time for program testing. In order to test large and complicated programs, testing must be as systematic as possible. Therefore, from all testing methods, only one that must not be applied is ad hoc testing method, since it cannot verify quality and correctness regarding the specification, construction or application. Testing firstly certifies whether the program performs the job it was intended to do, and then how it behaves in different exploitation conditions. Therefore, the key element in program testing is its specification, since, by definition, testing must be based on it. Testing strategy includes a set of activities organized in well-planned sequence of steps, which finally confirms (or refutes) fulfillment of required software quality. Errors are made in all stages of software development and have a tendency to expand. A number of errors revealed may rise during designing and then increase several times during the coding. According to [3], program testing stages cost three to five times more than any other stages in a software life span.

In large systems, many errors are found at the beginning of testing process, with visible decline in error percent during mending the errors in the software itself. There are several different approaches to program testing. One of our approaches is given in [4]. Testing result may not be predicted in advance. On the basis of testing results it may be concluded how much more errors are present in the software.

The usual approach to testing is based on requests analyse. Specification is being converted into test items. Apart of the fact that incorrigible errors may occur in programs, specification requests are written in much higher level than testing standards. This means that, during testing, attention must be paid to much more details than it is listed in specification itself. Due to lack of time or money, only parts of the software are being tested, or the parts listed in specification.

Structural testing method belongs to another strategy of testing approaches, so-called "white box" (some authors call it transparent or glass box). Criterion of usual "white box" is to execute every executive statement during the testing and to write every result during testing in a testing log. The basic force in all these testings is that complete code is taken into account during testing, which makes easier to find errors, even when software details are unclear or incomplete.

According to [5] testing may be descriptive and prescriptive. In descriptive testing, testing of all test items is not necessary. Instead, in testing log is written whether software is hard to test, is it stable or not, number of bugs, etc... Prescriptive testing establishes operative steps helping software control, i.e. dividing complex modules in several more simple ones. There are several tests of complex software measurements. Important criterion in measurement selection is equality (harmony) of applications. It is popular in commercial software application because it guarantees to user a certain level of testing, or possibility of so-called internal action [6]. There is a strong connection between complexity and testing, and methodology of structural testing makes this connection explicit [6]. Firstly, complexity is the basic source of software errors. This is possible in both abstract and concrete sense. In abstract sense, complexity above certain point exceeds ability of the human mind to do an exact mathematical manipulation. Structural programming techniques may push these barriers, but may not remove them completely. Other factors, listed in [7], claim that when module is more complex, it is more probable that it contains an error. In addition, above certain complexity threshold, probability of the error in the module is progressively rising. On the basis of this information, many software purchasers define a number of cycles (software module cyclicity, McCabe [8]) in order increase total reliability. On the other hand, complexity may be used directly to distribute testing attempts in input data by connecting complexity and number of errors, in order to aim testing to finding most probable errors ("lever" mechanism, [9]). In structural testing methodology, this distribution means to precisely determine number of testing paths needed for every software module being tested, which exactly is the cyclic complexity. Other usual criteria of "white box" testing has important flaw that may be fulfilled with small number of tests for arbitrary complexity (using any possible meaning of the "complexity") [10].

The program correctness demonstration and the programming of correct programs are two similar theoretical problems, which are very meaningful in practice [11]. The first is resolved within the program analysis and the second within the program synthesis, although because of the connection that exists between the program analysis and the program synthesis it is noticed the reciprocal interference of the two processes. Nevertheless, when it is a matter of the automatic methods that are to prove the correctness and of the

---

1 McCabe, measure based on a number and structure of the cycle.
methods of automatic program synthesis, the difference between them is evident.

In reference [12] describes the initial possibility of automatic synthesis of simple programs using the resolution procedure of automatic demonstration of theorems (ADT), more precisely with the resolution procedure of deduction of answer to request. The demonstration that the request that has a form of (\(\exists x\)W(x)) is the logical consequence of the axioms that determine the predicate W and determinate (elementary) program operators provides that the variable x in the response obtains the value that represents the requested composition of (elementary) operators, i.e. the requested program. The works of Z. Mann, observe in detail the problems of program analysis and synthesis using the resolution procedure of demonstration and deduction of the response.

The different research tendency is axiomatic definition of the semantics of the program language Pascal in the form of specific rules of the program logic deduction, described in the works [14,15].

Although the concepts of the two mentioned approaches are different, they have the same characteristic. It is the deductive system on predicate language. In fact, it is a matter of realization in the special predicate computation that is based on deduction in formal theory. With this, the problem of program correctness is to be related to automatic checkup of (existing) demonstrations regarding mathematical theorems. The two approaches mentioned above and their modifications are based on that kind of concept.

