A PROOF SYSTEM FOR BRINCH HANSEN'S DISTRIBUTED PROCESSES

by

Marly Roncken

Niek van Diepen

Mark Kramer

Willem P. de Roever

RUU-CS-81-5

February 1981

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

A PROOF SYSTEM FOR BRINCH HANSEN'S DISTRIBUTED PROCESSES

by

Marly Roncken

Niek van Diepen

Mark Kramer

Willem P. de Roever

Technical Report RUU-CS-81-5

February 1981

Department of Computer Science

University of Utrecht

P.O. Box 80.002, 3508 TA Utrecht

the Netherlands

A PROOF SYSTEM FOR BRINCH HANSEN'S DISTRIBUTED PROCESSES

by

Marly Roncken

Niek van Diepen

Mark Kramer

Willem P. de Roever

University of Utrecht

February 1981

Abstract. A proof system is presented for proving partial correctness of Brinch Hansen's Distributed Processes. The system is not complete for programs that may deadlock because of inter-process communication. Proofs of single processes are given in relation to their parallel environment. As usual, a slight mistake was found in a published concurrent algorithm. This time, it concerns a sorting algorithm that can be used as a priority scheduler. The adapted version (for maintaining a priority scheduler) is proved correct.

Keywords and Phrases:
distributed processes in general and Brinch Hansen's notion of them in particular, monitors, concurrency, (simultaneous) message passing, partial correctness in absence of global deadlock, auxiliary variables, process invariant, global invariant, (proof theoretical) environment, proof outline.

CR Categories: 5.24.

1. INTRODUCTION

We present a Hoare-like proof system for Distributed Processes (DP), a language concept defined by Brinch Hansen [4]. This system deals with proofs of partial correctness concerning programs that are free from dead-lock caused by external requests (cf. section 7).

By a distributed model we mean a model, in which there is a system of processors, each having its own local store, and in which processors interact by means of message passing. For these models it seems more natural to look upon synchronization as a simultaneous rather than mutual exclusive action. On this observation we base our proof system.

Because communication may alter the store of a process, and hence may affect the validity of assumptions made in a proof, we need a system that considers proofs of properties about a single process not only within the context of this process but within the context of the whole program. This implies the validity of assertions in a proof of this process with respect to the complete program.
Therefore, we introduce the (proof theoretical) environment of a program. This environment contains the text of the program together with some special assertions: a global invariant and for each process a process invariant. Introduction of this environment has been inspired by de Bakker's book [3].
The global invariant expresses the inter-process communications. It is used in a similar way as the global invariant in the CSP proof system [2].

Further guides for the system came from various related papers by: Apt, Francez and de Roever [1,2], Lamport [5], Levin [6], Owicki and Gries [8].

The rest of the paper is organized as follows. In section 2, we give a brief description of DP. Section 3 contains some simple examples to illustrate several aspects of the language. In section 4 the proof system is presented (4.1, 4.2), together with a discussion of the various rules (4.3). In section 5 the system is applied to prove the examples in section 3. Section 6 contains a more advanced example: a sorting program which is also applicable as a priority scheduler is proved correct. This example was taken from Brinch Hansen's paper [4], but has been adapted, because there was a slight mistake in the original version, which caused the program to function improperly as priority scheduler. Finally, section 7 summarizes and evaluates the results.

2. THE LANGUAGE CONCEPT DP

DP is especially conceived for real-time applications, controlled by microcomputer networks with distributed storage. A real-time program interacts with an environment in which many actions take place simultaneously at high speed. Therefore, it may be convenient that the nondeterministic requests to such programs from its environment are handled in arbitrary order, and that the program never terminates but continues to serve its environment as long as the system works. The last two properties are clearly visible in DP's convention for process execution (cf. 2.4).

2.1. A DP program consists of a fixed number of concurrent and persistent processes that are started simultaneously. A process does not contain parallel statements.
The syntax of a process is as follows:

```
process<name>;
<private variables>
<common procedures>
<initial statement>
```

2.2. Communication

A process can access its private (i.e., local) variables only. No common (i.e., shared) variables are used. However, a process can call common procedures within other processes (but not itself) in the program: external requests (short: requests). When such a request is honoured, a new incarnation of the common procedure being called is created for the requesting process.
Syntax of a procedure:

```
proc<name>(<input parameters> # <output parameters>)
<local variables>
<statement>
```

('#' separates the formal input and output parameters; if only input parameters are used, the separator '#' is deleted.) An external request in process P to procedure qr within process Q is written as:
call Q.qr(u_1, ..., u_n, v_1, ..., v_m), where the actual parameters u_1, ..., u_n, respectively, v_1, ..., v_m agree in number and type with the formal input, respectively, output parameters of procedure qr in process Q.

DP uses call-by-value-result for parameter transfers. To avoid unnecessary complications in the proof system, subscripted variables do not occur as parameters (ch. 9 of [3] shows how this complication can be solved, in general). Furthermore, an external request is regarded as an elementary action within the requesting process: in the above setting, process P is suspended until process Q has completed the request.
In other words the possibility for implementing distributed recursion within a fixed network of processors, has not been realized. If inter-process communications had been allowed to the requesting process during

the call (i.e., after sending the values of the input variables, and before receiving the values of the output variables), then the full expressive power of recursion could have been exploited!
This possibility is too exciting to be ignored, and research is going on to find the appropriate version of Scott's induction rule in this new setting.

2.3. Synchronization in DP

Synchronization is established by means of nondeterministic statements: guarded regions. These statements have the following syntax and meaning:

('|' separates the guarded statements)

when-statement:

when $b_1 : S_1 \mid \ldots \mid b_n : S_n$ **end**

meaning:

wait until at least one of the conditions b_i is true, then select one of these arbitrarily and execute the corresponding statement S_i.

cycle-statement:

cycle $b_1 : S_1 \mid \ldots \mid b_n : S_n$ **end**

meaning:

endless repetition of the corresponding **when**-statement.

2.4. Process execution

A process begins by executing its initial statement. This continues until the statement either terminates or waits in front of a guarded region for a condition to become true. Then the process is idle until another operation is started as the result of an external request. Note that this is the only possible way to continue, for only the process itself can access the variables, occurring in the conditions of a guarded region within its initial statement.

When this operation terminates or waits in front of a guarded region, the process will

(1) either begin yet another operation, i.e. honour an external request,
(2) or resume an earlier operation as result of a condition becoming true,
(3) or remain idle until one of the above situations occurs.

If the initial statement terminates, the process will still honour external requests. Globally, a process acts like a monitor but for implicit signalling operations, associated with guarded regions.

Brinch Hansen's processes never terminate. Hence Hoare's {p} Q {r} formalism could be trivially applied with r ≡ false. However, one can very well consider a program as being 'terminated' iff each of its processes

either (1) has terminated its initial statement

or (2) has not yet terminated its initial statement, and control is at a guarded region within the initial statement with all (constituent) guards false.

Now {p} Q {r} is valid iff for initial states satisfying p, upon 'termination' of Q, in the sense above, r holds for the possible resulting states.

Notice that if a process satisfies condition (1), it may very well be the case that some of its common procedures are being executed. Thus the question arises: if all processes satisfy condition (1) or (2), it is not possible that those processes which satisfy (2) are executing common procedures? The answer is no. A common procedure of one process being executed presupposes a call of that common procedure in another process, again not satisfying condition (2), which may occur in its turn in a common procedure body, etc.. The resulting chain of incarnations has a first call which necessarily occurs inside an initial statement neither satisfying condition (1) or (2). Hence the program is not 'terminated'.

We are working on formalizing this convention in a framework such as that of Lamport [5], in which properties of the location counter are expressible.

2.5. Nested external requests and recursive procedures

Program executions in which a suspended process (cf. 2.2) has to honour requests for computation to continue, are not considered correct executions. In particular recursive procedures are taboo in the DP language concept. Beside the ambiguity about this in Brinch Hansen's paper [4], the main reason for this convention is the following.

While a process is suspended, as result of an external request within it, no operations take place in it. Consequently, this process is not able to honour a request; especially requests to its own procedures; consult section 2.2 for a possible extension of DP in which such requests can be honoured.

2.6. Nondeterminism

Nondeterminism in DP is introduced:

(1) implicitly by the unpredictable order in which earlier operations are resumed, and

(2) explicitly by means of guarded statements:

(i) guarded region : also enables synchronization (cf. 2.3)

(ii) guarded commands: with syntax and meaning as follows: (as in 2.3, '|' separates the guarded statements)

if-statement:

$$\underline{\text{if}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}$$

meaning:

if some of the conditions are true, select one of them arbitrarily and execute the corresponding statement, otherwise abort the process.

do-statement:

$$\underline{\text{do}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}$$

meaning:

while some of the b_i's are true, select one of them arbitrarily and execute the corresponding statement.

