AFDELING INFORMATICA IW 154/80 DECEMBER
(DEPARTMENT OF COMPUTER SCIENCE)

R.J.R. BACK

CORRECTNESS OF EXPLICITLY SPECIFIED PROCEDURES

Preprint

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

1980 Mathematics subject classification: 68B10

ACM Computing Reviews: 5.24

Correctness of explicitly specified procedures [*)]

by

R.J.R. Back[**)]

ABSTRACT

Explicit specification of a procedure, by a program statement describing the effect of executing the procedure, is contrasted with the usual way in which a procedure is specified in program verification, by giving its pre- and postconditions. The explicit technique is found to give simpler and more readable specifications. A proof system for the correctness of explicitly specified procedures is described. The notion of correctness used is partial correctness together with absence of run-time errors. The proof rules cover both recursive procedures and procedures with parameters. Special attention is given to the modular structure of programs with procedures. An extension of the technique, making explicit specifications as powerful as pre- and postconditions is also presented.

KEY WORDS & PHRASES: Procedure proof rules, specifications, partial correctness, program failure, modules, recursion, parameters

---

*) This report will be submitted for publication elsewhere.

**) Present address: University of Helsinki, Computing Centre, Tukholmankatu 2, 00250, Helsinki 25

1. INTRODUCTION

The purpose of using procedures in programs is to allow the construction and verification of programs to be factored into a number of independent tasks. Successful use of procedures in program construction requires a careful specification of the effect of each procedure, independently of how the procedure is implemented. The specification serves as an interface between the program module implementing the procedure and the program modules which call the procedure. It lays down the properties which the calling modules may assume about the effect of a procedure call, and at the same time it describes what the procedure implementation should achieve. The specifications form the prime means by which we can understand the working of larger programs, built as collections of modules.

The techniques for specifying procedures vary, but can essentially be classified into two broad categories: implicit and explicit specifications. In the former, the effect is described by giving certain properties which any call on the procedure will satisfy. In the latter, the effect is described by presenting another, simpler mechanism, which is to have essentially the same effect as the procedure specified. The former technique is the one usually favored in program verification, where procedures are specified by their pre- and postconditions. The latter is more common in program construction, as it is practiced in real life programming projects, where procedures usually are described by the actions an invocation will cause.

To make things more concrete, let us consider an example procedure and the different ways in which it can be specified. We choose a procedure for computing the factorial function, with special attention given to the detection of overflow during the computation. An explicit, informal specification of this procedure could be something like the following:

```
procedure fact:
          The procedure will set the variable x
          to the factorial of n, if possible.
          If n<0 this is not possible and the
          effect of the procedure is undefined.
          If n > maxint, the biggest integer allowed,
          then this is also not possible. In this
          case the overflow indicator oflo
          (which is assumed to be 0 before
          calling the procedure) is set to 1.
```

Such a specification is easy to understand and provides the programmer with all the flexibility of natural languages. However, it is difficult to make informal specifications precise enough (is e.g. the value of n changed by this procedure, what happens with oflo when $n! \le maxint$, does the procedure have other side-effects). Moreover, when one tries to make them precise, they become verbose and difficult to comprehend. Also, this kind of specifications are not suitable for a mathematically rigorous reasoning about program correctness.

Consider now an implicit specification of the procedure, in terms of pre- and postconditions. A first attempt would be:

```
procedure fact:
          precondition:  n ≥ 0 & oflo = 0;
          postcondition: (n ≤ maxint => x = n!) &
                         (n > maxint => oflo = 1).
```

However, this is probably not what we want, because it does not prevent certain undesired side-effects of the procedure. We would not e.g. want the procedure to change any other variables than x and oflo. If y,z, ... are the other variables accessible to the procedure, the procedure specification becomes:

```
procedure fact:
        precondition:   $n \geq 0$ & oflo=0 & $n=n_0$
                        & $y=y_0$ & $z=z_0$ & ...;
        postcondition:  ($n \leq maxint$ => x=n! & oflo=0)
                        & (n>maxint => oflo=1) & $n=n_0$
                        & $y=y_0$ & $z=z_0$ & ...;
```

The advantages of such a specification is that it is flexible and precise. However, it is not so easy to comprehend, as it has a similar tendency to become verbose as the informal explicit specification technique.

A third possibility is to give an explicit and formal specification of the procedure. This means that we write the specification as a simple program. Using e.g. Dijkstra´s guarded commands [7], the procedure can be specified as follows:

```
procedure fact:
        if n ≥ 0 and oflo = 0 ->
                if n ≤ maxint -> x:= n!
                [] n > maxint  -> oflo:=1
                fi
        fi
```

This specification is much simpler than the preceeding one, yet it is as precise. There are two reasons for this simplicity. First, the default assumption about variables which are not mentioned in the specification is the opposite to the one used in the previous specification. There a variable not mentioned in the postcondition is allowed to have any value whatsoever upon exit from the procedure. Here the opposite is the case: a variable not mentioned in the specification remains unchanged by default. Actually we have an even finer control, as we know that any variable not explicitly changed by an assignment statement will keep its old value (e.g. the value of n is not changed by the procedure and either x or oflo remains unchanged, depending on the branch taken). Another reason for the simplicity of this specification is the use of the conditional statement. This gives a well-needed structuring of the specification, clearly indicating the different possible cases that are covered by it.

