stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 125/79 NOVEMBER

R.J.R. BACK

EXCEPTION HANDLING WITH MULTI-EXIT STATEMENTS

Preprint

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O).

1980 Mathematics subject classification: 68B05, 68B10

ACM-Computing Reviews-category: 5.24, 4.2

Exception handling with multi-exit statements*)

by

R.J.R. Back

ABSTRACT

A new language construct, the *multi-exit statement*, is proposed. This provides a clean way of handling exceptional situations in programs, and makes the programs easy to prove correct. The multi-exit statement is intended to support the program construction technique recently proposed by John Reynolds and Martin van Emden which is based on considering programs as state transition diagrams. Proof rules for showing the total correctness of multi-exit statements will be given which provide a new axiomatisation of goto-statements. This axiomatisation is based on the symbolic execution technique. It conforms closely to the intuition of the programmer making manual proofs of the program correctness easy to perform.

KEY WORDS & PHRASES: *exception handling, multiple exits, goto-statements, partial correctness, total correctness, axiomatisation, symbolic execution, program construction.*

---

*) This report will be submitted for publication elsewhere.

1. INTRODUCTION

A large part of the code in programs actually meant to be used is devoted to the detection and handling of exceptional situations, with the aim of making the program more robust. These exceptional situations may result from errors in the input data, special cases of the algorithm requiring different treatment from the normal cases, and things like that. The code concerned with the exceptional situations is typically added late in the development of the program. Partly, this is because the programmer first wants to concentrate on designing the main computation of his program, before he starts to think about the exceptions. Partly the reason is that the need for considering certain exceptional situations only becomes evident as the design proceeds, sometimes only in the testing and maintenance of his program.

The present emphasis on using structured control structures in programs is not very favourable to exception handling. Adding code for a new exceptional situation often requires a restructuring of the control structure of the program, making the program more difficult to understand. The main computation is easily hidden in a web of exception handling computations. One possible solution to this problem is to add to the structured control structures a special mechanism for handling exceptions. Proposals along this line are given in GOODENOUGH [8] , LEVIN [13] and are incorporated in the design of the language ADA [11], just to mention a few.

Another approach is to design the set of control structures in a way which permits a more flexible way of handling exceptions. The simplest way is, of course, to add go to-statements to the set of control structures allowed. This, however, is not a very good idea, as it is known to lead again to programs which are difficult to understand. On the other hand, the basic idea behind using goto's for exception handling, i.e. separating the handling of exceptions from the handling of the normal cases, is sound. What is needed therefore is a more restricted way of using goto's, which allows flexible exception handling but does not lead to programs unduly difficult to understand.

One example of this approach is the construct proposed by ZAHN [17]. This allows one to separate the detection of an exception from the handling of it in the framework of ordinary structured programs, by introducing the concept of an *event* Exits from blocks and loops are made to depend on the occurrence of certain events. Another example is provided by the *tail recursion* construct, proposed by HEHNER [10]. Here flexibility of exception handling is achieved by replacing the iteration construct of structured programs by a less restrictive recursion construct.

In this article we propose a new construct for exception handling, the *multi-exit statement*, which is also based on a restricted use of goto-statements. This has some similarity with Zahns and Hehners constructs, but is based on an

essentially different idea. The multi-exit statement is actually designed to make the construction of programs and the verification of their correctness easier. That it also permits a flexible way of handling exceptions in programs is a pleasant by-product of the design.

The multi-exit statement is intended to support the program construction technique recently proposed by REYNOLDS [17] and by VAN EMDEN [7]. There, programs are viewed as state transition diagrams, and program construction starts by identifying and describing the basic invariants of the program under design. These invariants correspond to the states of the diagram. The states are connected by transitions, which show how one moves from one state to another, by testing the program variables and assigning new values to them. The values of the program variables after a transition must satisfy the invariant corresponding to the target state, whenever the values of the program variables before the transition satisfy the invariant corresponding to the source state.

This approach to program construction has some important consequences. First, it is easy to prove the correctness of the programs constructed, as the invariants needed for the proof already are there, and need not be deduced from the program text. Secondly it becomes easy to separate the exceptional cases from the normal cases in the program. Exceptions are handled by simply adding new states to the diagram, i.e. by giving new invariants that describe the exceptional situations. The code for handling the exceptions will not interfere with the code for handling the normal cases. A third consequence is that there is no need to put restrictions on the flow of control in the program. In fact, restricting oneself to, say, structured control only would do more harm than good, as that might prevent one from finding the simplest possible invariants for the program. In any case, such a restriction will not make proving the program correct any easier, so the basic argument for this restriction does not apply.