2. DESCRIPTION OF METHOD FOR ONE PASSAGE SYMBOLIC TESTING PROGRAM

The method is based on transformation of given Pascal program, into a sequence of prologue clauses, which comprise axiomatic base for functioning of deductive resolution mechanism in a BASELOG system [10]. For given Pascal program, by a single passage through resolution procedure of BASELOG system, all possible outputs in Pascal program are obtained in a symbolic shape, together with paths leading to every one of them. Both parts, transformation and resolution one, are completely automated and are naturally attached to each other. When a resolution part has finished, a sequence of paths and symbolic outputs is reading out for given input Pascal program. This is a transformation of programming structures and programming operators into a sequence of clauses, being realized by models depending on concrete programming language. Automation covers branching IF-THEN and IF-THEN-ELSE structures, as well as WHILE-DO and REPEAT – UNTIL cyclic structures, which may be mutually nested in each other. This paper gives review of possibilities in work with single-dimension sequences and programs within a Pascal program. Number of passages through cyclic structures must be fixed in advance using counter. During the testing process of given (input) Pascal program both parts are involved, transformation and resolution, in a following way: Transformation part

- ends function by a sequence of clauses, or
- demands forced termination, depending on input Pascal program.

Impossibility of generating a sequence of clauses in transformation part points that a Pascal program has no correct syntax, i.e. that there are mistakes in syntax or in logical structure (destructive testing). In this case, since axiomatic base was not constructed, resolution part is not activated and user is prompted to mend a Pascal program syntax. In the case that transformation part finishes function by generating a sequence of clauses, resolution part is activated with following possible outcomes:

Ra) ends function giving a list of symbolic outputs and corresponding Pascal program routes, or
Rb) ends by message that id could not generate list of outputs and routes, or
Rc) doesn't end function and demands forced termination.

Ra) By comparing symbolic outputs and routes with specification, the user may
- declare a given Pascal program as correct, if outputs are in accordance to specification (constructive testing), or
- if a discrepancy of some symbolic expression to specification has been found, this means that there is a semantic error in a Pascal program (destructive testing) at the corresponding route.

Rb) Impossibility to generate a list of symbolic expressions in resolution part, which means that there is a logical-structural error in a Pascal program (destructive testing).

Rc) Too long function or a (unending) cycle means that there is a logic and/or semantic error in a Pascal program (destructive testing).

In this way, by using this method, user may be assured in correctness of a Pascal program or in presence of syntax and/or logic-structure semantic errors. As opposite to present methods of symbolic testing of the programs, important feature of this method is single-passage, provided by specific property of OL – resolution [11] with marked literals, at which a resolution module in BASELOG system is founded.
3. DEDUCTION IN FORMAL THEORY AND PROGRAM CORRECTNESS

The program verification may lean on techniques for automatic theorem proving. These techniques embody principles of deductive reasoning, same ones that are used by programmers during program designation. Why not use same principles in the automatic synthesis system, which may construct program instead of merely proving its correctness? Designing the program demands more originality and more creativity than proving its correctness, but both tasks demand the same way of thinking. [13]

Structural programming itself helped the automatic synthesis of computer programs in the beginning, establishing principles in program development on the basis of specification. These principles should be guidelines for programmers. In the matter of fact, advocates of structural programming were very pessimistic regarding possibility to ever automatize their techniques. Dijkstra went so far to say that we should not automatize programming even if we could, since this would deprive this job from all delight.

Proving program correctness is a theoretical problem with much practical importance, and is done within program analyse. Related theoretical problem is the design of correct programs that is solved in another way – within program synthesis. It is evident that these processes are intertwined, since analysis and synthesis of programs are closely related. Nevertheless, differences between these problems are distinct regarding automatic method of proving program correctness and automatic method of program synthesis.

If we observe a program, it raises question of termination and correctness, and if we observe two programs we have question of equivalence of given programs. Abstract, i.e. non-interpreted program is defined using pointed graph. From such a program, we may obtain partially interpreted program, using interpretation of function symbols, predicate symbols and constant symbols. If we interpret free variables into partially interpreted program, a realized program is obtained. Function of such a program is observed using sequence executed. Realized program, regarded as deterministic, has one executive sequence, and if it does not exist at all, it has no executive sequence. On the other hand, when the program is partially interpreted, we see several executive sequences. In previously stated program type, for every predicate interpreted it is known when it is correct and when not, which would mean that depending on input variables different execution paths are possible. Considering abstract program, we conclude that it has only one executive sequence, where it is not known whether predicate P or his negation is correct.