Note that in contrast with a guarded region, a guarded command cannot delay an operation.

2.7. A few notational remarks

(1) There is a shorthand notation for an array of n identical processes within a program:
process name[n].
A standard function 'this' defines the identity of an individual process within this process-array. Likewise.functions as 'succ' and 'pred' for the successor and the predecessor of an individual process, if any, may be added.

(2) seq[n]T or array[n]T denote a sequence or array with at most n elements of type T.

(3) We use $[P_1 \parallel \ldots \parallel P_n]$ to denote the parallel execution of processes $P_1, \ldots, P_n$ $(n \geq 2)$, i.e., execution of the DP program that consists of the processes $P_1, \ldots, P_n$ $(n \geq 2)$.

(4) Concatenation of array (or sequence) a1 with array (or sequence) a2 is denoted as: a1^a2. (This is needed in the proof system).

Further details on the language concept DP may be found in [4].

3. SOME SIMPLE EXAMPLES

- to illustrate several language aspects -

3.1. One of the simplest examples of message passing in DP programs is the following.

```
process P1;
x : int
begin x := 0; call P2.pass(x) end

process P2;
y : int
proc pass (z : int) begin y := z end
begin y := 1 end
```

evidently, $y = 0$ is valid after termination of this program.

3.2. The next example illustrates the arbitrary order in which different outstanding requests to the same process are honoured.

Process 'soldier' with private variable 'salary' is requested simultaneously by an array of n processes (cf. 2.7). Honouring a request results in assigning the value the identity of the requesting process to 'salary'.

```
process soldier;
salary:int
proc change (z:int) begin salary := z end
begin salary := 0 end

process military_department[ n] ;
begin call soldier.change(this) end
```

After termination of this program the assertion:
$salary=1 \vee \ldots \vee salary=n$, will be valid. And this is indeed the strongest assertion possible !

3.3. Next, let us consider a program with synchronization by means of a when-statement. The single guard of this statement is initially made false and can only become true inside a procedure body in the same process. Consequently a request to the procedure is honoured before the process continues the execution of this when-statement.

Who believes that monkeys are not able to write nonsense in cooperation - even people can. In this program two monkeys are obliged to write "nonsense" for getting a banana. They can do this by typing the syllable 'HUM' or 'BUG', which in cooperation and depending on the typed order, may give 'HUMBUG'.

```
definition : mode syllable = seq[3]char.

process monkey[2];
proc eat(x) begin "eat x" end
begin call writer.type end

process writer;
b, c, d : bool; word : seq[2]syllable; message : seq[16]char
proc type if true : word := word^<HUM>; b := true; c := false;
                    d := true
            |true : word := word^<BUG>;
                    if c : d := false; b := true
                    |¬c : c := true
                    end
          end
begin b := c := d := false; message := word := <>;
      when b : if c∧d : call monkey[1].eat(banana);
                        call monkey[2].eat(banana)
                 |¬(c∧d): message := <no_bananas_today>
                 end
      end
end
```

Note: The boolean guard d is introduced to ensure termination of the program (b := true) without the possibility of incorrect feeding (d := false).

Unfortunately the writer of this program has fallen into monkeyism. For even though the monkeys succeed in typing 'HUMBUG', they cannot be sure of getting their desired bananas: process writer may resume the when-statement directly after the first honoured request is executed (that resulted in: word = <HUM>∧b = true∧c = false), and in that way stop the supply of bananas for today.
However, when the program terminates the assertion
word = <HUMBUG> ∨ message = <no_bananas_today>, holds.

3.4. This example concerns the nesting of procedure calls.

A process-array calculates n! when the input item is n, provided n is smaller than the length of this array. Item n is input from the user process through calculate-process [1], that outputs 1 if n=0, otherwise n-1 is passed to the successor, calculate-process [2]. The latter outputs 1 to its predecessor if n-1=0 and in the other case n-2 is passed to its successor, and so on.

Finally calculate-process [n+1] receives the value 0 and consequently outputs 1 to the user process if n=0, else to calculate-process[n]. In the latter case the remaining requests are finished successively by assigning the value of (n+1 - this) * (passed value) to one's predecessor in the sequence of nested requests.
Remember that the parameter passing mechanism is call-by-value-result.

```
Process calculate [m] ;
proc faculty (x:int # y:int) if x>0 : call calculate[succ].faculty(x-1,y);
                                      y := y*x
                             | x=0 : y := 1
                            end
begin skip end

process user;
value:int
begin call calculate [1] .faculty(value,value) end
```

3.5. For those familiar with CSP (Communicating Sequential Processes), a language concept by Hoare and related to DP [2,6,10], the former programs are expressed by:

(1) P_1 :: x := 0; P_2! x

P_2 :: y := 1; P_1?y

(2) soldier ≡ P_0 :: salary := 1; y := 1;*[P_1?y→ salary := y [] ...

...[] P_n?y→ salary := y]

P_i :: z := i; P_0!z , $1 \le i \le n$.

```
(3)  monkey[i] :: [true → writer!<HUM> [] true → writer! <BUG>];
                 *[writer?y → "eat y"]                        (i = 1,2)

     writer :: word := <>;
                [monkey[1]?x → word := word^x; monkey[2]?x; word := word^x
               [] monkey[2]?x → word := word^x; monkey[1]?x; word := word^x];
                [word = <HUMBUG> → monkey[1]!banana; monkey[2]!banana
               []¬word = <HUMBUG> → message := <no_bananas_today>]
```

Note that in this program process writer is indeed a fair judge.

```
(4)  user ≡_DEF P_0 :: value := n; P_1!value; P_1?value
     P_i :: [P_{i-1}?x → [x > 0 → P_{i+1}!(x-1); P_{i+1}?y; P_{i-1}!(x*y)
                         [] x = 0 → P_{i-1}!1
                          ]
          ],
     (1 ≤ i ≤ n).
```

4. THE PROOF SYSTEM

4.1. Introduction of process invariant, global invariant, auxiliary variables, bracketed sections and environment.

The absence of global variables in DP programs creates an ideal situation to prove properties about programs by proving properties of the processes within them in isolation, and then deducing the desired properties by comparison. Assertions in a separate proof of a process may contain only private variables and private auxiliary variables of this process, analogous to Owicki's system [7] and the CSP proof system [2]. An inevitable phenomenon of languages for parallel programs, however, is that the meaning of a single process is no longer independent of its context, in contrast with sequential programs in which, for instance, the meaning of a procedure (without global variables) can be defined in situ. Consequently, a proof system that captures deduction of properties about a program, but that contains no meta elements such as the cooperation test in the proofsystem for CSP or Owicki's interference freedom test, must provide these elements (i.e., context dependency of isolated proofs) implicitly. Obviously, this context dependency is expressed by predicates on values received at synchronization which took place when the process in question was waiting. This leads to the following definition.

<u>definition</u> 1: A process is at a <u>waiting point</u> (cf. 2.4):

(1) before a guarded region when all its guards evaluate to <u>false</u>,

(2) on termination of the initial statement,

(3) on completion of an honoured request.

For a proof system, this implies that rules concerning waiting points must provide assertions about the actual state of the process.
Since we disregard programs that may deadlock because of inter-process communication, the assertions in a proof of a process do not provide information about suspension of the process, but are concerned only with the values of the private variables and the parameters of the process. Consequently, the actual state of the process at a waiting point can be expressed by means of relations between, and values of, these private variables; this will be done in the, so called, <u>process invariant</u>.

Before we continue with the process invariant, first some remarks about <u>auxiliary variables</u>. The need for these variables is a familiar fact in proofs concerning parallel programs.

<u>definition</u> 2: AV = set of auxiliary variables, different from program variables and parameter incarnations;

Variables in AV do not affect the flow of control in a program and appear only in assignments of the form $x := t$, where $x \in AV$.
Each process has its own set of auxiliary variables.

A process invariant, denoted $PI_{<\text{process name}>}$ or, if the processes are numbered, $PI_{<\text{process index}>}$, may contain only private variables and private auxiliary variables.
Henceforth, in references to the process invariant in general, the invariant will be denoted by PI.
PI must hold at all waiting points in the process corresponding with PI and, as such, provide information about the actual state of this process (notice the analogy with monitor invariants).