The use of multi-exit statements in program construction has been discussed in BACK [2] and will not be further considered here. We will mainly be concerned with the correctness of programs constructed with multi-exit statements. First, we define the syntax of multi-exit statements and explain their meaning informally with an example. A more formal definition of their meaning is provided by a Hoare-like axiomatisation of their *partial correctness*. After this we present another axiomatisation, of the *total correctness* of multi-exit statements, based on the *symbolic execution technique*. This latter axiomatisation is much more natural to use, and gives the programmes a method for checking the correctness of his program in a straightforward way.

2. MULTI-EXIT STATEMENTS

The syntax of *simple multi-exit statements* is as follows.

$S ::= L$ (*label*)

$\mid x_1,\ldots,x_m := e_1,\ldots,e_m; S_1$ (*assignment*)

$\mid$ **if** $b_1 \rightarrow S_1 \,\square \ldots \square\, b_m \rightarrow S_m$ **fi** (*conditional*)

Here $m \geq 1$, $S, S_1,\ldots,S_m$ stand for simple multi-exit statements, L is a label, $x_1,\ldots x_m$ are variables, $e_1,\ldots,e_m$ are expressions and $b_1,\ldots,b_m$ are boolean expressions.

The syntax of simple multi-exit statements is essentially the same as that of the statements given in HEHNER [10]. The interpretation of labels is, however, different. Hehner interprets labels as standing for *actions*, while we interpret them as standing for *invariants*, i.e. assertions about the values of the program variables. When programs are considered as state transition diagrams, then a label identifies a state and a simple multi-exit statement defines a transition from an initial state to the final states identified by the labels in the statement.

As an example, consider the simple multi-exit statement

$x_1 = e_1$; **if** $b_1 \rightarrow x_2 := e_2$; L_1
$\square\ b_2 \rightarrow L_2$
fi

Execution of this statement starts by performing the assignment $x_1 := e_1$. Then the conditional is executed. If b_1 holds, then $x_2 := e_2$ is performed, and execution ends in final state L_1. If b_2 holds, then execution ends in final state L_2. If both b_1 and b_2 hold, then either alternative is choosen (nondeterministically). Finally, if neither b_1 nor b_2 hold, then exectuion is *aborted*, meaning it stops without having reached a final state. Abortion can also occur as a result of trying to evaluate an undefined expression in an assignment statement or in a conditional statement.

The *multi-exit statements* are now defined by adding a fourth production to the syntax definition above:

$S ::= \ldots \mid$ **begin** $D; S_0 \,\square\, L_1(R_1):S_1 \ldots \square\, L_n(R_n):S_n$ **end**. (*block*)

$S, S_0,\ldots,S_n$ now stand for multi-exit statements, $L_1,\ldots,L_n$ are labels and $R_1,\ldots,R_n$ are assertions. D is a list of local variable declarations.

The labels $L_1,\ldots,L_n$ are *internal* (*local*) labels of the block, and are associated with corresponding assertions $R_1,\ldots,R_n$. The statement S_0 is executed on entry to the block. If S_0 ends in some internal lable L_i, $1 \leq i \leq k$, then execution continues with the corresponding statement S_i. If this again ends in an internal label, the statement associated with that label is executed and so on. As soon as the execution reaches an *external* (*global*) label, that is, a label not declared in

the block, the block is exited. The execution then continues in the outer block in which this label is declared, provided it *is* declared in some outer block, otherwise the execution stops.

As is evident from this description, the semantics of multi-exit statements is the ordinary one, which we get by replacing each label L, at the end of a branch in a simple multi-exit statement, by the statement *goto* L. Our programs are really *goto*-programs, although in a slightly disguised form. The syntax given does, however, enforce a certain discipline in the use of goto's. The execution is not allowed to fall through to the next statement (i.e. serial execution is not the default), because each branch in a multi-exit statement must end in a label, signalling an explicit jump to that label. Even more important, each label must be associated with an assertion, describing the situation which always holds when the label is reached.

The block construct attains a number of different, but related goals. It provides a way in which multi-exit statements can be compounded. It also provides a way of achieving iteration. The statement S_0 makes the initial preparations needed for the iteration, and the declaration D provides the local variables needed in the iteration.