According to The basic presumptions of programming logic are given in [14]. The basic relation \{P\}S\{Q\} is a specification for program S with following meaning: if predicate P at input is fulfilled (correct) before execution of program S, then predicate Q at the output is fulfilled (correct) after execution of program S. In order to prove correctness of program S, it is necessary to prove relation \{P\}S\{Q\}, where input values of variables must fulfill predicate P and output variable values must fulfill predicate Q. Since it is not proven that S is terminating, and that this is only presumption, then we may say that partial correctness of the program is defined. If it is proven that S terminates and that relation \{P\}S\{Q\} is fulfilled, we say that S is completely correct. For program design, we use thus determined notion of correctness.

The basic idea is that program design should be done simultaneously with proving correctness of the program for given specifications[15,16]. First the specification \{P\}S\{Q\} is executed with given prerequisite P and given resultant post condition Q, and then subspecifications of \{P\}S\{Q\} type are executed for components S, from which the program S is built. Special rules of execution provide proof that fulfillment of relation \{P\}S\{Q\} follows from fulfillment of relations \{P\}S\{Q\} for component programs S.

Notice that given rules in [9] are used for manual design and manual confirmation of program's correctness, without mention about possibility of automatic (resolution) confirmation methods. If we wish to prove correctness of program S, we must prove relation \{P\}S\{Q\}, where input values of variables must fulfill the formula P and output values of variables must fulfill the formula Q. This defines only partial correctness of program S, since it is assumed that program S terminates. If we prove that S terminates and that relation \{P\}S\{Q\} is satisfied, we say that S is totally correct. Thus designated principle of correctness is used for program designation. Designation starts from specification \{P\}S\{Q\} with given precondition P and given resulting postcondition Q. Formula \{P\}S\{Q\} is written as K(P, S, Q), where K is a predicate symbol and P, S, Q are variables of first-order predicate calculation.

\{P', y\} z : - y \{P\}

we are writing as K(t(P,Z,Y), d(Z,Y), P)...

where t,d are function symbols and P,Z,Y are variables;

...Rules R(τ):

\textbf{P1.}: \{P\}S\{R\} \Rightarrow \{P\}S\{Q\}$

we write: $K(P,S,R) \wedge \text{Im}(R,Q) \Rightarrow K(P,S,Q)$
where \( \text{Im} \) (implication) is a predicate symbol, and 
\( P, S, R, Q \) are variables;

**P2** \( R \Rightarrow P \) \( \{P; S; Q\} \)

\[ \{R\} S \{Q\} \rightarrow \text{we write } \text{Im}(R, P) \land K(P, S, Q) \Rightarrow K(R, S, Q) \]

**P3** \( \{P; S1; R\} \rightarrow \{R\} S2 \{Q\} \)

\[ \{P\} \begin{array}{l} \text{begin } S1; \text{end } \{Q\} \end{array} \]

\( K(P, S1, R) \land K(R, S2, Q) \Rightarrow K(P, s(S1, S2), Q) \)

where \( s \) is a function symbol, and \( P, S1, S2, R, q \) are variables

**P4** \( \{P \land B\} S \{Q\} ; \{P \land \neg B\} S2 \{Q\} \)

\[ \{P\} \text{if } B \text{ then } S1 \text{ else } S2 \{Q\} \]

\( K(k(P, B), S1, Q) \land K(k(P, n(B)), S2, Q) \Rightarrow K(P, \text{ife}(B, S1, S2), Q) \)

where \( k, n, \text{ife} \) are function symbols

**P5** \( \{P \land B\} S \{Q\} ; P \land \neg B \Rightarrow Q \)

\[ \{P\} \text{if } B \text{ then } S \{Q\} \]

\( K(k(P, B), S, Q) \land \text{Im}(k(P, n(B)), Q) \Rightarrow K(P, \text{if}(B, S), Q) \)

where \( k, n, \text{if} \) are function symbols

**P6** \( \{P \land B\} S \{P\} \rightarrow \{P\} \)

\[ \{P\} \text{while } B \text{ do } S \{P \land \neg B\} \]

\( K(k(P, B), S, P) \Rightarrow K(P, \text{wh}(B, S), k(P, n(B))) \)

where \( k, n, \text{wh} \) are function symbols

**P7** \( \{P\} S \{Q\} ; Q \land \neg B \Rightarrow P \)

\[ \{P\} \text{repeat } S \text{ until } B \{Q \land B\} \]

\( K(P, S, Q) \land \text{Im}(k(Q, n(B)), P) \Rightarrow K(P, \text{ru}(S, B), k(Q, B)) \)

where \( k, n, \text{ru} \) are function symbols

Transcription of other programming logic rules is also possible.