A disadvantage of this method is that it can lead to overspecification. Thus, we cannot e.g. leave the value of a variable unspecified. In the previous specification we left the value of x undefined when n>maxint, but in this specification we have to say what the value of x is in this case too. It is chosen to be the same as the initial value of x. We will show how to overcome the shortcoming of overspecification in section 7. Other disadvantages of this specification technique are a possible lack of flexibility and the potential for misuse. If one were to restrict the operations and tests allowed in such a specification to only those available as primitives in the programming language used, then this would severly limit the expressive power of the specifications (imagine writing the specification above without being allowed to use the factorial function). The possibility of misuse follows from this, when the programmer tries to circumvent the lack of suitable basic operations by programming them in the specification. This can make the complexity of the specification similar to the complexity of the procedure implementation, so one might as well use the latter directly as a specification.

Both these disadvantages can be avoided, if we restrict the programming language allowed in specifications drastically, by e.g. disallowing the use of loops, while at the same time allowing any mathematically well-defined functions and predicates to be used in the specifications. This encourages the programmer to consider the design of the basic concepts needed as a task separate from writing the specifications themselves, thereby inducing a well-needed separation of concerns.

We will in this report consider explicit and formal procedure specifications from the point of view of program verification. We will design a simple programming language with explicit specification of procedures, and show how the correctness of such programs can be proved. The soundness of the proof system to be presented here will be proved in an accompanying report [4].

2. A SIMPLE PROGRAMMING LANGUAGE

We start by describing a simple programming language in which to write programs with parameterless recursive procedures (later on we extend the languages with parameterized procedures). The language will be defined in three steps, by first concentrating on procedure specifications, then describing simple programs with procedure calls and finally describing the modular structure of programs with procedures.

Procedure specifications

Let Id be a set of identifiers, let Sig be a signature, with constant, function and predicate symbols, let Exp be a set of expressions over this signature and let Bool be the set of boolean expressions over the signature. The set of procedure specifications is denoted Spec and is defined recursively by

$$\begin{aligned} S ::= &\ \text{skip} \mid x := e \mid S_1 ; S_2 \mid \\ &\ \text{if } b_1 \text{ -> } S_1 \text{ [] } \ldots \text{ [] } b_k \text{ -> } S_k \text{ fi} . \end{aligned}$$

Here x is a list of distinct identifiers in Id, e is an equally long list of expressions in Exp, $b_1,\ldots,b_k$ are boolean expressions in Bool and $S, S_1,\ldots,S_k$ are specifications in Spec. The last example in the previous section obviously conforms to this syntax, which, of course, is just a subset of the guarded commands of [7].

As with the guarded commands, execution of a specification S may lead to an abortion, i.e. the execution fails. An abortion occurs if a conditional statement is reached with none of the guards true, or if an expression or boolean expression is evaluated with arguments for which it is not defined. Also, because of the possible nondeterminism in the conditional statement (the guards need not be mutually exclusive), there may be more than one possible execution of S.

A procedure will satisfy a given specification S in Spec, if the following two conditions are met. First, the procedure is not allowed to fail for initial states in which S is guaranteed not to fail. For the procedure fact, described in the previous section, this means that the procedure may not fail when $n \geq 0$ and oflo=0.

Secondly, the effect of executing the procedure is to be essentially the same as that of executing S. By this we mean that if the procedure is started in an initial state in which S is guaranteed not to fail, any possible final state of the procedure must be a possible final state of S too. If S is deterministic, this means that the procedure must produce the same final state as S if it terminates (however, the procedure is allowed to run for ever). For procedure fact, this means that if initially $n \geq 0$ and oflo=0 and the procedure terminates, then either only x has been changed, with $x = n!$ (when $n! \leq$ maxint) or only oflo has been changed, with oflo = 1 (when $n! >$ maxint).

The motivation for this definition of satisfaction is that the procedure should be a correct refinement of the specification, in the sense that it should be correctness preserving. This notion of correct refinements is described in detail in [3].

This specification language is rather restricted in expressive power, and is not as powerful as the implicit specifications with pre- and postconditions, when the signature is kept fixed. We will later ,in section 7, extend the language in a way which makes it as expressive as the implicit method.

Simple programs

The signature Sig is to contain all constant, function and predicate symbols needed in order to give simple specifications of procedures. The constants, functions and predicates one is actually allowed to use in a program form a subset of Sig, which we denote Sig´. Let Exp´ and Bool´ be the corresponding set of expressions and boolean expressions over Sig´.

The statements Stat are defined recursively by

$$\begin{aligned} S ::= {} & \text{skip} \mid x := e \mid p \mid B \mid S_1 ; S_2 \mid \\ & \text{if } b_1 \to S_1 \,[]\, \ldots \,[]\, b_k \to S_k \text{ fi} \mid \\ & \text{do } b_1 \to S_1 \,[]\, \ldots \,[]\, b_k \to S_k \text{ od} . \end{aligned}$$

Here we have added procedure calls (p), blocks (B) and iterations (do ... od). The expressions in e and the boolean expressions b_i are required to be in Exp´ and Bool´ respectively, while p must be in Id. The syntax of blocks is described below. As evident from the syntax, a statement has more powerful control structures than a specification but less powerful basic statements and guards.