As an example of using multi-exit statements, we will show how to program the *binary search algorithm*. Let A,x and h be variables declared in some outer block by

```
var x,h: integer;
    A : array [1..N] of integer;
```

N is some integer constant. We will give a multi-exit statement which searches for the value x in the array A, setting h to indicate the position of x in A. We may assume that $N \geq 1$ and that the values in A are strictly increasing, i.e. that

$$A[i] < A[i+1], \text{ for } i = 1,2,\ldots,N-1.$$

The multi-exit statement has two exits. Either the element x occurs in A, in which case the exit

element found ($A[h] = x$, $1 \leq h \leq N$)

is taken, or x does not occur in A, in which case the exit

element not in the array ($A[h] < x < A[h+1]$, $0 \leq h \leq N$)

is taken. Here we have also given the assertions associated with the exits, the association being defined in some outer block in which these labels would be declared. In the latter assertion, we have used the convention that $A[0] = -\infty$ and

A[N+1] = + ∞. The variable h is used here to indicate the position where x should be. Either one of the two exits may be considered as exceptional, depending on the application at hand. It is also possible to consider both exits as normal, in which case the statement functions as a test with a side-effect (setting h to the location of x in A).

The multi-exit statement performing the binary search is as follows:

```
begin var m,n,i: integer;
     m,n: = 0,N+1; element not found yet
■ element not found yet (A[m] < x < A[n], 0 ≤ m < n ≤ N+1):
     if m+1 = n → h:=m, element not in the array
     □  m+1 < n → i:=(m+n) div 2;
                  if x < A[i] → n:=i; element not found yet
                  □  x = A[i] → h:=i; element found
                  □  x >  [i] → m:=i; element not found yet
                  fi
     fi
end.
```

This program is a very simple one, containing just one internal label. More realistic examples, with several internal labels, are given in BACK[2].

3. PARTIAL CORRECTNESS OF MULTI-EXIT STATEMENTS

Proof rules for goto-statements were first presented by CLINT and HOARE [6]. They used Hoare's axiomatic approach extending it with special proof rules for goto-statements and labels. Other axiomatisations along these lines have been presented by KOWALTOWSKI [12], ARBIB and ALAGIC [1] and DE BRUIN [5]. They all consider the partial correctness of programs with goto-statement while WANG [15] gives an axiomatisation of total correctness of programs with goto-statements, based on the *intermittent assertion method*.

The multi-exit statement can be very simply axiomatised in a Hoare-like system, by changing the correctness formulas. Instead of using the Hoare notation P{S}Q, we use correctness formulae of the form

$$E \vdash P:S.$$

Here E is a list $L_1(R_1),\ldots,L_k(R_k)$ of labels with associated assertions, which is referred to as the *environment*. P is a *precondition* and S is a multi-exit statement. The correctness formula states that if P holds initially for the program variables, and if execution of S terminates in label L in E, then the assertion

R_i, associated with L_i in E, must hold for the final values of the program variables. Thus the formula expresses *partial correctness* of S with respect to the precondition R and the environment E.

The following proof rules are sufficient for establishing the partial correctness of multi-exit statements (no axioms are needed).

1. *Consequence*

$$\frac{E \vdash R:S,\ R' \Rightarrow R}{E \vdash R':S}$$

2. *Label*

$$\frac{R \Rightarrow E(L)}{E \vdash R:L}$$

3. *Assignment*

$$\frac{E \vdash R:S}{E \vdash R[e/x]:\ x:=e\ S}$$

4. *Conditional*

$$\frac{E \vdash R \wedge b_i : S_i,\ \text{for } i=1,\ldots,m}{E \vdash R:\ \underline{\text{if}}\ b_1 \rightarrow S_1 \,\square\, \ldots \,\square\, b_m \rightarrow S_m\ \underline{\text{fi}}}$$

5. *Block*

$$\frac{E, L_1(R_1), \ldots, L_n(R_n) \vdash R_i : S_i,\ \text{for } i=0,1,\ldots,n}{E \vdash R_0:\ \underline{\text{begin}}\ S_0\ \blacksquare\ L_1(R_1):S_1 \ldots \blacksquare\ L_n(R_n):S_n\ \underline{\text{end}}}$$

Here E(L) denotes the assertion associated with L in E and R[e/x] denotes the formula we get by substituting e for all free occurrences of x in R. For simplicity we have only considered single variable assignments and blocks without local variable declarations. We also assume for simplicity, that redeclaration of labels in inner blocks is not allowed.