Axiom \( A(\tau) \): 
\[
A1 \ K(t(P, Z, Y), d(Z, Y), P)
\]

assigning axiom

Formal theory \( \tau \) is given by \( (\alpha(\tau), F(\tau), A(\tau), R(\tau)) \), where \( \alpha \) is a set of symbols (alphabet) of theory \( \tau \), \( F \) is a set of formulae (correct words in alphabet \( \alpha \)), \( A \) is a set of axioms for theory \( \tau(A \subset F) \), \( R \) is a set of derivation rules for theory \( \tau \). \( B \) is a theorem within theory \( \tau \) if and only if \( B \) is possible to derive within calculus \( k \) from set \( R(\tau) \cup A(\tau) \) (k is a first-order predicate calculus). Let \( S \) be special predicate calculus (first-order theory) with its own axioms \( A(S) = R(\tau) \cup A(\tau) \). This means that derivation of theorem \( B \) within theory \( \tau \) could be replaced with derivation within special predicate calculus \( S \), whose own axioms \( A(S) = R(\tau) \cup A(\tau) \). Axioms of special predicate calculus \( S \) are: \( A(S) = A(\tau) \cup R(\tau) \).

We assume that \( s \) is a syntax unit whose (partial) correctness is being proven for certain input predicate \( U \) and output predicate \( V \).

Within theory \( S \) is being proved

\[ \frac{\text{program} \text{ is partially correct. It is necessary to}}{\text{establish that input and output predicates } U, \ V \text{ are}}{ \text{in accordance with } P_0, Q_0, \text{ and also that } \text{Im} (X_n, Y_n) \text{ is}}{ \text{really fulfilled for domain predicates ant terms. Accordance means } \text{confirmation that } ... \text{ is}}{ \text{valid. } U \Rightarrow P_0, \quad Q_0 \Rightarrow V, \quad X_0 \Rightarrow Y_0, \text{ there}}{ \text{are two ways to establish accordance: manually or}}{ \text{by automatic resolution procedure. Realization of}}{ \text{these ways is not possible within theory } S, \text{ but it is}}{ \text{possible within the new theory, which is defined by}}{ \text{predicates and terms which are part of the program}}{ \text{s and input-output predicates } U, \ V. \text{ Within this}}{ \text{theory } U, \ P, \ Q, \ V, \ X, \ Y \text{ are not variables, but}}{ \text{formulae with domain variables, domain terms and}}{ \text{domain predicates. This method concerns derivation}}{ \text{within special predicate calculus based on}}{ \text{deduction within the formal theory. Thus the}}{ \text{program’s correctness problem is associated with}}{ \text{automatic proving of (existing) proofs of}}{ \text{mathematical theorems.}} \]

The formal theory \( \tau \) is determined with the formulation of \( (S(\tau), F(\tau), A(\tau), R(\tau)) \) where \( S \) is the set of symbols (alphabet) of the theory \( \tau \), \( F \) is the set of formulas (regular words in the alphabet \( S \)), \( A \) is the set of axioms of the theory \( \tau \) (\( A \subset F \)), \( R \) is the set of rules of execution of the theory \( \tau \).

**Deduction (proof) of the formula** \( B \) in the theory \( \tau \) is the final sequence \( B_1, B_2, \ldots, B_n \) (\( B_n \) is \( B \)) of formulas of this theory, of that kind that for every element \( B_i \) of that sequence it is valid: \( B_i \) is axiom, or \( B_i \) is deducted with the application of some rules of deduction \( Ri \in R \) from some preceding elements.
of that sequence. It is said that B is the theorem of the theory \( \tau \) and we write \( \vdash B \) [17].

Suppose \( S(\tau) \) is a set of symbols of predicate computation and \( F(\tau) \) set of formulas of predicate computation. In that case, the rules of deduction \( R(\tau) \) can be written in the form: \( Bi1 \land Bi2 \land ... \land Bik \Rightarrow Bi \) (\( R_i \)) where \( Bik, Bi \) are formulas from \( F(\tau) \).

Now we can formulate the following task:

\[ R(\tau), A(\tau) \vdash B \] if \( \vdash B \) \( \kappa \) \( \tau \) \( (1) \)

\( B \) is theorem in the theory \( \tau \) if and only if \( B \) is deductible in computation \( \kappa \) from the set \( R(\tau) \cup A(\tau) \).

