TECHNISCHE HOGESCHOOL TWENTE

MEMORANDUM NR. 178

EXCHANGING ROBUSTNESS OF A PROGRAM FOR A RELAXATION OF ITS SPECIFICATION

C. BRON
M.M. FOKKINGA

SEPTEMBER 1977

Department of Applied Mathematics,
Twente University of Technology,
P.O. Box 217, Enschede, The Netherlands.

ONDERAFDELING DER TOEGEPASTE WISKUNDE

Contents

Abstract

This paper motivates a programming concept which realizes the exchange of some robustness of a correct program part for a relaxation of its specification. The technical tool proposed restricts program abortions to a specified part of the program only, so that the execution of that part can - from the outside - not be distinguished from normal termination. The abstract (= axiomatic) semantics closely reflect the intuitive appeal of the construct. The tool is powerful in the sense that it provides a means to restrict abortions to program parts which may be unknown at the site where the abortion has been "programmed".

0 Introduction

There is a general understanding that in program composition the programmer's restriction to control structures which have simple semantic properties is of beneficial value to his product, the program. Thus one consideres the goto harmful, Dijkstra (68). Yet there is clearly a need for abnormal flow of control, whenever in the course of a computation conditions arise due to the failure of the input to meet the assumptions on which the program is based.

In this paper we express our views on the intrinsic needs for some form of abnormal flow of control and, carefully avoiding to introduce just another control structure, we propose a programming concept with strikingly simple Hoare(69)-like axiomatic semantics. In addition to what has been said in the abstract, one of the attractive features is the fact that programs written on rather strong input assumptions may be adapted to a relaxation of these assumptions by slight additions to to program-text which do not have any impact on the original textual structure.

Our proposal constitutes a refinement of the most restrictive form of abnormal flow of control: program abortion as implicitly provided by the guarded commands of Dijkstra (75). It is far more powerfull than restricted forms of goto, e.g. Wirth (71), in the sense that the user can react to system detected error occurrences. It is not primarily intended

as a proposal for untimely termination of repetitive constructs, like Zahn (74), Knuth (75) and Adams (77), although it may be (mis-)used for that purpose.

The paper is organized as follows. In the section 1 we recall the notion of correctness and briefly treat the important notion of robustness and the programming methodology leading to robust programs. Section 2 describes the abstract needs of the programming concept; section 3 gives the operational realization and section 4 the abstract semantics. Section 5 and 6 mainly adress to the audience interested in formal verification methods and may be skipped; we do not introduce any formalism so that interested non-experts may appreciate the main ideas as well. For completeness sake some remarks about parametrization are made in section 7. Section 8 reviews the proposal and briefly compares it with other proposals found in the literature. Throughout the paper we will give examples that refer back to the program of example 1.

(Note 0. The defining occurrence of a new notion is written in the type font <defining occurrence>. Sometimes the definition is only implicit in the context. *End of note 0).*

1 Correctness and Robustness of Programs

Recall the concept of total correctness:

> A program part is <correct>, sometimes called totally correct, w.r.t. <precondition> R and <post-condition> S, if its execution in a state satisfying R leads to termination in a state satisfying S.

Correctness should not be confused with partial correctness:

> A program part is <partially correct> w.r.t. R and S if its execution in a state satisfying R either does not lead to termination or does lead to termination in a state satisfying S.*)

We assume the reader familiar with a methodlogy for the construction of correct programs, preferably using the alternative and repetitive constructs of Dijkstra (75, 76):

> if..fi and do..od.

An excellent treatment can be found in Dijkstra (76).

In this paper it is the alternative construct which plays a fundamental role, the reason being its potential cause of abortion. For completeness sake, we recall the semantics.

Consider

> if B1 → stmnt1 □..□ B*n* → stmnt*n* fi.

IF (a) stmnt*i* is correct w.r.t. R ∧ B*i* and S, for all *i*:1..*n*

(b) R → B1 ∨..∨ B*n*

THEN the whole construct is correct w.r.t. R and S.