This axiomatisation is considerably simpler than those of the references mentioned above, partly because of the introduction of the environment and partly because we do not have to consider the possibility of exiting through the end of a statement, i.e. all exits in multi-exit statements are by explicit jumps to labels in the environment (de Bruin uses environments in a similar way).

This axiomatisation, however, is not very useful in practice. It forces one to construct the verification conditions by backward substitution, which is not very natural, and it only formalises partial correctness of programs, whereas in practice one is interested in total correctness. We therefore proceed to a more useful axiomatisation, in which total correctness of multi-exit statements is formalised.

4. TOTAL CORRECTNESS OF MULTI-EXIT STATEMENTS

The proof rules for total correctness of multi-exit statements are based on the *symbolic execution technique* (see e.g. HANTLER & KING [9]). The correctness formulae will be of the form

$$E \mid\!\!\vdash R \wedge x = f: S,$$

where E is an environment as before, R is an assertion, x is a list of program variables, f is a list of terms and S is a multi-exit statement. We require that no program variables occur free in R and also that no program variables occur in any term in f. The equality x = f stands for $x_1 = f_1 \wedge x_2 = f_1 \wedge \ldots \wedge x_n = f_n$, where $x = x_1,\ldots,x_n$ and $f = f_1,\ldots,f_n$.

The correctness formula $E \mid\!\!\vdash R \wedge x = f: S$ states that if $R \wedge x = f$ holds initially, then execution of S terminates in an external label L of S, and E(L) will then hold for the final values of the program variables. Thus the correctness formula expresses *total correctness* of the multi-exit statement S with respect to the precondition $R \wedge x = f$ and the environment E (i.e. neither nontermination nor abnormal termination of the execution is allowed).

The proof rules are as follows:

1. *Label*

$$\frac{R \wedge x = f \Rightarrow E(L)}{E \mid\!\!\vdash R \wedge x = f: L}$$

2. *Assignment*

$$\frac{R \wedge x = f \Rightarrow \mathrm{def}[e] \qquad E \mid\!\!\vdash R \wedge x = f': S}{E \mid\!\!\vdash R \wedge x = f: x_i := e;S}$$

Here $f' = f_1,\ldots,f_{i-1}, e[f/x], f_{i+1},\ldots,f_n$, where e[f/x] denotes the result of substituting f_i for each free occurrence of x_i in e, i = 1,...,n. In the first assumption, def[e] is some condition on the program variables which guarantees that e is well-defined.

3. *Conditional*

$$\frac{R \wedge x = f \Rightarrow \mathrm{def}[b_i], \text{ for } i = 1,\ldots,m \qquad R \wedge x = f \Rightarrow b_1 \vee \ldots \vee b_m \qquad E \mid\!\!\vdash R \wedge b_i[f/x] \wedge x = f: S_i, \text{ for } i = 1,\ldots,m}{E \mid\!\!\vdash R \wedge x = f: \underline{\text{if}}\ b_1 \rightarrow S_1 \square \ldots \square\ b_m \rightarrow S_m\ \underline{\text{fi}}}$$

Here $\mathrm{def}[b_i]$ serves the same purpose as def[e] above.

4. *Block*

$$\frac{R_i[z'/z] \wedge z = z' \Rightarrow t \geq 0, \text{ for } i = 1,\ldots,n \qquad E,E' \mid\!\!\vdash R \wedge x = f \wedge y = y': S_o \qquad E,E'' \mid\!\!\vdash R_i[z'/z] \wedge z = z': S_i, \text{ for } i = 1,\ldots,n}{E \mid\!\!\vdash R \wedge x = f: \underline{\text{begin}}\ D;\ S_o \blacksquare L_1(R_1):S_1 \ldots \blacksquare L_n(R_n):S_n\ \underline{\text{end}}}$$

Here $E' = L_1(R_1),\ldots,L_n(R_n)$ and

$E'' = L_1(R_1 \wedge t < t\,[z'/z]),\ldots,L_n(R_n \wedge t < t\,[z'/z])$,

z is the list x,y (i.e. the variables in x followed by the variables in y), where y is the list of new variables declared in D, t is a term describing the termination function, i.e. an integer function on the program variables in z, and z' and y' are lists of fresh variables, not used elsewhere in the assumptions. For simplicity we assume in this proof rule that redeclaration of variables and labels is not allowed.