Suppose \( S \) is a special predicate computation (theory of first line) with its own axioms:

\( A(S) = R(\tau) \cup A(\tau) \), (rules of deduction in \( S \) are rules of deduction of computation \( \kappa \)) then it is valid

\[ A(S) \vdash B \] if \( \vdash B \) \( \kappa \) \( S \), so that \( (1) \) can be written:

\[ \vdash B \] \( S \) \( \tau \) \( (2) \)

That means that the deduction of theorem \( B \) in theory \( \tau \) can be replaced with deduction in special predicate computation \( S \), that has its own axioms \( A(S) = R(\tau) \cup A(\tau) \).

Now we can formulate the following task:

The sequence of formulas has been given \( B1, B2, ..., Bn \) (\( Bn \) is \( B, Bi \) different from \( B \) for \( i < n \)) of theory \( \tau \).

Implementation of programs for proving theorems was in the beginning only in mathematics area. When it was seen that other problems could be presented as possible theorems which need to be proven, application possibilities were found for areas as program correctness, program generating, question languages over relation databases, electronic circuits design.

As for formal presentation where theorem is being proven, it could be statement calculus, first-order predicate calculus, as well as higher-order logic. Theorems in statement calculus are simple for contemporary provers, but statement calculus is not expression enough. Higher-order logic is extremely expression, but they have a number of practical problems. Therefore a first-order predicate calculus is probably the most used one.

Regarding technique of automatic theorems proving, most investigations have been done in resolution rules of derivation. Resolution is a very important derivation rule with completeness property.

**Demonstrate that the mentioned sequence is deduction of formula \( B \) in theory \( \tau \).**

One way of solving this problem is to verify that the given sequence corresponds to definition of deduction in theory \( \tau \). The other way is to use \( (1) \), i.e. \( (2) \):

If we demonstrate that

\[ R(\tau), A(\tau) \vdash B \land B2 \land ... \land Bn \] \( \kappa \) \( \tau \) \( (3) \)

that is sufficient for conclusion that \( B1, B2, ..., Bn \) is deduction in \( \tau \).

And also it is sufficient to demonstrate that \( R(\tau), A(\tau) \vdash Bi \), for \( i = 1, 2, ..., n \), with this it is demonstrated \( (3) \).

Demonstration for \( (3) \) can be deducted with the resolution invalidation of the set \n
\[ R(\tau) \cup A(\tau) \cup \{ \neg B1 \lor \neg B2 \lor ... \lor \neg Bn \} \]

or with \( n \) invalidations of sets \( R(\tau) \cup A(\tau) \cup \{ \neg B1 \} \).

Notice that for the conclusion that \( B1, B2, ..., Bn \) is deduction in \( \tau \) it is not enough to demonstrate \( R(\tau), A(\tau) \vdash \left( B1 \land B2 \land ... \land Bn-1 \Rightarrow Bn \right) \) \( \kappa \), i.e. it is not enough to realize resolution invalidation of the set \( R(\tau) \cup A(\tau) \cup \{ B1, B2, ..., Bn-1 \} \cup \{ \neg Bn \} \), because this demonstrate only that \( Bn \) is deductible in \( \tau \) supposing that in \( \tau \) is deduction \( B1 \land B2 \land ... \land Bn-1 \).

Always when \( B1, B2, ..., Bn \) is really deduction in \( \tau \), \( (B1 \land B2 \land ... \land Bn-1 \Rightarrow Bn) \) will be correct, but vice versa is not always valid. It can happen that \( (B1 \land B2 \land ... \land Bn-1 \Rightarrow Bn) \) is deductible in \( \tau \), but that \( B1 \land B2 \land ... \land Bn-1 \) is not deductible in \( \tau \), (see example \( 1' \)).

And also, the demonstration for \( R(\tau), A(\tau) \vdash Bn \), that can be realized with resolution invalidation of the set \( R(\tau) \cup A(\tau) \cup \{ \neg Bn \} \), means that \( Bn \) is theorem in \( \tau \), i.e. that \( Bn \) is deductible in \( \tau \), but this is not enough for the conclusion that \( B1, B2, ..., Bn \) is deduction in \( \tau \) (except for the case that
Finally, here it is necessary to underline that not correspondence to set \( R(\tau) \cup A(\tau) \cup \{~B\} \) does not mean not correspondence to formula \(~B\) as it is, but only in the presence of \( R(\tau) \cup A(\tau) \).