Operationally: upon execution of the alternative construct it is required that at least one of the guards B*i* is true, otherwise program execution will be aborted; one of the statements stmnt*i* whose guard B*i* is true, is nondeterministically selected and executed.

(Example 1. Cluster Counting Program CCP.

The program given below will be frequently referred to in the sequel. We first define some notions; they apply to strings of parentheses (opening par'('and closing par')') only. In extended BNF, where *

---

*) A partially correct program may have false as its postcondition; this implies that the program will not terminate.

denotes repetition of zero or more times, we define

<cluster>::=(<cluster>*)

<extensible>::=<cluster>* | <cluster>*(<extensible>

So a nonempty (sub)string S is a cluster iff for each initial proper substring of S the number of opening par's is greater than the number of closing par's, whereas these numbers are equal for the whole string S.Further, a possibly empty string is extensible iff addition of zero or more closing pars to the right yields a sequence of clusters. By <the clusters in an extensible string> we mean, as the definition of extensibility suggests, those clusters in the string before the first non-matched opening par which are not contained in another cluster.

Now let *A*:array[1..100] of *character* be a non-assignable array of characters and *m*:*integer* a non-assignable integer and suppose a program is required which on precondition {$0 \le m \le 100 \wedge A[1..m]$ contains parentheses only and is extensible} has to establish postcondition {number of clusters in $A[1..m]$ has been printed}. (What is initially true about A and m remains so, because they are constants. Throughout the paper we mention such invariantly true assertions only explicitly in the precondition.) The following program fulfils the specification:

```
prelude var i, open, close, clus:integer
begin   i, open, close, clus:=0,0,0,0;
        do i≠m → i:=i+1;
                 if A[i]='(' → open:=open+1
                 □  A[i]=')' → close:=close+1;
                               if open=close → clus := clus+1
                               □  open>close → skip
                               fi
                 fi
        od;
        print(clus)
end.
```

The correctness may be verfied using invariant relation {$0 \le i \le m \wedge A[1..i]$ contains *clus* clusters and needs *(open-close)* closing parentheses for extension} and variant function $(m-i)$. *End of example 1).*

Less known than the notion of correctness is the notion of robustness, introduced by Dijkstra (76) only informally:

> A program part is <robust in> a <robustness condition> R if its execution in a state satisfying R leads to abortion.

Note that robustness in R implies robustness in R' if R' → R, and robustness in both R and R' implies robustness in R ∨ R'.
Further, robustness of a procedure carries over to each call.

When - due to "external accidents", - a program is executed in a state not satisfying the precondition, abortion is to be preferred, as a kind of alarm, over termination in a state not satisfying the expected postcondition. Note that in general one can hardly require the precondition to be checked before execution of the program, because more often than not this would require a sort of pre-execution of at least some part of the program. Thus preferably a program is robust in that part*) of the negation of its precondition which leads to "wrong" termination.
(Example 2. CCP is robust in {$0 \le m \le 100 \wedge A[1..m]$ either contains not only parentheses or is not extensible}. So "erroneous" input, provided $0 \le m \le 100$, does not give erroneous output: abortion will take place in stead.
Example 3. Assuming that the array selection $A[i]$ is robust in {not$(1 \le i \le 100)$}, CCP is also robust in {not$(0 \le m \le 100)$}.
End of examples 2 and 3).

The implication on conditions, or equivalently inclusion on sets, induces a partial order on robustness: the weaker (i.e. less restrictive) the robustness condition, the more robust the program is, as more initial states lead to abortion. Robust programs are obtained by choosing the guards of an alternative construct as strong as possible. Indeed, one may weaken the robustness condition $(B1 \vee .. \vee Bn)$ of an alternative construct if B1 → stmnt1 □..□ Bn → stmntn fi by strenthening the guards Bi whereas - assuming the validity of the precondition - they yet remain equivalent. Fortunately this way of program construction is quite harmoneous with the overall goal of structured programming in the sense that as much as possible is clear by local inspection of the text: a