However, it is not enough to add the PI's to the proof system. Consider the following example:

```
process P[2];
j[this] : int
begin j[this] := 1; call P_3.change end

process P_3;
x,i:int
proc change begin i:=i+1; x:=x+1 end
begin i:=1; x:=0 end
```

The number of times procedure change in process P_3 has been called after termination of the program cannot be deduced in a proof about P_3. Not even after the private auxiliary variables j[1], j[2] and i are added. This is a result of the fact that requests are honoured implicitly; only the requesting process can keep pace with its number of requests.

A solution is obtained by a well known strategy: introduction of a global invariant of the program. This invariant will be denoted by GI. GI catches the interaction between the processes. In the example above GI $\equiv$ $i=j[1]+j[2]+1$ will indeed provide the value of i on termination of the program.
For reasons that will become clear in section 4.3 (rules R6 and R13), GI may contain only auxiliary variables (i.e. variables in AV). This is no limitation because auxiliary variables can trivially simulate the behaviour of process variables.

At present however, GI is not valid at all places, which contradicts the property of being a global invariant, e.g., in the example above the assignments to j [this] and i are not executed simultaneously. Therefore we introduce program sections S in which GI need not hold: bracketed sections. They will be denoted by $< S >$.
Outside bracketed sections GI must be valid, and to ensure this assignments to variables free in GI are restricted to these sections.

Our first idea concerning the form of a bracketed section $<S>$ within the program $[P_1 \parallel \ldots \parallel P_n]$ was the following:

$S \equiv S_1$; <u>call</u> $P_j.pr(u_1,\ldots,u_r,\ v_1,\ldots,v_s)$; S_2 $\quad (1 \le j \le n)$

where pr denotes a procedure declared in P_j, and S_1 and S_2 do not contain guarded regions nor external requests; they usually contain assignments to auxiliary variables but may be empty. Because a procedure body may contain assignments to auxiliary variables of GI, all requests are cast within bracketed sections.

This definition is based on the fact that by the grain of interleaving that is used here, execution of a request is viewed as an elementary action within the requesting process.
The strategy to be used depends on the validity of GI at the beginning of a bracketed section: GI is indeed a global invariant of the program if we can prove that GI is valid again at the end of each bracketed section within the program.

However, with this definition of bracketed section our strategy fails, which will become clear by the next program.

```
process P1;
i : int
begin i := 0; <i := 1; call P2.pr2> end

process P2;
j2 : int
proc pr2 begin <j2 := j2 + 1; call P3.pr3> end
begin j2 := 0 end

process P3;
x, k : int
proc pr3 begin <k := k + 1; x := x + 1> end
begin k := 0; x := 0 end
```

i, j2 and k are auxiliary variables: i is used as counter of the requests to pr2 in process P_1, j2 keeps pace with the number of the requests to pr3 within the process P_3, and k and j2 are counters for the procedure body of pr3 and pr2 respectively. In order to prove $x = 1$ on termination of the program $[P_1 \parallel P_2 \parallel P_3]$, we need the global invariant $GI \equiv k = j2 \wedge j2 = i$.

Let us take a look at the bracketed section <j2 := j2 + 1; <u>call</u> P_3.pr3>

in process P_2. This section will always occur within another bracketed section in the program, because every request is inside a bracketed section. In this case, the enclosing section is <i := 1; <u>call</u> P_2.pr2> in process P_1. This situation is disastrous for our strategy: GI does not hold at the beginning of <j2 := j2 + 1; <u>call</u> P_3.pr3> because, by this time, i = 1 and j = 0.

Actually, the existence of nested bracketed sections is the cause of this. We eliminate this, by the introduction of countersections. The new definition is based on the fact that the validity of GI is in question only when parameter passing occurs, because the free variables in GI need be updated at these points. Inside the requesting process, there are at most two places at which parameter passing occurs (on account of call-by-value-result): at the beginning and at the end of the procedure execution.

Let pr be a procedure within P_j. Rearrange the assignments to auxiliary variables within the body so that pr has the following form:

<u>proc</u> pr(x_1, ..., x_r # y_1, ..., y_s) <u>begin</u> T_1; T; T_2 <u>end</u>

where T_1 and T_2 do not contain guarded regions nor external requests, and may be empty. Furthermore, in T no auxiliary variables free in GI appear outside bracketed sections. (The form need not be unique for pr!)

The next step is taken by associating the countersection T_1 with S_1 in the calling process and likewise for T_2 and S_2. The boundaries T_1 and T_2 (remember the former bracketed section definition) are indicated by the brackets ">" and "<" respectively, and the begin, respectively, end-brackets of the procedure body:

<u>proc</u> pr(x_1, ..., x_r#y_1, ..., y_s) <u>begin</u> T_1; > T <; T_2 <u>end</u>

Note that <S_1;T_1> and <T_2;S_2> are not bracketed sections according to our previous definition of S , with S ≡ S_1; <u>call</u> P_j.pr(u_1, ..., u_r, v_1, ..., v_s); S_2 , where:

<u>Definition</u> 3: A bracketed section within the DP program [P_1 ‖ ... ‖ P_n] is of the form <S_1; T_1>, <T_2; S_2>, or <S_1; <u>call</u> P_j.pr(u_1, ..., u_r, v_1, ..., v_s); S_2>, where:

(i) <S_1, and S_2> are taken from the construction
<S_1; <u>call</u> P_j.pr(u_1, ..., u_r , v_1, ..., v_s); S_2> in P_i $(1 \le i \le n)$,

(ii) T_1>, and <T_2 are taken from the construction

<u>proc</u> $pr(x_1, \dots, x_r \# y_1, \dots, y_s)$ <u>begin</u> T_1;> T <; T_2 <u>end</u>, in process $P_j (1 \le j \le n)$, and

(iii) S_1, S_2, T_1 and T_2 do not contain guarded regions nor external requests, and may be empty. In the last situation they may be indicated with <u>skip</u>.

No assignments to auxiliary variables free in GI may appear outside bracketed sections.

In order to enforce correct use of GI and the PI's in the formulae, by means of which the axioms and rules of the proof system are expressed, we introduce the (proof-theoretical) environment, E. Besides these invariants, E also contains the texts of the procedure bodies in the program, needed for application of R12, and the variables declared in the processes within the program.

<u>Definition</u> 4: The (proof theoretical) <u>environment</u> E of the program $[P_1 \parallel \dots \parallel P_n]$ has the following composition:

$$\begin{aligned} E \equiv {} & \langle\langle\langle \text{private variables} \in P_i\rangle \\ & |\langle pr_j(x_1, \dots, x_{in_j} \# y_1, \dots, y_{out_j}) \in P_i\rangle_{j=1}^{i_n}\rangle_{i=1}^{n} \\ & |\langle PI_i\rangle_{i=1}^{n} \mid GI\rangle, \end{aligned}$$

where:

(1) the procedures in process P_i are numbered from 1 to i_n,

(2) no free variable of PI_j is subject to change in P_k, $k \neq j$,

(3) no free variable of GI is subject to change outside a bracketed section.

Let each process in the program be numbered. We will fill the last "denotational holes" in the system by addition of the process index to the axioms and rules. In the notation, they appear directly after the statement in question, e.g., {p} S, i {q}. In this way, we ensure correct choice of the process invariant on application of the <u>when</u>-rule, R9, namely $PI_{<i>}$, and on application of the <u>call</u>-rule, R12.

As will be evident from the <u>when</u>-rule, we also need a formal system for expressing and deducing properties of the location counter, such as Lamport's at (S), in (S), after (S), cf. [5]. We are still working on this.

4.2. List of axioms and proof rules

Notes

1. Where the index of the process has no particular role in the axiom or rule, it has been omitted, to improve readability.

2. $E]\{p_1\}\ S_1\ \{q_1\}, \ldots, \{p_n\}\ S_n\ \{q_n\}$ is short for $E]\{p_1\}\ S_1\ \{q_1\}, \ldots, E]\{p_n\}\ S_n\ \{q_n\}$.

Axioms:

A1 assignment

$$E]\{p[t/x]\}\ x := t\ \{p\}$$

A2 skip

$$E]\{p\}\ \underline{\text{skip}}\ \{p\}$$

A3 invariance

$$E]\{p\}\ \underline{\text{call}}\ P_j.pr(u_1, \ldots, u_n, v_1, \ldots, v_m), i\{p\}$$

provided $\text{freevar}(p) \cap \{v_1, \ldots, v_m\} = \emptyset$, and no variables free in p are changed in process P_k, $k \neq i$.