Example 1. Suppose \( A(\tau) \) is: \{r(1,1), r(1,3)\} and \( R(\tau) \) contains three rules of deduction:
\[
\alpha: r(m,n) \Rightarrow r(n,m); \\
\beta: r(m,n) \Rightarrow r(m+1,n+1) ; \\
\gamma: r(m,n) \land r(n,p) \Rightarrow r(n,p)
\]
symmetry correspondence with the next one transitivity
Demonstrate that the sequence \( J \) is: \( r(3,1), r(4,2), r(5,3), r(5,1), r \) is predicate symbol, one correct deduction of formula \( r(5,1) \) in theory \( \tau \).
It is sufficient to demonstrate:
\[
\{\alpha, \beta, \gamma\}, A(\tau) \rightarrow J.
\]

In the next demonstration \( x+1 \) is signed with \( S(x) \) and axioms for ‘the next one’ are added:

1
\( \sim r(3,1) \sim r(4,2) \sim r(5,3) \sim r(5,1) \& \)
9
\( r(1,1) \& r(3,1) \& \)
\( \sim r(X1,Y1) r(Y1,X1) & \)
\( \sim r(X1,Y1) \Rightarrow (S(X1),U1) \Rightarrow (S(Y1),V1) r(U1,V1) \& \)
\[ \Rightarrow (S(1),2) \& (S(2),3) \& (S(3),4) \& (S(4),5) \& \]
Demonstration with invalidation:
number of generated resolvents = 934
maximum level = 10
DEMONSTRATION IS PRINTED
level on which empty composition is generated = 10
LEVEL=6; central composition
\[
\Rightarrow \sim r(3,1) \sim r(4,2) \sim r(5,3) \sim r(5,1) \&
\]
5.side, 3.lateral :
\[ \Rightarrow \sim r(X1,Y1) \Rightarrow r(Y1,Z1) r(X1,Z1) \& \]
LEVEL=2; resolvent:
\[ \Rightarrow r(3,1) \& \]
LEVEL=3; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \sim r(5,3) \& \]
4.side, 4.lateral :
\[ \Rightarrow r(X1,Y1) \Rightarrow (S(X1),U1) \Rightarrow (S(Y1),V1) r(U1,V1) \& \]
LEVEL=4; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \sim r(5,3) \sim r(X1,Y1) \Rightarrow (S(X1),5) \sim r(Y1,U1) \sim (S(Y1),3) \& \]
7.side, 1.lateral :
\[ \Rightarrow (S(2),3) \& \]
LEVEL=5; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \sim r(5,3) \sim r(5,1) \& \]
9.side, 1.lateral :
\[ \Rightarrow (S(4),5) \& \]
LEVEL=6; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \& \]
4.side, 4.lateral :
\[ \Rightarrow r(X1,Y1) \Rightarrow (S(X1),U1) \Rightarrow (S(Y1),V1) r(U1,V1) \& \]
LEVEL=7; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \sim r(X1,Y1) \Rightarrow (S(Y1),1) \Rightarrow r(U1,V1) \& \]
6.side, 1.lateral :
\[ \Rightarrow (S(1),2) \& \]
LEVEL=8; resolvent:
\[ \Rightarrow r(3,1) \sim r(4,2) \sim r(X1,Y1) \Rightarrow (S(Y1),2) \& \]
8.side, 1.lateral :
\[ \Rightarrow (S(3),4) \& \]
LEVEL=9; resolvent:
\[ \Rightarrow r(3,1) \& \]
2.side, 1.lateral :
\[ r(3,1) \& \]
LEVEL=10; resolvent: & DEMONSTRATION IS PRINTED

Example 2
begin
p:=x;
i:=0;
while i<=n do
begin
i:=i+1;
p:=p*i;
end;
end.

Given program is written:
\[
s(s(d(p,x),d(i,0)),w(i<=n,s(d(i,i+1),d(p,p*i))))
\]
Constant b is a mark for predicate i<=n
constant t1 is i+1, constant t2 is term p*i
thus we obtain
\[
s(s(d(p,x),d(i,0)),w(b,s(d(i,t1),d(p,t2))))
\]
1
\[ /O(X1,V1) \sim K(X1,s(h,g),Y1) \sim K(Y1,w(b,s(d(i,i+1),d(p,p*i)))) \]
8
\[ \sim K(Y1,d(p,x),V1) K(Y1,h,V1) & /reserve for shortening the note \]
\[ \sim K(Y1,d(i,0),V1) K(Y1,g,V1) & /reserve for shortening the note \]
\[ \sim K(X1,Y1,U1) \sim K(U1,Y2,V1) K(X1,s(Y1,Y2),V1) \& /sequence rule \]
\[ \sim K(k(X1,V2),U0,X1) K(X1,w(V2,U0),k(X1,ng(V2))) & /rule for while \]
\[ K(t(X1,Z1,Y1),d(Z1,Y1),X1) & /assigning axiom \]
\[ \sim IM(X2,Y1) \sim K(Y1,w(V2,U0),K(X1,s(Y1,Y2),V1) \& /sequence rule \]