---

*) This part is the negation of the precondition iff the precondition happens to be the weakest one w.r.t. the given postcondition.

stronger guard B shows more than a weaker one B', which happens to be equivalent to B only on account of the context.
(Example 4. Replacing $A[i]$ = '(' by $A[i] \neq$ ')' diminishes both the robustness and the clarity of CCP without disturbing the correctness and correctness proof. *End of example 4).*

Thus alternative constructs may check the validity of essential parts of the precondition and cause abortion if invalidity is detected. For future use we allow alternative constructs to be optionally suffixed with an identifier, the <robustness identification>, to denote the associated robustness condition*). Such use of a robustness identification is termed a <rob-identification>. Distinct constructs need not be distinctly rob-identified if both check the same condition. Constructs not explicitly rob-identified are assumed to be rob-identified "by default" by the standard robustness identification *abort*, short for "a bit of robustness transmitted".
(Example 5. In CCP both fi's could be identified by *incorrect-input;* more refined identifications might read *non-par* for the outer fi and *non-ext* for the inner one. This latter program will be referred to as CCP'. *End of example 5).*

The introduction of rob-identifications suggest a refined notion of robustness. We redefine

> A program part is <robust in> (id, R) if its execution in a state satisfying the <robustness condition> R leads to abortion due to the execution of an alternative construct rob-identified by the robustness identification id.

So, for not rob-identified program parts, robustness in R is robustness in (*abort*, R).
(Example 6. CCP' is robust in (*nonpar*, {$0 \leq m \leq 100 \wedge A[1..m]$ does not contain parentheses only but its initial part up to the first non-parenthesis is extensible}). CCP' is also robust in (*nonext*, {$0 \leq m \leq 100 \wedge$ an initial part of $A[1..m]$, containing parentheses only,

---

*) At this point one may imagine that the robustness identification is part of the errormessage which will be given when program abortion occurs.

is nonextensible}). *End of example 6).*

(Note 1. Due to the nondeterminism it may be impossible to assert any particular robustness for a correct program, whereas without rob-identifications it is non-trivially robust. For example, the following program is correct w.r.t. pre x = 0 and post x = 0 and - without identifications - robust in (*abort*, x ≠ 0).
It is however neither robust in *wrongx1* nor in *wrongx2*:

```
if true → if x ≠ 0 → skip fi wrongx1
□  true → if x ≠ 0 → skip fi wrongx2
fi
```

Thus the formalism for robustness forces alternative constructs, which are nondeterministically treated alike, to be rob-identified alike as well. *End of note 1).*

(Note 2. The kind of alarm provided for by the do-od construct is nontermination: assuming that $A[i]$ yields an opening parenthesis for all i, an initial state satisfying $m < 0$ leads to nontermination! Here, the methodology is to choose the guards as weak as possible: in CCP the guard reads $i \neq m$, which is weaker than $i < m$. Indeed, $i < m$ would not lead to nontermination. Hehner (76) shows that the use of "recursive refinement" in stead of repetition delivers - o.a. - robustness in stead of non-termination. *End of note 2).*

This completes our treatment of the notion of, and methodology leading to robust programs.

2 Exchanging Robustness for Relaxation of the Program Specification

In this section we decribe the need and motivation and some requirements for a programming concept. A realization will be proposed in the next section.

Once a satisfactory and correct program has been written, it seems yet quite natural to make another step in the development of the program: one may want to exchange some robustness for a relaxation of the correctness specification. For instance, in stead of some robustness one may prefer an error message to be printed, followed by termination of the program. Thus the original program specification, say (R,S), has been relaxed into (R ∨ R', S ∨ S') where R' is a robustness condition of the original program and S' asserts that the error message has been printed.
(Example 7. We may wish to relax the program specification of CCP into precondition {$0 \leq m \leq 100$} and postcondition {(*A*[1..*m*] both contains parentheses only and is extensible and its number of clusters has been printed) or (otherwise, "*A*[1..*m*] doesnot contain parentheses only or is not extensible" has been printed)}. *End of example 7).*