Rules:

R1 begin-end

$$\frac{E]\{p\}\ S\ \{q\}}{E]\{p\}\ \underline{\text{begin}}\ S\ \underline{\text{end}}\ \{q\}}$$

where S denotes a statement.

R2 composition

$$\frac{E]\ \{p\}\ S_1\ \{q\},\ \{q\}\ S_2\ \{r\}}{E]\ \{p\}\ S_1;\ S_2\ \{r\}}$$

R3 consequence

$$\frac{p \rightarrow p_1,\ E]\ \{p_1\}\ S\ \{q_1\},\ q_1 \rightarrow q,}{E]\ \{p\}\ S\ \{q\}}$$

$$\frac{p \rightarrow p_1,\ \{p_1\}\ [P_1 \parallel \ldots \parallel P_n]\ \{q_1\},\ q_1 \rightarrow q}{\{p\}\ [P_1 \parallel \ldots \parallel P_n]\ \{q\}}$$

R4 conjunction

$$\frac{E]\ \{p\}\ S\ \{q\},\ \{p\}\ S\ \{r\}}{E]\ \{p\}\ S\ \{q \wedge r\}}$$

R5 disjunction

$$\frac{E]\ \{p\}\ S\ \{r\},\ \{q\}\ S\ \{r\}}{E]\ \{p \vee q\}\ S\ \{r\}}$$

R6 substitution

$$\frac{E]\ \{p\}\ S\ \{q\}}{E]\ \{p[z/x]\}\ S\ \{q\}}$$

provided $x \notin \mathrm{freevar}(E,S,q)$

$$\frac{\{p\}\ [P_1 \parallel \ldots \parallel P_n]\ \{q\}}{\{p[z/x]\}\ [P_1 \parallel \ldots \parallel P_n]\ \{q\}}\ ,$$

provided $x \notin \mathrm{freevar}(P_1, \ldots, P_n, q)$

R7 <u>if</u>

$$\frac{E]\ \{p \wedge b_1\}\ S_1\ \{q\}, \ldots, \{p \wedge b_n\}\ S_n\ \{q\},\ p \rightarrow \bigvee_{i=1}^{n} b_i}{E]\ \{p\}\ \underline{\text{if}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}\ \{q\}}$$

R8 <u>do</u>

$$\frac{E]\ \{p \wedge b_1\}\ S_1\ \{p\}, \ldots, \{p \wedge b_n\}\ S_n\ \{p\}}{E]\ \{p\}\ \underline{\text{do}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}\ \{p \wedge \neg \bigvee_{i=1}^{n} b_i\}}$$

R9 <u>when</u>

$$\frac{\begin{array}{l} E]\ \{p \wedge b_1\}\ S_1, i\ \{q\}, \ldots, \{p \wedge b_n\}\ S_n, i\ \{q\}, \\ p \wedge \neg \bigvee_{j=1}^{n} b_j \rightarrow PI_i, \\ \{b_1 \wedge PI_i \wedge at(S)\}\ S_1, i\ \{q\}, \ldots, \{b_n \wedge PI_i \wedge at(S)\}\ S_n, i\ \{q\} \end{array}}{E]\ \{p\}\ \underline{\text{when}}\ b_1 : S_1 \mid \ldots \mid b_n : S_n\ \underline{\text{end}}, i\ \{q\}}$$

where S in at(S) denotes the considered occurrence of the <u>when</u>-statement.

R9' <u>cycle</u>

$\underline{\text{cycle}}\ b_1{:}S_1 \mid \ldots \mid b_n{:}S_n\ \underline{\text{end}} \equiv \underline{\text{do}}\ \underline{\text{true}}{:}\underline{\text{when}}\ b_1{:}S_1 \mid \ldots \mid b_n{:}S_n\ \underline{\text{end}}\ \underline{\text{end}}$

R10 local variables

$$\frac{E]\ \{p\}\ S[z/x], i\ \{q\}}{E]\ \{p\}\ \underline{\text{begin}}\ \underline{\text{new}}\ x;\ S\ \underline{\text{end}}, i\ \{q\}}$$

provided $z \notin \mathrm{freevar}(E,S,p,q)$

R11 bracketed section

$$\frac{E]\ \{p \wedge GI\}\ S\ \{q \wedge GI\}}{E]\ \{p\}\ \langle S\rangle\ \{q\}}$$

R12 <u>call</u>

$$\frac{\begin{array}{l} E]\ \{p \wedge PI_j\}\ x_1[i] := u_1; \ldots; x_n[i] := u_n;\ T_1, j\ \{p_1 \wedge p_2 \wedge GI\}, \\ E]\ \{p_1\}\ T[x_1[i]/x_1, \ldots, x_n[i]/x_n, y_1[i]/y_1, \ldots, y_m[i]/y_m], j\ \{q_1\}, \\ E]\ \{q_1 \wedge p_2 \wedge GI\}\ T_2;\ v_1 := y_1[i]; \ldots; v_m := y_m[i], j\ \{q \wedge PI_j\} \end{array}}{E]\ \{p\}\ \underline{\text{call}}\ P_j.pr(u_1, \ldots, u_n, v_1, \ldots, v_m), i\ \{q\}}$$

where (i) $x_k[i]$, respectively, $y_k[i]$denote the fresh copies of x_k, respectively, y_k for the incarnation of procedure pr belonging to this request,

(ii) procedure pr in P_j is declared in E as follows:

$\underline{\text{proc}}\ pr(x_1, \ldots, x_n \# y_1, \ldots, y_m)\ \underline{\text{begin}}\ T_1; > T <; T_2\ \underline{\text{end}},$

(iii) free variables of p and q which do not occur in GI are not changed outside P_i; $\{p_1\}$ T[...] $\{q_1\}$ may not contain free variables subject to change in P_k, $k \neq j$; p_2 may not contain free variables subject to change in P_k, $k \neq i$.

R13 parallel composition

$$\frac{<E]\ \{p_i\}\ \mathrm{Init}_i,\ i\ \{PI_i\}>_{i=1}^{n}}{\{\bigwedge_{i=1}^{n} p_i \wedge GI\}\ [P_1 \parallel \ldots \parallel P_n]\ \{\bigwedge_{i=1}^{n} PI_i \wedge GI\}}$$

where (i) Init_i denotes the (annotated) initial statement of P_i,

(ii) auxiliary variables and bracketed sections are contained in E and Init_i.

R14 AV

$$\frac{E]\ \{p\}\ S'\ \{q\}}{E]\ \{p\}\ S\ \{q\}}$$

provided free-variables(q) $\cap$ AV = $\emptyset$, and S is obtained from S' by deletions of all assignments to variables of AV, as defined in definition 2 of section 4.1.

In addition axioms and rules formalizing location properties are needed (for at(S), in(S), after(S), cf. [5]). These are not provided in the present version, but are needed in the formulation of R9, and the application of R12 (cf. the correctness proof of the monkey-banana example in section 5.3).

4.3. Justification of the proof system

Justification of E, PI, GI and the introduction of statement-index pairs in a formula is given in section 4.1 and will not be discussed here. The axioms A1 and A2 and the rules R2 to R5 are well-known and do not need explanation. Rule R1 is obvious. Rule R6 is of importance because of the elimination of auxiliary variables from pre-assertions. The axiom A3 will be explained directly after the discussion of rule R12.

(R7) According to the definition of the _if_-statement in section 2.6, the program is aborted when all guards evaluate to false on execution of this statement. In this case, $p \wedge \bigvee_{i=1}^{n} b_i$ does not hold and the statement cannot be proved with the proposed rule, as should be. Besides this, R7 is the usual rule of alternative command.

(R8) R8 is the ordinary rule of the repetitive command.

(R9) Rule R9 is of great interest because the <u>when</u>-statement may act as a waiting point: when the guards of this statement evaluate to <u>false</u>, the process may resume an earlier operation or start a new one by honouring a request. In the discussion in section 4.1, we demand that at this moment PI_i holds, which is expressed by $p \wedge \neg \overset{n}{\underset{j=1}{W}} b_j \rightarrow PI_i$ in R9.