A declaration will be of the form

$$D ::= \text{var } x \mid \text{const } x \mid \text{proc } p\text{: } S \text{ end} ,$$

where x is a list of distinct identifiers, p is an identifier and S is a specification. For simplicity, no type information is associated with the variables and constants, which are assumed to be all of the same type (in examples the type integer is assumed). Extending the language with type information does not present any great difficulties, and can be done essentially as in [5].

The difference between declaring a variable by "var x" and by "const x" is that in the former case the value of x may be changed by an assignment statement and in the latter case not. The need for making such a distinction arises in connection with the parameter mechanism, to be described in section 8. However, it also simplifies the proofs of program correctness, by decreasing the amount of detail which needs to be checked.

A procedure declaration

proc p:S end

associates a specification S with the procedure identifier p. (Note that S is not the body of the procedure, only a specification of the effect of the procedure.) The language thus requires each procedure to be associated with an explicit specification of the kind described above.

An environment E is a (possibly empty) sequence of declarations, i.e.

$$E ::= D_1;\ldots;D_n.$$

The set of environments is denoted Env. It is not allowed to declare an identifier twice in an environment E. Finally, a block consists of an environment E and a statement S, and is of the form

B ::= begin E;S end.

The set of blocks is denoted Block.

Modular structure of programs

The procedure bodies form the modules in this programming language. A procedure body is of the form

M ::= body p: E;S;M_1;...;M_n end.

Here p is a procedure identifier and M_1, ..., M_n are modules (procedure bodies), $n \geq 0$. We require that each module M_i is declared in E. Let Mod be the set of modules.

Notice that, because of the block construct, there may be local variables also in the statement S of a body declared as above, in addition to the local variables declared in E. The latter can be seen as communication variables, shared between S and the modules $M_1,\ldots,M_n$. The variables which are declared in a block in S are, on the other hand, strictly local, only needed as temporary variables in the computation of S.

A closure of a module M is a module of the form

```
body q: E;M end,
```

where E declares all identifiers used but not declared in M (i.e. global identifiers). In particular, E must contain a declaration of M itself.

In the design of the modular structure of this language, we have followed ADA [9] and Alphard [10] in separating the specification of a module from its implementation. Besides having a benevolent effect on the clarity of the program structure, this will also simplify the design of proof rules for program correctness and make the structure of correctness proofs easier to grasp.

An example program

As an example of a program written in this language, we will show how to compute the factorial of 25, using the procedure fact specified in the introduction. The following closure describes this program.

```
body main:
        const maxint;
        var x;
        proc fact25:
                if 25!≤maxint -> x:=25 fi
        end;


        body fact25:
                var n;
                var oflo;
                proc fact:
                        if 0≤n & oflo=0 ->
                                if n!≤maxint -> x:=n!
                                [] n!>maxint -> x,oflo:= 0,1
                                fi
                        fi
                end;
                begin   n:= 25;
                        oflo:= 0;
                        fact
                end
        end;
end.
```

The correctness of this program, i.e. that the body of fact25 has the effect of assigning to x the value 25! when 25!≤maxint, should be evident from an inspection of the code. This correctness does not depend on the way in which the procedure fact is implemented, only the specification of fact is needed for the correctness argument. The procedure fact is implemented as follows:

```
body fact:
        begin   var m;
                if n=0 v n=1 -> x:= 1
                [] n>0 ->
                        n:= n-1;
                        fact;
                        n:= n+1;
                        if oflo=1 -> skip
                        [] oflo=0 ->
                                m:= maxint/n;
                                if x≤m -> x:= x*n
                                [] x>m  -> x,oflo:=0,1
                                fi
                        fi
                fi
        end
end;
```

Here the procedure fact is called recursively inside the body of fact. In the computation, a test is performed to check that no overflow can occur. If an overflow would occur, the computation of the factorial value is skipped and instead the overflow indicator is set to 1. This scheme prevents an overflow from actually occurring during the computation, thus giving the calling program full control of the effect of a procedure call.

The closure as a whole has the following structure:

```
body main:
        const maxint;                         ⎤
        var x;                                ⎬ E1
        proc fact25: ...                      ⎥
        end;                                  ⎦
        body fact25:
                var n;                        ⎤
                var oflo;                     ⎬ E2
                proc fact: ...                ⎥
                end;                          ⎦
                begin ... fact ...            ⎫ S2
                end;                          ⎭
                body fact:
                        begin ... fact ...    ⎫ S3
                        end                   ⎭
                end
        end
end.
```

More concisely, the modular structure can be described by

M_0 = body main: E_1;M_1 end
M_1 = body fact25: E_2;S_2;M_2 end
M_2 = body fact: S_3 end.

Note that here the highest module M_0 does not contain any statement to be executed (i.e. it is the closure of the module fact25), while the lowest module M_2 does not contain any submodules.

3. CORRECTNESS OF STATEMENTS

The proof rules for showing correctness of programs with procedures are on two levels. On the lower level we have the proof rules by which the correctness of statements is established. On the higher level there is a proof rule by which the correctness of the whole program, seen as a collection of modules, is established. In this section we present the proof rules for statements. The proof rule for programs is derived in the following two sections.

Let Assn be the set of assertions (first-order formulas) over the signature Sig. A correctness formula will be of the form

$$H ::= E \;]\!-\; P\{S\}Q,$$

where P and Q are assertions in Assn, E is an environment and S is a statement. We require that each identifier free in P or Q must be declared in E, either as a variable or as a constant, that assignments in S are only to identifiers declared as variables in E and, as before, that no identifier is declared twice in E.