This holds even for *parts of a program:* one may prefer, if a program part happens to be invoked when a robustness condition prevails, in stead of robustness to be induced to the whole program, the part to be terminated under establishment of a suitable, relaxed, postcondition. One may prefer so if the context is rather independent of the postcondition established by that program part. The whole program is made more useful in the sense that more initial states lead to -suitable - termination.
(Example 8. The relaxation of example 7 is equally well desirable if CCP is used by a larger program which merely processes a series of array's *A* and integers *m*.
Example 9. If the context of CCP does not allow the condition {*A*[1..*m*] does not contain parentheses only} to be part of a relaxed postcondition, we still may desire the relaxed precondition {$0 \leq m \leq 100 \wedge$ *A*[1..*m*] contains parentheses only} and postcondition {(*A*[1..*m*] is extensible and its number of clusters has been printed) or (otherwise, both "*A*[1..*m*] non-extensible" and the greatest i, for which *A*[1..i] is extensible, have been printed)}. *End of examples 8 and 9).*

Note that one must be free to choose the program part so large as to have a weak enough postcondition to allow for a relaxation without disturbing the correctness proof in which it has to fit.
(Example 10. Addition of "$A[i] \neq$ '('or')' → skip" to the outer alternative construct in CCP does exchange robustness of that very construct for a relaxation of its correctness specification, but the relaxed postcondition does not fit in the context. (The addition modifies rather than relaxes the program (specification)). *End of example 10).*

We consider it essential that "system generated" robustness be treated on equal footing with "programmed" one. On the one hand, each system generated action during the execution of a program can be considered as a procedure ultimately invoked from within the program. Fortunately, robustness of the body of a procedure carries over to each call. So the exchange of robustness of a program part indeed affects as well the robustness induced by the system. Typical robustness identifications we think of are: *overflow, index-out-of-bounds, attempt-to-read-beyond-end-of-file, singularity* (in a library procedure for matrix inversion). On the other hand, the system part invoking the user program may be considered to exchange all robustness for an extreme relaxation of the program specification: the relaxed precondition is true and the relaxed postcondition merely asserts that the output and/or an appropriate error message (including a dump of the stack) have been sent to the printing device. Indeed, for the remainder of the system program - which takes care of finalizations like closing files and deallocating resources and so on - it is irrelevant what condition has been established by the user program. There seems, moreover, no distinction reasonable for the system in the way it should exchange the robustness in the various identifications invented by the programmer. Hence, user identified robustness may become evident to the system as robustness in, say, the single standard identification *abort*.
(Example 11. For CCP as incremented in example 9 it may be left to the calling environment how and of what program part - if any and of which system part otherwise - to exchange the robustness in *non-par* (resp. *abort)* and *index-out-of-bounds*. *End of example 11).*

Preferably, of course, the incremental step (of exchanging robustness for a relaxation) should in no way influence the original structure of the program, so that it brings about additions to rather than alterations of the original program text, the original documentation and original correctness proof.

3 A Realization: Restricting Abortions

We give a realization of the programming concept described in section 2.

Consider a program part whose robustness, in (id,R') say, has to be exchanged for the relaxation of the correctness specification (R,S) into (R ∨ R', S ∨ S') for a suitable S'. Obviously:
(1) the exchange of robustness must be realized by restricting the abortions, caused from within the execution of the program part and "rob-identified" by id, to just that very program part, so that - as seen from the outside - it is just terminated, and
(2) the establishment of S' must be realized by the execution of some <terminating statements>, term say.
Syntactically we propose:
(3) without loss of generality the program part to be a block and
(4) to indicate the <restriction of abortions> by the following *addition* to the prelude of the block:

rab id by term bar

to be termed a <rab-definition>.
Further we define:
(5) in order to make valuable errormessages and terminating actions possible (cfr. the i to be printed in example 9 and the dump of the stack in section 3) the evaluation of term to take place in the environment (= collection of *declared* objects) valid for the statement part of the block and
(6) the abortions caused from within the execution of term (and not restricted by the interior of term itself) to be restricted - if at all - by the restrictions of abortions valid when the block is to be executed. (This decision eliminates recursion but has been motivated by the wish to keep the abstract semantics simple, see section 4 and *note 4* in section 5).
We complete the realization proposal by:
(7) requiring the robustness identifications to be declared, say in the format: rid id, so that the scope of this <rid-declaration> contains all the rab-definitions and rob-identifications of id, and
(8) letting the declaration also mean the implicit definition rab id by if false → skip fi bar, thus causing abortions rob-identified by

the standard default identification *abort* for any abortion identified by id which will not be restricted by the programmer.
(Example 12. Examples 11 and 9 may be realized by an addition to the prelude of CCP', so that it reads:

prelude var *i, open, close, clus: integer;* rid *non-ext;*
rab *non-ext* by *print* ("*A*[1..*m*] non-extensible", *i*-1) bar.

If rid *non-par* had been added to the prelude too, and no rab-definition, then the context could not restrict those additions: they would cause program abortion (identified by *abort). End of example 12).*

Note that abortions caused from within a procedure body (or standard operation) will be restricted by the rab-definitions surrounding the site of call rather than the site of declaration.

4 Abstract Semantics of Restriction of Abortions

More important than (or as important as) the precise operational definition are the schemes of reasoning involved in the incremental step of the program development. They need to specify the correctness and robustness of an incremented block in terms of the correctness and robustness of the original block and the terminating statements.

Evidently, however the robustness assertions as defined so far are insufficient, due to the inability to express in what state the abortion will be caused and consequently in what state the terminating statements will be executed. So we refine once more the notion of robustness:

> A program part is <robust in> (id, R, A) if its execution in an initial state satisfying the robustness condition R leads to abortion due to the execution of an alternative construct, rob-identified by id, in a state satisfying the <abortion condition> A.
>
> The condition A should be interpreted in the environment valid for the statement part of the block, if the program part happens to be a block. (Or equivalently, by convention the robustness for the statement part of a block is said to hold for the block).

Note that abortion condition true always suffices; in general robustness in (id, R, A) implies robustness in (id, R', A') if R' → R and A → A'.
(Example 13. An abortion condition for the robustness in *non-ext* as asserted in example 6, is {$A[1..i\text{-}1]$ is extensible but $A[1..i]$is not}.
End of example 13).

For the well-known programming concepts there exist intuitively appealing formal methods to verify robustness assertions, similar in nature to formal methods for the verification of correctness. In section 6 we will describe such methods. For the time being we assume that satisfactory methods exist.

Now we turn to the semantics of restricting abortions. Consider a block to be incremented by the addition of rab id by term bar to its prelude.

The rule for correctness has already been suggested in the motivation of the concept:

1. IF (a) the block has been proved correct w.r.t. R1 and S1 and
   (b) the block is robust in (id, R2, A) and
   (c) term is correct w.r.t. A and S2,
   THEN the incremented block is correct w.r.t. R1 ∨ R2 and S1 ∨ S2.

Second, for the robustness identification id we have

2. IF (a) the block is robust in (id, R, A') and
   (b) term is robust in (id, A', A),
   THEN the incremented block is robust in (id, R, A).

Third, for any identification id' distinct from id,

3. IF (a) block is robust in (id', R1, B) and
   (b) block is robust in (id, R2, A) and
   (c) term is robust in (id', A, B),
   THEN the incremented block is robust in (id', R1 ∨ R2, B).

(Example 14. Using the robustness assertion as completed in example 13, it is almost trivial to prove that the incremental step of example 12 indeed yields a program satisfying the relaxed specification of example 11. The robustness assertions for *non-par* and *index-out-of-bounds* will not be affected by the incrementation. *End of example 14).*

Now we motivate clause (6) of the operational definition. Consider a block, to be incremented by the addition of rab id by term bar to its prelude. Two alternatives to (6), which for appropriate reasons might be judged simpler, suggest themselves: the abortions caused from within term are restricted - if at all - either

(6') by the rab's valid when the statement part of the block is to be executed (so that, by extending the notion of <environment> to contain as well the valid rab's, clause (5) alone would suffice), or

(6") by the rab's valid when the abortion, identified by id, is caused from within the statement part (so that both for procedures and for rab's, the rab-definitions surrounding the site of invocation restrict the abortions caused from within the bodies and terminating statements).