~IM(Y1,V1)~K(X1,U0,Y1)K(X1,U0,V1) / consequence rule
~O(X1,V1) / negation addition
0
0

LP system generates next negation
number of resolvents generated = 10
maximal obtained level = 11
DEMONSTRATION IS PRINTED
level where the empty item is generated = 11
LEVEL=1; central item
/O(X1,V1)~K(X1,s(h,g),Y1)~K(Y1,w(b,s(d(i,t1),d(p,t2))),V1)&
4.lateral, 2.literal :
~K(k(V1,ng(b)),X1)K(X1,w(V2,U0),k(X1,ng(V2)))&
LEVEL= 2; resolvent:
/O(X1,k(X0,ng(b)))~K(X1,s(h,g),X0)~K(X0,w(b,s(d(i,t1),d(p,t2))),k(X0,ng(b)))~K(k(X0,b),s(d(i,t1),d(p,t2)),X0)&
3.lateral, 3.literal :
~K(X1,Y1,U1)~K(U1,Y2,V1)K(X1,s(Y1,Y2),V1)&
LEVEL= 3; resolvent:
/O(X1,k(V1,ng(b)))~K(X1,s(h,g),V1)~K(V1,w(b,s(d(i,t1),d(p,t2))),k(V1,ng(b)))~K(k(V1,b),s(d(i,t1),d(p,t2)),V1)&
5.lateral, 1.literal :
K(t(X1,Z1,Y1),d(Z1,Y1),X1)&
LEVEL= 4; resolvent:
/O(X1,k(X0,ng(b)))~K(X1,s(h,g),X0)~K(X0,w(b,s(d(i,t1),d(p,t2))),k(X0,ng(b)))~K(k(X0,b),s(d(i,t1),d(p,t2)),X0)&
6.lateral, 3.literal :
~IM(X2,Y1)~K(Y1,U0,V1)K(X2,U0,V1)&
LEVEL= 5; resolvent:
/O(X1,k(X0,ng(b)))~K(X1,s(h,g),X0)~K(X0,w(b,s(d(i,t1),d(p,t2))),k(X0,ng(b)))~K(k(X0,b),s(d(i,t1),d(p,t2)),X0)&
5.lateras, 1.literal :
K(t(X1,Z1,Y1),d(Z1,Y1),X1)&
LEVEL= 6; resolvent:
/O(X1,k(X0,ng(b)))~K(X1,s(h,g),X0)~K(X0,w(b,s(d(i,t1),d(p,t2))),k(X0,ng(b)))~K(k(X0,b),s(d(i,t1),d(p,t2)),X0)&
5. lateral, 2.literal :
~IM(k(X1,b),t(t(X1,p,t2),i,t1))/O(X2,k(V1,ng(b)))/
~K(X2,s(h,g),V1)~K(X2,h,U1)~K(U1,g,V1)&
2.lateral, 2.literal :
~K(Y1,d(i,t1),V1)K(Y1,g,V1)&
LEVEL= 8; resolvent:
/O(X1,k(X0,ng(b)))~K(X1,s(h,g),X0)~K(X0,w(b,s(d(i,t1),d(p,t2))),k(X0,ng(b)))~K(k(X0,b),s(d(i,t1),d(p,t2)),X0)&
5. lateral, 1.literal :
K(t(X1,Z1,Y1),d(Z1,Y1),X1)&
LEVEL= 9; resolvent:
/O(Y1,k(X1,ng(b)))/
~K(Y1,s(h,g),V1)~K(Y1,w(b,s(d(i,t1),d(p,t2))),k(Y1,ng(b)))/
4. lateral, 2.literal :
~IM(k(X1,b),k(V1,ng(b)))~K(X1,s(h,g),X1)&
LEVEL= 10; resolvent:
/O(Y1,k(X1,ng(b)))/
~K(Y1,s(h,g),V1)~K(Y1,w(b,s(d(i,t1),d(p,t2))),k(Y1,ng(b)))/
5. lateral, 1.literal :
K(t(X1,Z1,Y1),d(Z1,Y1),X1)&
LEVEL= 11; resolvent:
/O(Y1,k(X1,ng(b)))/
~K(Y1,s(h,g),V1)~K(Y1,w(b,s(d(i,t1),d(p,t2))),k(Y1,ng(b)))/
6. lateral, 1.literal :
K(t(X1,Z1),d(Z1,Y1),X1)&
LEVEL= 12; resolvent:
By getting marks to domain level we obtain:
(X1 ∧ (i<=n) ⇒ X1i 0 p x) ∧ (X1 ∧ ¬(i<=n) ⇒ V)
Putting X1: p = x ⋅ \prod_{j=0}^{i} (j - 1) we obtain
following correct implications:
\[ p = x \prod_{j=0}^{i} (j - 1) \Rightarrow \]
\[ p \cdot i = x \cdot i \prod_{j=0}^{i} (j - 1) \Rightarrow \]
\[ p \cdot i = x \cdot (i - 1 + 1) \prod_{j=0}^{i} (j - 1) \Rightarrow \]
\[ p \cdot i = x \prod_{j=0}^{i+1} (j - 1) \]