Whenever the statement is resumed again, anything might have happened to the variables in assertion p and, consequently, p need not hold. However, now one of the b_i's is true and we just passed a waiting point, otherwise control in P_i could not have switched to this <u>when</u>-statement. Therefore, PI_i holds and we have arrived at a situation equivalent to the alternative command with precondition $PI_i \wedge at(S)$, where S denotes this particular occurrence of the when-statement, at(S) expresses that flow of control reached S, and hence, $PI_i \wedge at(S)$ "specializes to that part of PI_i (using modus ponens) applicable to S" (cf. [5]). Thus, in order to prove that q holds after execution of the when-statement, at least

$$E]\ \{b_1 \wedge PI_i \wedge at(S)\}\ S_1\ \{q\},\ \ldots,\ \{b_n \wedge PI_i \wedge at(S)\}\ S_n\ \{q\}$$

has to be proved (cf. R8).

In this case, validity of the first part of the premiss in R6:

$$E]\ \{p \wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p \wedge b_n\}\ S_n\ \{q\}$$

is easily checked, because $p \wedge b_j$ = <u>false</u>, for all j.

If the <u>when</u>-statement is not a waiting point, then at least one b_j is <u>true</u> on arrival. The statement is now equivalent to the corresponding alternative command and, consequently,

$E]\ \{p \wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p \wedge b_n\}\ S_n\ \{q\}$ has to be proved.

But in this case R9 forces also the proofs of $p \wedge \neg \overset{n}{\underset{j=1}{W}} b_j \rightarrow PI_i$, which is trivial because $p \wedge \neg \overset{n}{\underset{j=1}{W}} b_j$ = <u>false</u>, and of the third premiss:

$$E]\ \{b_1 \wedge PI_i \wedge at(S)\}\ S_1\ \{q\},\ \ldots,\ \{b_n \wedge PI_i \wedge at(S)\}\ S_n\ \{q\}.$$

In practice, by appropriate manipulation of the auxiliary variables, i.e., by encoding that the disjunction of the guards of a <u>when</u>-statement did or did not hold, immediately prior to entry, one could obtain that $(p \wedge W b_j \rightarrow \neg(PI_i \wedge at(S)))$ holds in that context.

(R10) This is a familiar rule (cf. [3], ch. 6).

(R11) This rule should be clear from section 4.1.

(R12) R12, the call-rule is relatively simple in comparison with the usual procedure call-rules, because DP has no recursive procedures. The appearance of PI_j in the premiss has been explained in section 4.1 , while discussing the arrival of control in P_j at a waiting point (P_j is ready to honour the request). The appearence of GI and its relation to the sections T_1, T_2 and the assertions p and q has been explained too in 4.1.
We have chosen this form for R12, because now the premiss $\{p_1\}$ T[...] $\{q_1\}$ may be copied from the proof of the procedure body in P_j. Because each request is within a bracketed section, the application of R12 is always combined with the application of R11. This implies that the assertions p and q may contain, besides variables and auxiliary variables private to P_i, also auxiliary variables in GI. Observe that the actual input parameters need not be distinct from the actual output parameters.
Intermediate assertion p_2 is needed because q may contain free variables from P_i different from the output parameters $v_1, \ldots, v_m$, and these are not allowed to occur freely in $\{p_1\}$ T[...] $\{q_1\}$. The precise formulation of the conditions upon the free variables has to do with the wish to apply the technique of "freezing the variables" in correctness proofs, see ch. 6.

(A3) In the present version, A3 is not needed because of the appearance of p_2 in R12. Since A3 is needed in a projected extension of the formalism and its innocent use in the examples can be easily eliminated in favour of R12 in its present formulation, A3 is still included.

(R13) As a result of the termination convention (cf. 2.4) and the definition of bracketed sections (cf. 4.1) program-control is not within a bracketed section at the beginning nor at the end of execution of the program. By R11 and the restriction that no variables free in GI may be subject to change outside a bracketed section (cf. 4.1), GI is valid at the beginning and end of the program. Further, the consistent usage of E in the premisses of R13 implies validity of the conclusion, provided that these premisses were valid.

(R14) Variables in AV do not affect the flow of control during execution of the program and have no influence on the values of the variables within the processes (by definition, cf. 4.1). Consequently, when no variables of AV appear in the post-assertion, we may delete all assignments to variables of AV in the proved program and claim the same post-assertion for the new program with respect to the original pre-assertion.

5. PROOFS OF THE EXAMPLES IN SECTION 3

In order to make proofs readable, proof outlines will be given in which the processes $P_1, \dots, P_n$ (modified by addition of auxiliary variables and bracketed sections to $P_1', \dots, P_n'$) are annotated with assertions at appropriate places. Because environment E is clear from the text, expressing amongst other things that the PI's and GI are chosen once, we omit E in the proofs. The index, used in the axioms and proofrules to indicate the process in in which the considered occurrence of a statement is situated, is likewise omitted. Where the rules and axioms, used for the proof of {p} S {q}, can be understood easily, we shall not mention them. Further, $\{p_1\}\{p_2\}$ occurring as adjacent assertions in a proof outline denote use of the consequence rule R3, with $p_1 \rightarrow p_2$. Insofar possible x_k, respectively, y_k is written instead of $x_k[i]$ and $y_k[i]$ in the use of rule R12.

5.1. Proof of example 3.1.

We must prove: {true} $[P_1 \parallel P_2]$ $\{y = 0\}$

For this purpose, we choose the following invariants and proof outlines:

$PI_1 \equiv h_1 = x \wedge x = 0,$

$PI_2 \equiv y = \text{last}(h_2),$

$GI \equiv h_1 = 0 \rightarrow h_1 = \text{last}(h_2),$

```
process P'_1;
x, h_1 : int
{h_1 = -1} begin {h_1 = -1} x := 0; {h_1 = -1 ∧ x = 0}
                 <h_1 := 0; {PI_1} call P'_2.pass(x)>
                 {PI_1}
           end   {PI_1}

process P'_2;
y : int; h_2 : seq[2]int
proc pass(z : int) {PI_2} begin {PI_2} h_2 := h_2^<z>;>
                                {last(h_2) = z} y := z <{PI_2}
                          end   {PI_2}
{h_2 = <1>} begin {h_2 = <1>} y := 1 {PI_2} end {PI_2}
```

(1) Verification of all assertions is straightforward (with A1, A2, R2, R3 and R4), except for those belonging to the bracketed section in process P_1'.

To prove:

$\{h_1 = -1 \wedge x = 0\}$ $<h_1 := 0;$ $\underline{\text{call}}$ $P_2'.pass(x)>$ $\{PI_1\}$ ... (i)

By rule R11, this reduces to proving

$\{h_1 = -1 \wedge x = 0 \wedge GI\}$ $h_1 := 0;$ $\underline{\text{call}}$ $P_2'.pass(x)$ $\{PI_1 \wedge GI\}$,

and, hence, by A1 to

$\{PI_1\}$ $\underline{\text{call}}$ $P_2'.pass(x)$ $\{PI_1 \wedge GI\}$ ... (ii)

We will prove this by the $\underline{\text{call}}$ rule (R12);
the premiss needed is:

$\{PI_1 \wedge PI_2\}$ $z := x;$ $h_2 := h_2\hat{}<z>$ $\{z = 0 \wedge last(h_2) = z \wedge PI_1 \wedge GI\}$,
$\{z = 0 \wedge last(h_2) = z\}$ $y := z$ $\{y = 0 \wedge PI_2\}$,
$\{y = 0 \wedge PI_2 \wedge PI_1 \wedge GI\}$ $\underline{\text{skip}}$ $\{PI_1 \wedge GI \wedge PI_2\}$,

and follows by application of A1, A2, R2 and R3.
By R12, we now conclude (ii) and, hence, we proved (i).

(2) From the proof outlines we conclude

$\{h_1 = -1\}$ $Init_1$ $\{PI_1\}$, and
$\{h_2 = <1>\}$ $Init_2$ $\{PI_2\}$.

Applying R13 and R3, we get

$\{h_1 = -1 \wedge h_2 = <1> \wedge GI\}$ $[P_1' \parallel P_2']$ $\{PI_1 \wedge PI_2 \wedge GI\}$ $\{y = 0\}$.

(3) Because $y \notin AV$, we may apply rule R14 to conclude

$\{h_1 = -1 \wedge h_2 = <1> \wedge GI\}$ $[P_1 \parallel P_2]$ $\{y = 0\}$.

Substituting -1 for h_1 and <1> for h_2, we finally derive by repeated application of R6:

$\{\underline{\text{true}}\}$ $[P_1 \parallel P_2]$ $\{y = 0\}$.

Q.E.D.

5.2. Proof of example 3.2.

We have to prove:

$\{\underline{\text{true}}\}$ [soldier $\parallel$ military-department[n]] $\{salary = 1 \vee \ldots \vee salary = n\}$

We associate in the proof outlines with each process a number. The processes in the array are indicated with their array-index. Process soldier gets number 0.