Rather unconsciously clause (6") has been chosen in our first design, Bron et al (76). The abstract semantics however are far more complicated and close inspection revealed a serious flaw, treated in fuller extent in Fokkinga (77) and essentially due to the possibility that restricting abortions to some block B would possibly restrict the abortions to an inner block of B; this is clearly not what we intuitively wanted!

In contrast to (6), clause (6') prescribes the rab's to be evaluated (mutually) recursively. This does not only give rise to additional constraints about a variant function in order to guarantee boundedness of recursion - just as in the case of (mutual) recursive procedures - but also to a relaxation, in the first and third rule, of the abortion condition A of the block into A ∨ A', with the additional premise that term is robust in (id, A', A ∨ A');the second rule however would be simplified: the incremented block would be no longer robust in id. We consider these modifications too inattractive, but the decision might be left open to discussion.

5 Using rab's as a normal control structure

In stead of writing robust programs which in an incremental step are extended by the addition of rab-definitions, one might wish to employ the restrictions of abortions while constructing the program. (An explicit abort-statement is available: if fi always aborts program execution because there does not exist a true guard). We give the abstract semantics for this approach.

First of all, it should be noted that the specification for which an inner program part has to be designed, is already the relaxed one and that the programmer is allowed to program abortions, assuming that these will be restricted by the context (at the site of execution). So apparently program parts are intended to be liberally correct rather than correct:

> A program part is <liberally correct> w.r.t. R and S if its execution in a state satisfying R either leads to abortion or leads to termination in a state satisfying S.

The notion of liberal correctness should not be confused with either correctness or partial correctness.

(Example 15. Because CCP is robust in the negation of its original precondition, it is liberally correct w.r.t. true and the original postcondition. *End of example 15).*

Rather than to give a specific formalism for proving liberal correctness, we give the way any existing formalism for proving correctness should be modified for the present purpose. The modification for the *formalism* is described below; the *interpretation* of formulae has to be modified so that what originally did express correctness, does express liberal correctness now.

First, we adapt the *language* of the proof system. On the one hand, just like the assumed existence of formulae expressing <proc/fun-specifications>, one should introduce formulae expressing <rab-specifications>: these formulae merely give an <abortion condition> on which it is allowed to invoke an abortion for the robustness identification under consideration. On the other hand, we need to extend the proc/fun-specifications so that they also may express some

<assumed rab-specifications>: in this way abortions invoked from within a proc/fun-body will be modelled as implicit parameters, see below.

Second, we adapt the *rules* of the proof system. Obviously, we need one new rule:
Rab-definition (a) A rab-specification (is an abstraction from the definition and) is verified by showing that the terminating statements, with the abortion condition of the specification as precondition, establish the block's postcondition. Within this verification the rab-specifications allowed to be used are those (assumed to be) valid for the block. (b) A verified rab-specification is valid in the normal, *textual*, scope of the definition (; redefinitions for the same identification cause a hole in the scope).
Three rules of the original proof system need be modified:
Proc/fun-declaration. A specification is verified (in the normal way) discharging however all rab-specifications valid at the site of declaration and *assuming to be valid* instead, the assumed rab-specifications as indicated in the proc/fun-specification.
Proc/fun-call. In addition (to what the original formalism says) it is required to verify the assumed rab-specifications from the rab-specifications currently (assumed to be) valid. This verification requires to show that the abortion condition of the latter rab-specification implies the abortion condition of the former rab-specification.
Alternative construct. In contrast (to what the original formalism says) it is now not required that at least one of the guards is true, but it is required that the falsity of all guards implies the truth of the abortion condition of the rab-specification currently (assumed to be) valid.
All other rules remain unchanged.
(Note 3. The above modification is equally well applicable to any formalism for proving partial correctness. The reader may consult Fokkinga (77) for a detailed presentation. *End of note 3).*

Although parts of the program have to be proved liberally correct, the whole program is proved correct if all rab-specifications are assumed to have false as abortion condition. If however those abortion conditions are assumed to be true - as actually validated by the system -, then the whole program is proved liberally correct only.