For \( i = 0 \) we obtain:
\[ p = x \prod_{j=0}^{i} (j - 1) \Rightarrow \]
\[ p = x \prod_{j=0}^{0} (j - 1) \Rightarrow \]
\[ p = x \]

By this the compliance is proven, which is enough to conclude that a given program is (partially) correct (until terminating).

5. INTERPRETATION RELATED TO DEMONSTRATION OF PROGRAM CORRECTNESS

Interpret the sequence \( J: B_1, ... , B_n \) as program \( S \). Interpret the elements \( A(\tau) \) as initial elements for the composition of program \( S \), and the elements \( R(\tau) \) as rules for the composition of program constructions.

Vice versa, if we consider program \( S \) as sequence \( J \), initial elementary program operators as elements \( A(\tau) \) and rules for composition of program structures as elements \( R(\tau) \), with this the problem of verification of the correctness of the given program is related to demonstration of correctness of deduction in corresponding formal theory. It is necessary to represent axioms, rules and program with predicate formulas.

With all that is mentioned above we defined the general frame for the composition of concrete proceedings for demonstration of program correctness with the deductive method. With the variety of choices regarding axioms, rules and predicate registration for the different composition proceedings are possible.

6. CONCLUSION

Software testing is the important step in program development. Software producers would like to predict number of errors in software systems before the application, so they could estimate quality of product bought and difficulties in maintenance process [18]. Testing often takes 40% of time needed for development of software package, which is the best proof that it is a very complex process. Aim of testing is to establish whether software is behaving in the way envisaged by specification. Therefore, primary goal of software testing is to find errors. Nevertheless, not all errors are ever found, but there is a secondary goal in testing, that is to enable a person who performs testing (tester) to trust the software system [19]. From these reasons, it is very important to choose such a testing method that will, in given functions of software system, find those fatal errors that bring to highest hazards. In order to realize this, one of tasks given to programmers is to develop software that is easy to test ("software is designed for people, not for machines") [20].

Program testing is often equalized to looking for any errors [20]. There is no point in testing for errors that probably do not exist. It is much more efficient to think thoroughly about kind of errors that are most probable (or most harmful) and then to choose testing methods that will be able to find such errors. Success of a set of test items is equal to successful execution of detailed test program. One of big issues in program testing is the error reproduction (testers find errors and programmers remove bugs) [21]. It is obvious that there must be some coordination between testers and programmers. Error reproduction is the case when it would be the vest to do a problematic test again and to know exactly when and where error occurred. Therefore, there is no ideal test, as well as there is no ideal product.[22] .Software producers would like to anticipate the number of errors in software systems before their application in order to estimate the quality of acquired program and the difficulties in the maintenance. This work gives the summary and describes the process of program testing, the problems that are to be resolved by testers and some solutions for the efficacious elimination of errors[23]. The testing of big and complex programs is in general the complicated process that has to be realized as systematically as possible, in order to provide adequate confidence and to confirm the quality of given application [24]. The deductions in formal theories represent general frame for the development of deductive methods for the verification of program correctness. This frame gives two basic methods (invalidation of added predicate formula and usage of rules of program logic) and their modifications.

The work with formula that is added to the given program implies the presence of added axioms and without them, the invalidation cannot
be realized. The added axioms describe characteristics of domain predicates and operations and represent necessary knowledge that is to be communicated to the deductive system. The existing results described above imply that kind of knowledge, but this appears to be notable difficulty in practice.

ACKNOWLEDGEMENTS

The work presented in the paper was developed within the IT Project “WEB portals for data analysis and consulting,” No. 13013, supported by the government of Republic of Serbia, 2008 – 2010.

7. REFERENCES