The proof is along the same line as the previous example. We choose the following invariants and proof outlines:

$PI_0 \equiv \text{salary} = \text{last}(h),$

$PI_i \equiv h_1[i] = 1, \text{ for } 1 \le i \le n,$

$$GI \equiv \bigwedge_{i=1}^{n} (h_1[i] = 1 \rightarrow i \in h) \wedge \text{length}(h) = 1 + \sum_{i=1}^{n} h_1[i],$$

$\underline{\text{process}}$ military-department[n]';

$h_1[\text{this}]$: int

$\{h_1[\text{this}] = 0\}$ $\underline{\text{begin}}$ $\{h_1[\text{this}] = 0\}< h_1[\text{this}] := 1;$

$\qquad\qquad\qquad$ $\underline{\text{call}}$ soldier'.change(this) $>\{PI_{\text{this}}\}$

$\qquad\qquad$ $\underline{\text{end}}$ $\{PI_{\text{this}}\}$

$\underline{\text{process}}$ soldier';

salary : int; h : $\underline{\text{seq}}$[n+1]int

$\underline{\text{proc}}$ change(z:int) $\{PI_0\}$ $\underline{\text{begin}}$ $\{PI_0\}$ h := h^<z>;>

$\qquad\qquad\qquad\qquad$ {last(h) = z} salary := z<$\{PI_0\}$

$\qquad\qquad\qquad$ $\underline{\text{end}}$ $\{PI_0\}$

{h = <1>} $\underline{\text{begin}}$ {h = <1>} salary := 1 $\{PI_0\}$ $\underline{\text{end}}$ $\{PI_0\}$

(1) As in 5.1., we only verify the assertions belonging to bracketed sections, and this time only those occurring within process-array military-department[n]'. We prove:

$\{h_1[\text{this}] = 0\}< h_1[\text{this}] := 1;$ $\{PI_{\text{this}}\}$ $\underline{\text{call}}$ soldier'.change(this)> $\{PI_{\text{this}}\}$

Assume this = i_0, then by R11 this reduces to proving:

$\{h_1[i_0] = 0 \wedge GI\}$ $h_1[i_0] := 1;$ $\{PI_{i_0}\}$ $\underline{\text{call}}$ soldier'.change(i_0) $\{PI_{i_0} \wedge GI\}$.

Hence, by A1 and the definitions of PI_{i_0} and GI, we have to prove:

$$\{PI_{i_0} \wedge \bigwedge_{\substack{i=1\\ i\neq i_0}}^{n} \text{and}(h_2[i] = 1 \rightarrow i \in h) \wedge \text{length}(h) = 1 + \sum_{\substack{i=1\\ i\neq i_0}}^{n} \text{and}\, h_1[i]\}$$

$\underline{\text{call}}$ soldier'.change(i_0)

$\{PI_{i_0} \wedge GI\}$.

By R12, this follows from premiss

$$\{PI_{i_0} \wedge \bigwedge_{\substack{i=1\\ i\neq i_0}}^{n} \text{and}(h_1[i] = 1 \rightarrow i \in h) \wedge \text{length}(h) = 1 + \sum_{\substack{i=1\\ i\neq i_0}}^{n} \text{and}\, h_1[i] \wedge PI_0\}$$

z := i_0; h := h^<z>

$\{last(h) = z \wedge PI_{i_0} \wedge \bigwedge_{i=1}^{n} (h_1[i]= 1 \rightarrow i \in h) \wedge length(h) = 1 + \sum_{i=1}^{n} h_1[i] \wedge z = i_0\}$

$\{last(h) = z \wedge PI_{i_0} \wedge GI\}$,

$\{last(h) = z\}$ salary := z $\{PI_0\}$,

$\{PI_0 \wedge PI_{i_0} \wedge GI\}$ skip $\{PI_{i_0} \wedge GI \wedge PI_0\}$,

which formulae are all valid by the definitions and A1, A2 and R2.

(2) From the proof outlines, we conclude validity of

$\{h = <1>\}$ $Init_0$ $\{PI_0\}$, and

$\{h_1[i] = 0\}$ $Init_i$ $\{PI_i\}$, $1 \le i \le n$.

Next we apply rules R13 and R3, resulting in

$\{h = <1> \wedge \bigwedge_{i=1}^{n} h_1[i] = 0 \wedge GI\}$ $[P'_0 \parallel \ldots \parallel P'_n]$

$\{\bigwedge_{i=0}^{n} PI_i \wedge GI\}$ $\{salary = 1 \vee \ldots \vee salary = n\}$.

(3) Analogous to part 3 of section 5.1., application of R14 and R6 to the formula above finishes the proof of

{<u>true</u>} $[P_0 \parallel \ldots \parallel P_n]$ $\{salary = 1 \vee \ldots \vee salary = n\}$, i.e., of

{<u>true</u>} [soldier ‖ military_department[n]] $\{salary = 1 \vee \ldots \vee salary = n\}$

(by the index convention at the beginning of this section).

Q.E.D.

5.3. <u>Proof of example 3.3.</u>

Let the programs be numbered as follows:

P_1 = monkey[1], P_2 = monkey[2], P_3 = writer.

To prove:

{<u>true</u>} $[P_1 \parallel P_2 \parallel P_3]$ {word = <HUMBUG> ∨ message = <no_bananas_today>}.

For this purpose we choose the following invariants and proof outlines:

PI_{this} ≡ h[this] = 1, for this = 1,2,

PI_3 ≡ (b ∧ d → HUM ∈ word) ∧ (c → last(word) = BUG) ∧
∧ length(word) = h_3 ∧ (loc = 1 → HUM, BUG ∈ word ∨
∨ message = <no_bananas_today>)
≡ PI'_3 ∧ length(word) = h_3,

GI ≡ h_3 = h[1] + h[2].

```
process monkey[2]';
proc eat(x) {$PI_{this}$} begin
            {$PI_{this}$} "eat x" {$PI_{this}$}
             end {$PI_{this}$}
{h[this] = 0} begin
             {h[this] = 0} <h[this] := 1;
                              {$PI_{this}$} call writer'.type> {$PI_{this}$}
              end {$PI_{this}$}

process writer';
b, c, d : bool; word : seq[2] syllable; message : seq[16]char;
proc type
{$PI_3$} begin {$PI_3$} $h_3$ := $h_3$ + 1> {$PI_3'$ ∧ length(word) = $h_3$ - 1}
           if true : {$PI_3'$ ∧ length(word) = $h_3$ - 1}
                     word := word^<HUM> {$PI_3$ ∧ last(word) = HUM}
                     b := true; c := false; d := true {$PI_3$}
           | true : {$PI_3'$ ∧ length(word) = $h_3$ - 1}
                     word := word^<BUG> {$PI_3$ ∧ last(word) = BUG}
                     if c : {$PI_3$ ∧ last(word) = BUG ∧ c}
                            d := false; b := true {$PI_3$}
                     | ¬c : {$PI_3$ ∧ last(word) = BUG ∧¬c}
                            c := true {$PI_3$}
                     end {$PI_3$}
           end {$PI_3$}
       <end {$PI_3$}
{$h_3$ = loc = 0} begin
    {$h_3$ = loc = 0} b := c := d := false;
                    message := word := <> {$PI_3$ ∧¬b}
                 when b : {b ∧ $PI_3$}
                    if c ∧ d : {b ∧ c ∧ d ∧ $PI_3$} {HUM, BUG ∈ word ∧ $PI_3$}
                               loc := 1; {$PI_3$ ∧ loc = 1}
                               <call monkey[1]'.eat(banana)>;
                               <call monkey[2]'.eat(banana)>
                               {$PI_3$ ∧ loc = 1}
                    ¬(c ∧ d) : {$PI_3$ ∧ ¬(c ∧ d)}
                               message := <no_bananas_today>;
                               {$PI_3$ ∧ message = <no_bananas_today>}
                               loc := 1 {$PI_3$ ∧ loc = 1}
                    end {$PI_3$ ∧ loc = 1}
                 end {$PI_3$ ∧ loc = 1}
end {$PI_3$ ∧ loc = 1}
```

1) The bracketed sections in P_3' are verified easily. First we check the annotated bracketed sections in process array monkey', and then the annotated when-statement.

a) We have to prove:

$\{h[\text{this}] = 0\} < h[\text{this}] := 1;$ call writer'.type $> \{PI_{\text{this}}\}$.