The validity of a correctness formula is defined by

```
E ]- P{S}Q  iff   if P holds initially
                  then S cannot fail
                  and  if S terminates
                       then Q holds upon termination.
```

Thus correctness is here taken to mean partial correctness, together with absence of failure during program execution. (As the execution of S can be nondeterministic, we require that none of its possible executions can lead to abortion.)

For simplicity, we assume in the sequel that expressions and boolean expressions are always well-defined. This means that the only possibility of failure comes from reaching a conditional statement with none of the guards true. The proof system can easily be extended to handle expressions and boolean expressions which are only partially defined, in a manner similar to the one used in [5].

The proof rules for statements are as follows:

0. $$\frac{E\ ]\text{-}\ P\{S\}Q,\ P'{=}{>}P,\ Q{=}{>}Q'}{E\ ]\text{-}\ P'\{S\}Q'}$$

1. $E\ ]\text{-}\ P\{\text{skip}\}P$

2. $E\ ]\text{-}\ P[e/x]\{x{:=}e\}P$

3. $$\frac{E\ ]\text{-}\ P\{S_1\}Q,\quad E\ ]\text{-}\ Q\{S_2\}R}{E\ ]\text{-}\ P\{S_1;S_2\}R}$$

4. $$\frac{E\ ]\text{-}\ P\&b_i\{S_i\}Q,\ i=1,\dots,k,\quad P{=}{>}bb}{E\ ]\text{-}\ P\{\text{if } b_1 \ {-}{>}\ S_1\ [\,]\dots[\,]\ b_k\ {-}{>}\ S_k \text{ fi}\}Q}$$

5. $$\frac{E\ ]\text{-}\ P\&b_i\{S_i\}P,\ i=1,\dots,k}{E\ ]\text{-}\ P\{\text{do } b_1\ {-}{>}\ S_1\ [\,]\dots[\,]\ b_k\ {-}{>}\ S_k \text{ od}\}P\&{-}bb}$$

6. $$\frac{E;E'\ ]\text{-}\ P\{S\}Q}{E\ ]\text{-}\ P\{\text{begin } E';S \text{ end}\}Q}$$

7. E; proc p:S end; E´]- P{S}Q

E; proc p:S end; E´]- P{p}Q

We use the abbreviation bb = b_1v...v b_k in proof rules 4 and 5. The following constraints on these rules are necessary to ensure that the restrictions on correctness formula given above are not violated:

Rule 0:All identifiers free in P´ or Q´ must be declared in E.

Rule 2:Each identifier in x must be declared as a variable in E and each identifier occurring in e must be declared in E, either as a variable or as a constant.

Rule 6:All identifiers free in P or Q must be declared in E and E´ may not contain declarations of identifiers already declared in E.

The absence of failures is checked in rule 4, by the condition P => bb. Redeclaration of identifiers in inner blocks is disallowed, by the constraint on rule 6. It is, of course, always possible to rename identifiers in inner blocks in a way which guarantees that this condition is met.

The most important rule here is rule 7 for procedure calls. It essentially says that whatever is true of a procedure specification will also be true of the procedure call. The simplicity of this rule is the most valuable payoff from using statements as procedure specifications. A similar rule for procedure calls appears in many proof systems (see e.g. [1]), with S standing for the body of the procedure, which, moreover, is assumed not to contain any recursive calls. The rule given here differs from these in that S is the specification of the procedure and not the body. A separate proof rule will be given for handling the procedure body, which is allowed to contain recursive calls on itself.

4. PREDICATE TRANSFORMERS

Before we can present the proof rule for modular programs, we need to define the weakest precondition and strongest postcondition of a specification with respect to a given condition. The weakest precondition WP(S,R) of the specification S for the condition R, S in Spec and R in Assn, is defined exactly as in [7]. For completeness, we repeat the definition here:

(1) WP(skip, R) = R

(2) WP(x:=e, R) = R[e/x]

(3) $WP(S_1;\ S_2,\ R) = WP(S_1,\ WP(S_2,\ R))$

(4) $WP(\text{if } b_1 \text{ -> } S_1 \text{ [] ... [] } b_k \text{ -> } S_k \text{ fi},\ R)$

$$= bb \ \ \& \ \ \underset{i=1}{\overset{k}{\&}} (b_i \Rightarrow WP(S_i,\ R)$$

Thus WP(S,R) is the weakest precondition which guarantees that execution of S cannot lead to abortion and that any execution terminates in a final state satisfying R.

The strongest postcondition SP(R,S) of a specification S and an assertion R is again defined by:

(1) $SP(R, skip) = R$

(2) $SP(R, x{:=}e) = \exists x_0.(R[x_0/x] \; \& \; x=e[x_0/x])$

(3) $SP(R, S_1; S_2) = SP(SP(R, S_1), S_2)$

(4) $SP(R, \text{if } b_1 \text{ -> } S_1 \; [] \ldots [] \; b_k \text{ -> } S_k \text{ fi})$

$$= \bigvee_{i=1}^{k} SP(R \; \& \; b_i, S_i)$$

Thus SP(R,S) is the strongest postcondition which holds when S terminates normally (i.e. without failure), if R was true initially.