(Note 4. Because in the approach of liberal correctness the correctness and robustness assertions are merged into one assertion, the complications described in the motivation of clause (6) of the operational definitions disappear when alternative (6') is chosen and consequently rab's are evaluated recursively. *End of note 4).*

6 Formal Verification of Robustness of Rab-free Programs

Because we are not interested in formalisms as such but only in the existence of intuitively appealing methods, we will employ the formalism for proving liberal correctness for convenience.

We describe first how separate correctness and robustness specifications for a procedure should be translated into a single liberal correctness specification and secondly how robustness of a program part - free of procedure declarations - should be verified in the formalism for liberal correctness. Thus one should verify robustness of procedures separately.

Consider a procedure(body), if it is correct w.r.t. R and S and robust in (id, R', A), then it is liberal correct w.r.t. R ∨ R'. and S, assuming a rab-specification for id, which has A as abortion condition.

Now consider a program part to have been proved robust in (id, R, A). If the program part is proved to be liberally correct w.r.t. R and false, assuming a rab-specification for id which has A as abortion condition and assuming further only rab-specifications with false as abortion condition, then the robustness has been verified. Indeed, postcondition false expresses that the program part can impossibly terminate and the only abortions allowed are those identified by id and taking place in a state satisfying condition A.
(Example 16. The robustness as completed in example 13 may be verified using the invariant relation {$0 \leq i \leq m \wedge A[1..i]$ is extensible and needs (*open-close*) closing parentheses for extension}. Indeed, the falsity of both *open* = *close* and *open* > *close* implies the abortion condition. Further, the invariant relation together with $i = m$ (after the od) imply false, because $A[1..m]$ is not extensible on account of the robustness condition. *End of example 16).*

It is beyond the scope of this paper to treat topics like the soundness of the formalism described in section 5 and 6, the completeness and the conditions under which they are equivalent and so on.

7 Passing Restrictions of Abortions as Parameters

Once restricting abortions is used as a normal control structure, we may consider parametrization of procedures by robustness identifications.

First of all it is necessary - in order to control the construction of (liberally) correct programs and not to invalidate the previously given abstract semantics! - that within procedures distinct (formal and global) identifiers denote distinct (actual) robustness identifications. (The same applies to variables as well!!). Such distinctness may be obtained either by undesirable syntax constraints or by considering the formal parameter specification of an robustness identification f as the declaration rid f (thus creating a *new* identity) for which the "default" rab-definition is rab f by if fi a bar, where a stands for the actual robustness identification.

Second we may choose to define the use in rob-identifications to be the only option for such parameters, or we may as well allow the option to give rab-definitions for such parameters. The latter case corresponds to the PASCAL and Algol 60 value parameter transfer, the former case to the original PASCAL constant parameter transfer.
The value-like parameter transfer can *very* easily be realized by the constant-like parameter transfer; the converse does not hold - in languages such as Algol 60 and PASCAL. As an example, the standard robustness identification *abort* may be considered to be passed to the user program with the only option to use it in rob-identifications.

8 Conclusion

Based upon abstract, intrinsic needs we have described a programming concept. It is applicable to correct programs, exchanges robustness for a relaxation of the program specification and treats all robustness on the same footing, disregarding the origin (which may lay in the system, a library procedure or the program itself). The operational tool proposed has the attractive feature that it merely brings about additions to the program text without any alteration, so that the original program documentation and correctness proof are not affected but only need be extended.