By R11, this reduces to

$\{h[\text{this}] = 0 \wedge GI\}\ h[\text{this}] := 1;$ call writer'.type $\{PI_{\text{this}} \wedge GI\}$.

Hence, by application of A1, we have to prove:

$\{PI_{\text{this}} \wedge h_3 = h[1] + h[2] - 1\}$ call writer'.type $\{PI_{\text{this}} \wedge GI\}$.

which, by R12, reduces to proving (assume this = i_0):

$\{PI_{i_0} \wedge h_3 = h[1] + h[2] - 1 \wedge PI_3\}\ h_3 := h_3 + 1$
$\{PI_3' \wedge \text{length(word)} = h_3 - 1 \wedge PI_{i_0} \wedge GI\}$,

$\{PI_3' \wedge \text{length(word)} = h_3 - 1\}$ if ... end $\{PI_3\}$
(cf. the proof outline of process P_3'),

$\{PI_3 \wedge PI_{i_0} \wedge GI\}$ skip $\{PI_3 \wedge PI_{i_0} \wedge GI\}$,

which all follow from the definitions, the proof outline of P_3, A1 and A2.

b) to prove: $\{PI_3 \wedge \neg b\}$ when b : if ... end end $\{PI_3 \wedge \text{loc} = 1\}$
For this when-statement in P_3',

$PI_3 \wedge \neg b \wedge b \rightarrow$ false holds,

Which by A1, A3 and R7 validates

$\{PI_3 \wedge \neg b \wedge b\}$ if ... end $\{PI_3 \wedge \text{loc} = 1\}$.

Furthermore, $PI_3 \wedge \neg b \wedge \neg b \rightarrow PI_3$.
The proof concerning the rest of the premiss for this application of R9 is outlined in the proof outline of P_3'.
By R9, we conclude $\{PI_3 \wedge \neg b\}$ when b : if ... end end $\{PI_3 \wedge \text{loc} = 1\}$.

2) Thus we proved

$\{h[i] = 0\}\ \text{Init}_i\ \{PI_i\}$ for i = 1,2 and
$\{h_3 = \text{loc} = 0\}\ \text{Init}_3\ \{PI_3 \wedge \text{loc} = 1\}$.

Application of R13, realizing that the location counter loc is used to encode after $(\text{Init}_{\text{writer}})$, that $\{p\}\ S\ \{\text{after}(S)\}$ is an axiom - cf. [5] - and application of R3 and R4 results in

$\{h[1] = h[2] = h_3 = \text{loc} = 0 \wedge GI\}[P_1' \parallel P_2' \parallel P_3']\{PI_1 \wedge PI_2 \wedge PI_3 \wedge GI \wedge \text{loc} = 1\}$
$\{\text{word} = \text{<HUMBUG>} \vee \text{message} = \text{<no\_bananas\_today>}\}$.

Now, analogous to part (3) in sections 5.1. and 5.2., succesive application of R14 and R6 (with 0 substituted for h[1], h[2], h_3 and loc) finishes the proof of

$\{\underline{\text{true}}\}[P_1 \parallel P_2 \parallel P_3]\{\text{word} = \text{<HUMBUG>} \vee \text{message} = \text{<no\_bananas\_today>}\}$

Q.E.D.

5.4. Proof of example 3.4.

In order to prove

$\{\text{value} = n \wedge 0 \le n < m\}[\text{user} \parallel \text{calculate}[1] \parallel \ldots \parallel \text{calculate}[m]]\{\text{value} = n!\}$

we choose the following invariants and proof outlines:

$PI_{\text{user}} \underset{\text{def}}{=} PI_0 \equiv \text{value} = n!$,

$PI_{\text{calculate[this]}} \underset{\text{def}}{=} PI_{\text{this}} \equiv \underline{\text{true}}$, for this = 1, ..., m,

$GI \equiv \underline{\text{true}}$,

```
process user';
value : int
{value = n ∧ 0 ≤ n < m} begin
            {value = n ∧ 0 ≤ n < m}<call calculate[1]'.
                                    faculty(value,value)>{PI_0}
                end {PI_0}

process calculate[m]';
proc faculty(x : int # y : int)
  {x = a ∧ 0 ≤ a ≤ m - this} begin>
     {x = a ∧ 0 ≤ a ≤ m - this} if x > 0 : {x = a ∧ 0 < a ≤ m - this}
                                        <call calculate[succ]'.
                                          faculty(x-1, y)>
                                        {y = (a-1)! ∧ x = a} y := y * x
                                        {y = a!}
                                 | x = 0 : {x = a ∧ a = 0} y := 1 {y = a!}
                                end {y = a!}
                            <end {y = a!}
{true} begin skip end {true}
```

Next, the annotated bracketed sections within the processes will be proved; the other annotated statements are easily verified using A1, A2 and R7.

We want to prove:

$\{value = n \wedge 0 \le n < m\}$

<call calculate[1]'.faculty(value,value)>

$\{value = n!\}$

i.e. validity of the bracketed section in process user.

The request call calculate[1]'.faculty(value,value) leads to the request call calculate[2]'.faculty(value-1, y) in process calculate[1]', etc., until the request call calculate[n+1]'.faculty in process calculate[n]' ends the iteration. The processes requested in this iteration are defined, because $n+1 \le m$.

a) Firstly, we have to prove this last request, i.e.,

$\{x = 1 \wedge 1 \le n+1 \le m\}$<call calculate[n+1]'.faculty(x-1, y)>$\{y = 0! \wedge x = 1\}$.

By R11 and the definition of GI, this leads to proving

$\{x = 1 \wedge 1 \le n+1 \le m\}$ call calculate[n+1]'.faculty(x-1, y) $\{y = 0! \wedge x = 1\}$.

For this, we must prove (R12)

```
{x = 1 ∧ 1 ≤ n+1 ≤ m} x[n] := x-1 {x[n] = 0 ∧ x = 1},
{x[n] = 0}
     if x[n] > 0 : {false}
          <call calculate[n+1]'.faculty(x[n] - 1, y[n])>
          {false} y[n] := y[n] * x[n]{false}{y[n] = o!}
      | x[n] = 0 : {x[n] = 0}
           y[n] :=1 {y[n] = 0!}
    end {y[n] = 0!}
{y[n] = 0! ∧ x = 1} y := y[n]{y = 0! ∧ x = 1},
```

which are valid by A1, A3, R11 and R7, the if-rule.

b) Secondly, assuming call calculate[n+1 - (j-1)]'.faculty(j-1, y) in calculate[n+1 - j]' to be correct for $1 \le j < n$, we must prove as induction step call calculate[n+1 - j]'.faculty(j, y) in calculate[n-j]' to be correct.

Hence we have to prove:

$\{x = j+1 \wedge 1 \le j < n \wedge m > n\}$<call calculate[n+1 - j]'.faculty(x-1, y)>

$\{y = j! \wedge x = j+1\}$

We have: $\{x = j+1 \wedge 1 \le j < n \wedge m > n\}$ x[n-j] := x-1

$\{x[n-j] = j \wedge 1 \le j < n \wedge m > n \wedge x = j+1\}$,

```
{$x[n-j] = j \wedge 1 \le j < n \wedge m > n$}
 if x[n-j] > 0 : {$x[n-j] = (j-1) + 1 \wedge 1 \le j < n \wedge m > n$}
                  <call calculate[(n+1) - (j-1)]'.faculty(x[n-j] - 1, y[n-j])>
                   (this call is correct for $1 < (n+1) - (j-1) \le n+1 \le m$)
                  {$x[n-j] = (j-1) + 1 \wedge y = (j-1)!$}
                   (by induction hypothesis)
                  y[n-j] := y[n-j] * x[n-j]
                  {$y[n-j] = j!$}
  | x[n-j] = 0 : {false} ....
                  {false}{$y[n-j] = j!$}
end {$y[n-j] = j!$},
{$y[n-j] = j! \wedge x = j+1$} y := y[n-j]{$y = j! \wedge x = j+1$},
```

which by R12 yields the result.

c) Thirdly, from a) and b) follows by induction

{$x = n \wedge 0 \le n < m$}
 <call calculate[2]'.faculty(x-1, y)>{$x = n \wedge y = (x-1)!$}

in the proof outline of calculate[1]'.

By reasoning as above, we obtain

{$\text{value} = n \wedge 0 \le n < m$}
 <call calculate[1]'.faculty(value,value)>{$\text{value} = n!$}.