The formula SP(R,S) can be simplified if we assume that R is of the form P & v=t, where P is an assertion, v is a list of distinct variables $v_1,\ldots,v_n$ and t is a list of terms, of the same length as v, such that each variable assigned to in S occurs in v, and no variable in v occurs in any term in t or occurs free in P. In this case the strongest postcondition for the assignment statement becomes:

(2´) $SP(P \; \& \; v=t, x{:=}e) = P \; \& \; v=u$,

where x is the list $x_1,\ldots,x_m$ and for $i = 1,\ldots,n$ we have $u_i = e_j[t/v]$ if v_i is x_j for some j, $1 \leq j \leq m$, and otherwise $u_i = t_i$. This is simpler than the previous formulation, in that the existential quantifier has been eliminated from the strongest postcondition.

Rules (1), (2´) and (3) preserve the appropriate form of the precondition P & v=t. Rule (4), however, violates the condition that no variable in v may occur in P. We change it to the equivalent rule

(4′) $SP(P \mathbin{\&} v=t,\ \text{if } b_1 \text{ -> } S_1 \text{ [] ... [] } b_k \text{ -> } S_k \text{ fi})$

$$= \bigvee_{i=1}^{k} SP(P \mathbin{\&} b_i[t/v] \mathbin{\&} v=t,\ S_i)$$

In addition, we need the following general result about SP(R,S):

$$SP(\bigvee_{i=1}^{k} R_i,\ S) = \bigvee_{i=1}^{k} SP(R_i,\ S)$$

This allows us to compute the strongest postcondition of a specification of the form $S_1;S_2$, where S_1 is a conditional statement. In such a case we compute

$$SP(P\&v=t,\ S_1;S_2)$$

$$= SP(SP(P\&v=t,S_1),S_2)$$

$$= SP(\ V\ (P_i\&v=t_i),\ S_2)$$

$$= V\ SP(P_i\&v=t_i,S_2)$$

$$= V\ (Q_i\&v=u_i).$$

With these changes, the operator SP will always be applied to a precondition which satisfies the given restrictions, thus allowing us to use the simpler rule for assignments in computing strongest postconditions.

5. CORRECTNESS OF MODULAR PROGRAMS

We need the predicate transformers in order to prove that an implementation of a procedure satisfies the specification given for it. Consider the environment

$$E = E_1\text{; proc p:S end; } E_2$$

and the body of p defined as

$$M = \text{body p: B end,}$$

i.e. there are no local modules in M. Proving that the body of p satisfies the specification will then amount to proving that

$$E\text{; const } v_0 \text{]- } P\{B\}Q$$

holds, where v is the list of identifiers declared as variables in E, v_0 is an equally long list of distinct identifiers not occurring in E or B,

$$P = WP(S,\text{true}) \;\&\; v=v_0 \quad \text{and}$$

$$Q = SP(v=v_0,S).$$

This expresses the requirement on a correct implementation laid down earlier: The implementation must not fail for initial states in which S is guaranteed not to fail (i.e. B may not fail when P holds), and any possible final state of the implementation for such an initial state must be a possible final state of S (any final state of B must then satisfy Q).

We are now ready to present the proof rule by which the correctness of modular programs is established. We introduce another kind of correctness formula for this purpose, of the form:

$$H ::= E \;]\!-\; M,$$

where E is an environment and M is a module (i.e. a procedure body), of the form

$$E = E_1;\ \text{proc } p{:}S \text{ end};\ E_2 \quad \text{and}$$
$$M = \text{body } p{:}E';S';M_1;\ldots;M_n \text{ end}.$$

We will assume that no identifier is redeclared in the closure "body E;M end", i.e. for each identifier there is only one declaration in the closure. The reason for this restriction is that we want static scope for procedures. The proof rule for procedure calls together with the proof rule for procedure implementations to be given below, however, give dynamic scope instead of static scope. With the restriction on redeclaration we avoid this problem, as dynamic and static scope will then agree. It is always possible to transform a program into an equivqalealent one which satisfies this restriction, by a suitable renaming of identifiers.

The proof rule for modules will be as follows:

7.
$$\frac{\begin{array}{c} E;E' \;]\!-\; M_i,\ i=1,\ldots,n \\ E;\ \text{const } v_0 \;]\!-\; P\{\text{begin } E';S' \text{ end}\}Q \end{array}}{E \;]\!-\; \text{body } p{:}\ E';S';M_1;\ldots;M_n \text{ end}}$$

where v_0, P and Q are as above.

Thus, to prove that E]- M is correct, one has to prove that the effect of the block "begin E´;S´ end" is the same as the effect of S and one has to prove that each local module of M satisfies its specification. Note that this proof rule does not require that each procedure specification is implemented by a module, so it is also applicable to the correctness of of programs which are being developed and where therefore not all modules have been written yet.

The soundness of the proof system presented above will be shown in an accompanying report [4]. However, it is worth noting that this proof system would not be sound if we did not require absence of failure in E]- P{S}Q, i.e. if we would use the following alternative definition of validity:

```
E ]- P{S}Q  iff if P holds initially
                and S terminates without failure
                then Q holds upon termination.
```

This would allow S to fail for initial states in which P holds. A simple example suffices to show that the proof rules are not sound with respect to this definition of validity:

Consider the specification

```
proc p: if x>0 -> x:= x-1 fi end.
```

Computing weakest preconditions and strongest postconditions of this specification (call it S), gives

$$WP(S,true) = x>0$$
$$SP(x=x_0,S) = x_0>0 \ \& \ x = x_0-1.$$

Let the implementation of p be

```
body p: if x>0 -> x:= x-1
        [] x=0 -> x:= -10
        fi
end.
```

Call the implementation S´. Let for the moment E]- P{S}Q be interpreted with the alternative definition of validity. We then have that

$$E\ ]\!-\ x > 0\ \&\ x = x_0\ \{S'\}\ x_0 > 0\ \&\ x = x_0 - 1,$$

so the procedure is correctly implemented. Also,

$$E\ ]\!-\ x \geq 0\ \{S\}\ x \geq 0$$

holds. By rule 7, we may then deduce that

$$E\ ]\!-\ x \geq 0\ \{p\}\ x \geq 0,$$

which, however, is not valid for the alternative interpretation of E]- P{S}Q.