All restricted forms of the goto, as mentioned in the introduction, are unable to deal with system generated errors and even the too unrestricted label variables of Algol 68 and PL/I are unpractical to do so. The Jumpout facility of Pop-2, Burstall (71), and the J-operator of Landin (66) require the presence of procedure variables and have a rather strange (standard) function for what we have made into a syntactic construct.

Much of the work done on exception handling does not (only) deal with "fatal errors" - after which the current computation can not be continued - but deal (in the first place) with "relatively infrequent events" so that continuation is possible, be it after some "exception handling". In our opinion other control structures, such as procedure parameters or may be new ones, should solve the (abnormal?) flow of control related to "exceptional events"; they should not be mixed up with the handling of "fatal errors". E.g. Levin (77) proposes to return control after execution of the exception handler to the place of invocation; Goodenough (75) 's proposal and the PL/I-ON construct have in addition the option to abort the current computation. Our proposal might be viewed as a deliberate simplification of the latter two constructs.

Most closely related are the exit-facility of Jones (75) - it may be considered as a single standard robustness identification - and the signal-enable facility of BLISS 11, Dec (74), - corresponding to rob-identifications and rab-definitions respectively -. But these and all of the above require to take care of the exceptional cases while contructing the program, thus lacking a separation of concern as made possible by the proposed exchange of robustness.

So we hope to have made a proposal in which a remaining troublespot in the area of control structures has fully been brought on the level of a calculus for the derivation of correct programs.

REFERENCES

Adams, J.M. (1977); A General Verifiable Iterative Control Structure. IEEE SE-3 (March 77)2, 144-150.

Bron, C. & Fokkinga, M.M. & de Haas, A.C.M. (1976); A proposal for dealing with abnormal termination of programs. TW-Memo 150 (nov. 1976). Twente University of Technology, Enschede, Netherlands.

Burstall, L.M. & Collins, J.S. & Popplestone, R.J. (eds.) (1971); Programming in POP-2. Edinburg, University Press (1971).

DEC, Digital Equipment Corporation (1974); BLISS-11 Programmer's Manual. Maynard, Mass., 1974.

Dijkstra, E.W. (1968); Goto-statement Considered Harmful. CACM 11 (1968) 3, 147-148.

Dijkstra, E.W. (1975); Guarded Commands, Nondeterminacy and Formal Derivation of Programs. CACM 18 (1975) 453-458.

Dijkstra, E.W. (1976); A Discipline of Programming. Prentice-Hall. New Jersey (1976).

Fokkinga, M.M. (1977); Axiomatization of Declarations and the Formal Treatment of an Escape Construct.To appear in Proceedings IFIP TC-2 Working Conference on Formal Description of Programming Concepts (August 1-5, 1977, St. Andrews, New Brunswick, Canada). (Ed. E.J. Neuhold) North Holland Publ. Co.

Goodenough, J.B. (1975); Exception-Handling: Issues and Proposed Notation. CACM 18 (1975) 683-696.

Hehner, E.C.R. (1976); do considered od: A Contribution to the Programming Calculus. Tech. Rep. CSRG-75 (1976) University of Toronto.

Hoare, C.A.R. (1969); An Axiomatic Basis of Computer Programming. CACM 12 (1969) 576-580, 583.

Jones, C.B. (1975); Formal Definition in Program Development.In Programming Methodology, Lect. Notes in Comp. Sc. 23 (1975) 387-443 (Springer Verlag).

Knuth, D.E. (1974); Structured Programming with goto statements. ACM Computing Surveys 6 (1974) 4, 261-302.

Levin, R. (1977); Program Structures for Exceptional Condition Handling, Ph.D. Thesis, Dept. of Comp. Sc. Carnegie-Mellon Univ., Pittburgh Pennsylvania, 1977.

Wirth, N. (1971); The Programming Language Pascal. Acta Inf 1 (1971) 35-63

Zahn, C.T. (1974); A Control Statement for natural top-down structured Programming. Proc. Symp. on Programming (Paris, France 1974) Lect. Notes in Comp. Sc. 19 (1974).