Abnormal termination is ruled out by the rules for assignment and conditional. The first assumption of the assignment rule requires that the expression e in the assignment statement statement is well-defined when evaluated. The first assumption of the conditional rule again requires all the guards to be well-defined when evaluated, while the second assumption requires some guard to be true when the conditional is to executed. Nontermination is ruled out by the block rule. The termination function is required to have a non-negative value in each internal state, by the first assumption. By the third assumption, each internal transition must decrease the value of t. This guarantees termination of the block in the usual way.

The proof rule for blocks given here is unnecessarily strict with respect to termination. It requires that the function t is decreased by every internal transition of the block. Actually, it is sufficient to assume that execution cannot return to an internal label from which it has started, without decreasing the value of t. Blocks that satisfy this weaker requirement can be handled by the following rule:

5. *Unfolding*

$$\frac{E \mid\!\!- \; R \wedge x = f: \underline{\text{begin}}\ D;\ S_0 \ldots \blacksquare\ L_i(R_i): S_i[S_k/L_k] \ldots \underline{\text{end}}}{E \mid\!\!- \; R \wedge x = f: \underline{\text{begin}}\ D;\ S_0 \ldots \blacksquare\ L_i(R_i): S_i \ldots \underline{\text{end}}}$$

Here $1 \le i,k \le n$, and $S_i[S_k/L_k]$ denotes the result of substituting the multi-exit statement S_k for some label L_k in S_i. This is the same as *unfolding* the iteration one step. Any block which can be shown (informally) to terminate according to the weaker requirement, can be transformed with a finite number of unfoldings to an equivalent block which can be shown to terminate according to the requirements of proof rule 4.

5. CHECKING THE CORRECTNESS OF MULTI-EXIT STATEMENTS

The proof rules given in the preceeding chapter enable the programmer to check the correctness of his program in a straightforward way. As the program already contains all the invariants needed for the proof, establishing the

correctness of the program does not require any real ingenuity on the part of the programmer.

We illustrate this by considering the binary search algorithm presented above. We wish to prove that

$$E \;||\!-\; \text{true} \wedge\; h = h' \colon S$$

holds, where S is the binary search program, and E is the environment

element found ($A[h] = x$, $1 \leq h \leq N$),
element not in the array ($A[h] < x < A[h+1]$, $0 \leq h \leq N$).

For simplicity, we treat N,A and x all as constants, satisfying the properties $N \geq 1$ and $A[i] < A[i+1]$, $i = 1,\ldots, N-1$. We choose $t = n-m$ as our termination function.

With the proof rules we construct a checklist for the program, stating all the facts which need to be proved, together with the assumptions which may be used in proving them. The checklist for the binary search algorithm looks as follows, where the proof rules used are indicated in parenthesis (e.g. 3.2 stands for the second assumption of proof rule no. 3).

```
{assume true (4.2)}
begin var m,n,i: integer;
      m,n:= 0,N+1;
      {prove that 0 and N+1 are well-defined expressions (2.1)}
      element not found yet
      {prove that A[m] < x < A[m] ∧  0 ≤ m < n ≤ N+1, when m=0,n=N+1 (1.1)}

■ element not found yet (A[m] < x < A[n], 0 ≤ m < n ≤ N+1):
      {assume that A[m'] < x < A[n'] ∧ 0 ≤ m' < n' ≤ N+1 (4.3)}
      {prove that n'-m' ≥ 0 (4.1)}
      if {prove that m+1 = n and m+1 < n are well-defined, when m=m',n=n'(3.1)}
         {prove that m+1 = n ∨ m+1 < n, when m = m', n= n' (3.2)}
         m+1 = n → {assume that m'+1 = n' (3.3)}
                   h:= m;
                   {prove that m is well-defined, when m = m' (2.1)}
                   element not in the array
                   {prove that A[h]<x<[h+1] ∧ 0 ≤h≤ N, when h = m' (1.1)}
      □  m+1 < n → {assume that m' + 1 < n' (3.3)}
                   i:= (m+n) div 2;
```

```
                {prove that (m+n)div 2 is well defined, when m=m',n=n'(2.1)}
                if {prove that x<A[i],x=A[i],x>A[i] are well-defined,
                        when i = (m'+n') div 2 (3.1)}
                   {prove that x<A[i] ∨ x = A[i] ∨ x > A[i], when
                        i = (m'+n') div 2 (3.2)}
                   x < A[i] → {assume that x < A[(m'+n')div 2] (3.3)}
                              n:= i;
                              {prove that i is well-defined, when
                                 i = (m'+n')div 2, (2.1)}
                              element not found yet
                              {prove that A[m] < x < A[n] ∧ 0 ≤ m < n ≤ N+1
                              ∧ n-m < n'-m', when m=m', n=(m'+n')div 2 (1.1)}
                   :
                   fi
        fi
end
```