6. PROOF OF THE EXAMPLE PROGRAM

We will illustrate the use of these proof rules by showing how to establish the correctness of the example program of section 2. The example program is of the form

M_0 = body main: $E_1;M_1$ end
M_1 = body fact25: $E_2;S_2;M_2$ end
M_2 = body fact: S_3 end,

with E_1, E_2, S_2 and S_3 as indicated in section 2. To prove the correctness of the closure, we have to prove that

$$\emptyset\ ]\!-\ M_0$$

holds, where ∅ stands for the empty environment. Using proof rule 8, this means that we have to prove

E_1]- M_1

(there is no statement in M_0 to consider). This again requires us to prove

E_1; const v_0]- P_1{begin E_2;S_2 end}Q_1 and (1)
E_1;E_2]- M_2,

with v_0, P_1 and Q_1 appropriately chosen. The latter reduces to proving

E_1;E_2; const w_0]- P_2{begin S_3 end}Q_2, (2)

with w_0, P_2 and Q_2 again appropriately chosen. Thus, all in all, use of proof rule 8 will generate all correctness formula for statements which need to be proved in order to establish that the program as whole is correct. In this case we have to prove that (1) and (2) hold.

Consider first (1). We have that

```
E1 =    const maxint;
        var x;
        proc fact25:
                if 25!≤maxint -> x:= 25! fi
        end.
```

Thus only x is variable in E_1. We now have that

WP(fact25, true) = $25! \leq$ maxint and

SP($x=x_0$, fact25) = $25! \leq$ maxint & $x = 25!$

Proving (1) thus amounts to proving

$$E_1 \;]\!-\; \begin{array}{l} 25! \leq \text{maxint} \\ \{\text{begin } E_2;S_2 \text{ end}\} \\ x = 25!\ . \end{array}$$

The conjunct 25! $\leq$ maxint can be omitted from the postcondition as it does not involve any variables which are changed in the statement.

Consider now (2). We have

```
E1;E2 =
        const maxint;
        var x;
        proc fact25: ... end;
        var n;
        var oflo;
        proc fact:
                if 0≤n & oflo=0 ->
                        if n!≤maxint -> x:= n!
                        [] n!>maxint  -> x,oflo:= 0,1
                        fi
                fi
        end.
```

This environment contains the variables x, n and oflo. We compute

$WP(fact, true) = 0 \leq n \ \& \ oflo=0$ and

$SP(fact, x,n,oflo = x_0,n_0,oflo_0) =$

$$0 \leq n_0 \ \& \ oflo=0 \ \&$$

$$(n_0 \leq maxint \ \& \ x,n,oflo = n_0!,n_0,oflo_0) \ v$$

$$(n_0 > maxint \ \& \ x,n,oflo = 0,n_0,1).$$

This means that we have to prove that

$E_1;E_2;$ const $x_0,n_0,oflo_0$]-

$0 \leq n \ \& \ oflo=0 \ \& \ x,n,oflo = x_0,n_0,oflo_0$

{begin S_3 end}

$(n_0 \leq maxint \ \& \ x,n,oflo = n_0!,n_0,oflo_0) \ v$

$(n_0 > maxint \ \& \ x,n,oflo = 0,n_0,1).$

Consider proving the correctness of this last formula. Knowing that the procedure call rule simply amounts to a macro substitution of the specification for the call, we can perform this substitution in advance, i.e. we substitute in S_3 the specification of fact for the call on fact. This then gives us the following correctness formula:

```
E_1;E_2; const x_0,n_0,oflo_0 ]-

        0≤n & oflo=0 & x,n,oflo = x_0,n_0,oflo_0
       {begin
               var m;
               if n=0 v n=1 -> x:= 1
               [] n>0 ->
                       n:= n-1;
                       if 0≤n & oflo=0 ->
                               if n!≤maxint ->
                                       x:=n!
                               [] n!>maxint  ->
                                       x,oflo:= 0,1
                               fi
                       fi;
                       n:= n+1;
                       if oflo=1 -> skip
                       [] oflo=0 ->
                               m:= maxint/n;
                               if x≤m -> x:=x*n
                               [] x>m  -> x,oflo:= 0,1
                               fi
                       fi
               fi
        end}
        (n_0≤maxint & x,n,oflo = n_0!,n_0,oflo_0) v
        (n_0>maxint & x,n,oflo = 0,n_0,1).
```

The correctness of this is easily seen by analyzing the program text and the different cases which can arise. A formal proof can be given using the proof rules of section 3.

7. STRENGTHENING THE SPECIFICATION LANGUAGE

We observed in the introduction that explicit specifications, as defined there, are not as powerful as implicit specifications using pre- and postconditions, on a given signature Sig. The reason for this is that one with the latter specifications can describe any first-order definable relation in Sig between input and output, while explicit specifications clearly are not capable of this. To remedy this shortcoming, we introduce a nondeterministic assignment into the specification language, by adding the following production to the syntax definition of Spec:

```
S ::= x:= y.Q.
```

Here x and y are lists of distinct identifiers, of the same length, and Q is a first-order formula on Sig.

The effect of this statement is to assign to the variables in x the corresponding values y, where the values y are chosen so that condition Q is satisfied (Q may contain free occurrences of variables in x and y). The choice of values for y is nondeterministic if there is more than one value combination which satisfies Q. The statement is considered to fail if there is no values for y which satisfy Q.

As an example, consider the following procedure specification:

```
proc sqroot:
        if u>0 -> u:= v.($(v-1)^2 < u \leq v^2$) fi.
end.
```

The effect of this procedure is to compute the integer square root of u.

As another example, consider the following alternative specification of fact:

```
proc fact:
        if 0≤n & oflo=0 ->
                if n!≤maxint -> x:= n!
                [] n!>maxint  -> oflo:= 1;
                                 x:= y.true
                fi
        fi
end.
```

This specification allows the procedure to terminate with any value of x in case an overflow would occur during the computation of the factorial.

In addition to allowing nondeterministic assignments in specifications, we also allow the guards in conditional statements to be arbitrary first-order formula. With these extensions, any pre- and postcondition can be expressed as an explicit specification. Thus, assume that we have a procedure p specified by a precondition P & $v=v_0$ and a postcondition Q, where P contains no occurrences of v_0 and Q may contain free occurrences of v and v_0, v being the variables in the environment where p is declared. This specification can be expressed in the strengthened specification language by

$$\text{if } P \text{ -> } v := v'.Q[v/v_0, v'/v] \text{ fi.}$$

To make use of the extended specification language, we need to give a proof rule (or rather an axiom) for the nondeterministic assignment. We also need to give rules for computing the weakest preconditions and strongest postconditions of these.

The axiom for nondeterministic assignment is

$$E\ ]\!-\ \exists y.Q\ \&\ \forall y(Q \Rightarrow R[y/x])\ \{x := y.Q\}\ R.$$

Notice the similarity with the rule of adaptation in HOARE [8]. The first conjunct in the precondition checks that the statement cannot fail, while the second checks that it has the required effect.

The weakest precondition is defined by

$$WP(x:= y.Q,R) = \exists y.Q \ \& \ \forall y(Q \Rightarrow R[y/x]),$$

i.e. it is the same as the precondition in the proof rule.

The strongest postcondition is as follows:

$$SP(R,x:= y.Q) = \exists x_0.(R[x_0/x] \ \& \ Q[x_0/x,x/y]).$$

In the case that R is of the form P & v=t, with P, v and t as in section 3, we get the following form for the strongest postcondition.

$$\begin{aligned} &SP(P \ \& \ v,x = t',t'', \ x:= y.Q) \\ &\qquad = \exists x_0.(P \ \& \ v,x_0 = t',t'' \ \& \ Q[x_0/x,x/y]) \\ &\qquad = P \ \& \ v = t' \ \& \ Q[t''/x,x/y] \\ &\qquad = \exists x'.(P \ \& \ Q[t''/x,x'/y] \ \& \ v,x = t',x'), \end{aligned}$$

where x´ is a list of distinct fresh identifiers. This formula is in the required form, except for the preceding existential quantifier. We can, however, use this definition in conjunction with the following fact about strongest postconditions:

$$SP(\exists x_0.P, \ S) = \exists x_0.SP(P,S)$$

when none of the variables in x_0 occur in S. This allows us to compute strongest postconditions using the simpler rules also in the case when the specification is of the form x:= y.Q; S. In this case we get

$$\begin{aligned} &SP(P \ \& \ v=t,\ x{:}=y.Q;S) \\ &\quad = SP(SP(P \ \& \ v=t,\ x{:}=y.Q),\ S) \\ &\quad = SP(\exists x_0(P' \ \& \ v=t'),\ S) \\ &\quad = \exists x_0(SP(P' \ \& \ v=t',S)) \\ &\quad = \exists x_0(P'' \ \& \ v=t''). \end{aligned}$$

We cannot get rid of the existential quantifiers, but they will not prevent computing with the simpler rules for strongest postconditions.

8. PROCEDURES WITH PARAMETERS

We now show how to handle procedures with parameters. First, we have to extend the syntax in an appropriate way. We add to the definition of declarations the production

D ::= proc p(E):S end ,

where E is an environment which may only contain variable and constant declarations. The statements are extended by the production

S ::= p(a) ,

where a is a list of expressions corresponding to the formal parameters declared in p. Finally, we also add the production

M ::= body p(E): E´;S´;M_1,...,M_n end

to the definition of modules.

The parameter mechanism assumed here is call by constant and call by variable. In the first case the parameter is declared as a constant. The actual parameter may be any expression, and is evaluated upon entry to the procedure. The parameter may not be assigned to in the specification or body of the procedure (i.e. it is a constant in the body). In the second case the parameter is declared as a variable. The actual parameter must then be a variable identifier. We will require that the actual parameters cannot give rise to aliasing. This means that all actual variable parameters must be distinct, and none of them may be used as global variables inside the specification or the body.