6. PRACTICAL EXPERIENCES OF TESTING MULTI-EXIT STATEMENTS

The multi-exit statements have been used in two programming projects at the Computing Centre of the University of Helsinki. The experiences of these projects indicate that concentrating on the program invariants does make the program construction task easier. The program invariants are not too difficult to find, once one knows what one is looking for. The flexibility in handling exceptions also contributes significantly to the ease of constructing programs, by allowing the programmer to work at the different cases independently. The possible danger of using the multi-exit statement lies in not being precise enough in describing the program invariants. Sloppy description of invariants leads to all the well-known problems of the undisciplined use of goto-statements. Luckily, there is a quite effective cure: the programmer should be asked to hand check the correctness of his program, in the manner shown in the preceeding section. This will immediately reveal most of the errors and omissions in the invariants.

The multi-exit statement has not yet been properly implemented. However, it is relatively easy to give a preprocessor, which translates the multi-exit statements into, say, Algol-code. Such a preprocessor was used in one of the programming projects mentioned above, while in the other project the multi-exit statements were hand translated into FORTRAN code. In a forthcoming report, BACK and KOSKENNIEMI [3], a modification of the language Modula-2 by WIRTH [16] is proposed, which incorporates the multi-exit statement. Experience in using multi-exit statements on a larger programming project will be reported in BACK and KOSKENNIEMI [4].

This joint work with Koskenniemi is essentially concerned with showing how to use multi-exit statements in programs with procedures and uses defined data structures, in a way which makes the programs easy to understand and to prove correct and also allows possible exception handling.

REFERENCES

1. ARBIB, M.A. & ALAGIC, S., *Proof rules for goto's,* Acta Informatica 11, 139-148, 1979.
2. BACK, R.J.R, *Program construction by situation analysis,* Computing Centre of University of Helsinki, Research report 6, 1978.
3. BACK, R.J.R, & KOSKENNIEMI,K, *Constructing verifiable programs: a language proposal,* in preparation.
4. BACK, R.J.R, & KOSKENNIEMI, K., *Constructing verifiable programs: a case study,* in preparation..
5. de BRUIN, A., *Goto statements: semantics and deduction systems* (preprint). Report IW 74/79, Mathematisch Centrum, 1979.
6. CLINT, M. & HOARE, C.A.R., *Program proving: jumps and functions.* Acta Informatica 1, 214-224, 1972.
7. van EMDEN, M.H., *Programming with verification conditions,* IEEE Transactions on Software Engineering, SE-5,2, 1979.
8. GOODENOUGH, J.B., *Exception handling: issues and a proposed notation.* Comm. of ACM, 18,12,683-696, 1975.
9. HANTLER, S.L. & KING, J.C., *An introduction to proving the correctness of programs,* Computing Surveys 8, 3, 331-353, 1976.
10. HEHNER, E., *Do considered od: a contribution to the programming calculus,* Acta Information 11, 287-304, 1979.
11. ICHBIAH, J.D & al, *Rationale for the design of the ADA programming language,* Sigplan Notices 14, 6, 1979.
12. KOWALTOWSKI, T., *Axiomatic approach to side effects and general jumps,* Acta Informatica 7, 357-360, 1977.
13. LEVIN, R., *Program structures for exceptional condition handling,* Dept. of Computer Science, Carnegie-Mellon University, 1977.
14. REYNOLDS, J.C, *Programming with transition diagrams,* In Gries, D. (ed.) Programming Methodology, Springer Verlag, Berlin, 1978.
15. WANG, A., *An axiomatic basis for proving total correctness of goto-programs,* BIT 16, 88-102, 1976.
16. WIRTH, N., *Modula-2,* Institut fur Informatik, ETH, Zurich, 1979.
17. ZAHN, C.T., *A control structure for natural top-down structured programming,* Symposium on Programming Languages, Paris 1974.