Let E contain the declaration

```
proc p(const c; var x): S end,
```

where c and x are identifier lists. The proof rule for calls on procedures with parameters is then

$$9.\quad \frac{E\ ]\!-\ P\{S'\}Q}{E\ ]\!-\ P\{p(e,z)\}Q}$$

where

$$S' = \text{begin var } w;\ w{:=}e;\ S[w/c,z/x] \text{ end}.$$

Here w is a list of distinct identifiers not occurring in E. The proof rule for procedure bodies with parameters becomes:

$$10.\quad \frac{\begin{array}{l} E;E';E''\ ]\!-\ M_i,\ i=1,\dots,n \\ E;E';\text{const } v_0\ ]\!-\ P\{\text{begin } E'';S'' \text{ end}\}Q \end{array}}{E\ ]\!-\ \text{body } p(E'){:}E'';S'';M_1;\dots;M_n \text{ end}}$$

where v is the list of variables declared in E;E´, v_0 is a corresponding list of fresh distinct identifiers and

$$P = WP(S,true) \& v=v_0,$$

$$Q = SP(v=v_0,S).$$

The proof rule for procedure calls is modeled after the syntactic substitution method for handling procedures with parameters described by DE BAKKER[6]. However, the syntactic substitution is here performed on the specification and not the body as is done by de Bakker. This necessitates the no aliasing restriction. Without this restriction, the usual anomalies, arising in connection with implicit specifications due to aliasing, can be shown to arise with explicit specifications too.

9. AN EXAMPLE WITH PARAMETERS

The example program of section 6 is here written using procedures with parameters. The closure main is as follows.

```
body main:
        const maxint;
        var x;
        proc fact25: ...
        end;
        body fact25:
                var oflo;
                proc fact(const n; var y):
                        if 0≤n & oflo=0 ->
                                if n!≤maxint -> y:=n!
                                [] n!.maxint  -> oflo:=1;
                                                 y:=z.true
                                fi
                        fi
                end;
                begin   oflo:= 0;
                        fact(25,x)
                end;
                body fact ...
                end
        end
end.
```

The body of fact is as follows:

```
body fact:
        begin   var m;
                if n=0 v n=1 -> y:= 1
                [] n>1 ->
                        fact(n-1,y);
                        if oflo=1 -> skip
                        [] oflo=0 ->
                                m:= maxint/n;
                                if y≤m -> y:= n*y
                                [] y>m  -> oflo:=1
                                fi
                        fi
                fi
        end
end.
```

Note that the value of y is not set to 0 in case of an overflow. This is allowed by the specification of fact, in which y is assigned an arbitrary value in case of overflow.

Proving the correctness of this program is done analogously with the proof in section 6. The main difference concerns the proof rule for procedure calls. Thus, proving the correctness of the implementation of fact amounts to proving the following correctness formula, for appropriate choices of E, P and Q. We have here substituted the expanded specification of fact for the call fact(n-1,y) in the body.

```
E ]-  P{begin   var m;
                if n=0 v n=1 -> y:= 1
                [] n>1 ->
                        begin   var w;
                                w:= n-1;
                                if 0≤w & oflo=0 ->
                                        if w!≤maxint ->
                                                y:= w!
                                        [] w!>maxint  ->
                                                oflo:= 1;
                                                y:= z.true
                                        fi
                                fi;
                        end;
                        if oflo=1 -> skip
                        [] oflo=0 ->
                                m:= maxint/n;
                                if y≤m -> y:= n*y
                                [] y>m  -> oflo:= 1
                                fi
                        fi
                fi
        end} Q .
```

REFERENCES

[1] APT,K.R., Ten years of Hoare´s logic, a survey. In Proc. 5th Scandinavian Logic Symposium (F.V. Jensen, B.H.Mayoh, K.K. Moller, eds.), pp. 1-44, Aalborg University Press, 1979 (to appear in TOPLAS).

[2] BACK, R.J.R., Correctness preserving program refinements: Proof theory and applications. Mathematical Center Tracts 131, Mathematisch Centrum, Amsterdam 1980.

[3] BACK,R.J.R., On the notion of correct refinement of programs. In Proc. 5th Scandinavian Logic Symposium (F.V. Jensen, B.H. Mayoh, K.K. Moller, eds.), Aalborg University Press, 1979 (to appear in the Journal of Computer and System Sciences).

[4] BACK,R.J.R., Soundness of a proof system for explicitly specified procedures (to appear).

[5] BACK, R.J.R., Checking whether programs are correct or incorrect. Report IW 144/80, Mathematisch Centrum, Amsterdam, 1980.

[6] DE BAKKER, J.W., Mathematical Theory of Program Correctness. Prentice-Hall, 1980.

[7] DIJKSTRA, E.W. A Discipline of Programming. Prentice-Hall, 1976.

[8] HOARE, C.A.R., Procedures and parameters: An axiomatic approach. In Symposium on Semantics of Algorithmic Languages (E. Engeler, ed.), pp. 102-116, Lecture Notes in Mathematics 188, Springer-Verlag, 1971.

[9] ICHBIAH,J.D. & al, Preliminary ADA reference manual. ACM Sigplan Notices 14,6A, June 1979.

[10] WULF, W.A., R.L. LONDON & M. SHAW, An introduction to the construction and verification of Alphard programs. IEEE Trans. on Software Engineering SE-2, 4, pp.253-265, 1976.