2) By the proof outlines, we deduced

{$\text{value} = n \wedge 0 \le n < m$} Init_0 {PI_0}, and
 {true} Init_i {PI_i}, for $i = 1, \ldots, m$.

Application of R13 yields

{$\text{value} = n \wedge 0 \le n < m$}[$P'_0 \parallel \ldots \parallel P'_m$]{$\text{value} = n!$},

by the definitions of PI_i and GI.

Finally, rule R14 delivers the result:

{$\text{value} = n \wedge 0 \le n < m$}[$P_0 \parallel \ldots \parallel P_m$]{$\text{value} = n!$}, i.e.,
{$\text{value} = n \wedge 0 \le n < m$}[user $\parallel$ calculate[1] $\parallel \ldots \parallel$ calculate[m]]{$\text{value} = n!$}

Q.E.D.

The proof given aboven reflects that in the DP processes concerned no synchronization takes place, which differs essentially from the kind encountered when considering procedure calls in sequential programming. Yet one might wish to make the DP-synchronization explicit. This can be done as follows:

Rename process user as calculate[0]. Then the functioning of the network can be visualized as below after call calculate[1].faculty(n, value) occurs:

$$\begin{tikzcd}
\text{calculate}[0] \arrow[r, leftrightarrow, "\overset{n}{\rightarrow}", "\underset{n!}{\leftarrow}"'] & \text{calculate}[1] \arrow[r, leftrightarrow, "\overset{n-1}{\rightarrow}", "\underset{(n-1)!}{\leftarrow}"'] & \ldots. \arrow[r, leftrightarrow, "\overset{1}{\rightarrow}", "\underset{1}{\leftarrow}"'] & \text{calculate}[n] \arrow[r, leftrightarrow, "\overset{0}{\rightarrow}", "\underset{1}{\leftarrow}"'] & \text{calculate}[n+1]
\end{tikzcd}$$

Introduce auxiliary variables a[j, j+1], a[j+1, j] for j = 0, ..., n, by assumption initialized to 0. Replace <call calculate[j+1]'.faculty(n-1, y)> in calculate[j]' by <a[j, j+1] := n-j; call calculate[j+1]'.faculty(n-1, y)>, j = 0, ..., n, and add a[j, j-1] := y as last instruction to the body of faculty in calculate[j]', j = 1, ..., n+1.
Now GI ≡ $\bigwedge_{j=0}^{n}(a[j+1,\ j] \neq 0 \rightarrow a[j+1,\ j] = a[j,\ j+1]!)$ expresses the functioning of the network.

Also, one could introduce auxiliary variables recieve[j], j = 0, ..., n (by assumption initialized to 1), and replace

<a[j, j+1] := n-j; call calculate[j+1]'.faculty(n-1, y)> by
<a[j, j+1] := n-j; call calculate[j+1]'.faculty(n-1, y); receive[j] := y>.

Then PI_j ≡ recieve[j] = a[j, j+1]! expresses the I/O-behaviour of call calculate[j+1]'.faculty(n-j, y) in process calculate[j]'.
Obviously, superposing all this on top of the proof sketched above, amounts to adding a chain of trivialities.

Finally, in an alternative version of this program, using dynamic process creation for execution of faculty(x, y), the proof-theoretical environment of $\text{calculate}_{\text{this}}$ would be based upon formalization of the functioning of the channels

$$\begin{tikzcd}
\text{calculate}_{\text{pred}} \arrow[r, leftrightarrow, "\overset{n-\text{pred}}{\rightarrow}", "\underset{(n-\text{pred})!}{\leftarrow}"'] & \text{calculate}_{\text{this}} \arrow[r, leftrightarrow, "\overset{n-\text{this}}{\rightarrow}", "\underset{(n-\text{this})!}{\leftarrow}"'] & \text{calculate}_{\text{succ}}
\end{tikzcd}$$

by means of:

$$\begin{aligned} EI_{\text{this}} \equiv\ & (a_{\text{this,pred}} \neq 0 \rightarrow a_{\text{this,pred}} = (a_{\text{pred,this}})!)\ \wedge \\ & (a_{\text{succ,this}} \neq 0 \rightarrow a_{\text{succ,this}} = (a_{\text{this,succ}})!), \end{aligned}$$

in analogy with the above.

Again we expect the correctness argument to amount to little more than superposition of a chain of trivialities on top of the correctness proof for the well-known sequential recursive procedure faculty(n, y), from which this exercise in distributed programming originates anyhow.

6. CORRECTNESS OF A DISTRIBUTED IMPLEMENTATION OF A PRIORITY QUEUE

In this section we give a proof for the following sorting program, which was first described by Brinch Hansen in [4].

```
process sort[n];
here : seq[2]int; rest, temp : int
proc put(c:int)when here.length < 2 : here.put(c) end
proc get( #v:int)when here.length = 1 : here.get(v) end
begin here := []; rest := 0;
     cycle
        here.length = 2:
             if here[1] ≤ here[2] : temp := here[2]; here := [here[1]]
              | here[1] > here[2] : temp := here[1]; here := [here[2]]
             end;
             call sort[succ].put(temp); rest := rest + 1
       | here.length = 0 ∧ rest > 0:
             call sort[succ].get(temp); rest := rest - 1;
             here := [temp]
end
```

Brinch Hansen claims the program could be used as a priority scheduling queue as well, but when trying to prove it, we discovered an error. This is mended by substituting the guard here.length < 2 in proc put by here.length = = 1 ∨ (here.length = 0 ∧ rest = 0).

In the following proof outline kept and away are auxiliary variables, denoting the bag of elements in the current process and of those sent to the following process, respectively.

For the union of bags we use the symbol ⫴. The notation con(here), in the assertions, stands for the bag of contents of here.

```
process sort[n]';
here : seq[2]int; rest, temp : int

proc put(c : int)
{PI ∧ GI} begin {PI ∧ GI} > {PI}
              when here.length = 1 ∨ (here.length = 0 ∧ rest = 0):
               {PI ∧ (here.length = 1 ∨ (here.length = 0 ∧ rest = 0))}
                         here.put(c) {post p}
              end; {post p}
```

```
                <{GI ∧ postp}
                 kept  := kept ⊔{c}     {PI ∧ outp}
             end {PI ∧ outp}
proc get(#v:int)
{PI ∧ GI} begin {PI ∧ GI} > {PI}
                 when here.length = 1 : {PI ∧ here.length = 1}
                        here.get(v) {postg}
                 end; {postg}
                 <{GI ∧ postg}
                  kept := kept - {v}     {PI ∧ outg}
             end {PI ∧ outg}
begin {kept = away = ∅}
     here := []; rest := 0 {cyinv}
     cycle here.length = 2 : {PI ∧ here.length = 2}
                if here[1] ≤ here[2] : temp := here[2]; here := [here[1]]
                 | here[1] > here[2] : temp := here[1]; here := [here[2]]
                end; {pre1}
                <call sort[succ]'.put(temp); rest := rest + 1;
                away := away ⊔{temp}; kept := kept - {temp}>
                {cyinv}
             | here.length = 0 ∧ rest > 0 : {PI ∧ here.length = 0 ∧ rest > 0}
                <call sort[succ]'.get(temp); rest :=  rest - 1;
                away := away - {temp}; kept := kept ⊔{temp}>; {post2}
                here := [temp]
                {cyinv}
     end
end
```

where $PI \equiv 0 \le here.length \le 2 \wedge rest \ge 0 \wedge rest = |away| \wedge kept = con(here)$
$\wedge\ (here.length > 0 \wedge rest > 0 \rightarrow here[1] \le min(away))$,

$GI \equiv \bigwedge_{i=1}^{n-1} away[i] = away[i+1] \sqcup kept[i+1]$,

$postp \equiv 0 < here.length \le 2 \wedge rest \ge 0 \wedge rest = |away| \wedge kept \sqcup \{c\} = con(here)$
$\wedge\ (rest > 0 \rightarrow here[1] \le min(away))$,

$outp \equiv away[pred] \sqcup \{c\} = away[this] \sqcup kept[this]$
$\wedge \bigwedge_{i=1}^{n-1} (i \ne pred \rightarrow away[i] = away[i+1] \sqcup kept[i+1]$,

$postg \equiv here.length = 0 \wedge rest \ge 0 \wedge rest = |away| \wedge kept = \{v\}$
$\wedge\ (rest > 0 \rightarrow v \le min(away))$,

outg $\equiv$ $\text{away[pred]} = \text{away[this]} \amalg \{v\} \wedge \text{kept[this]} = \emptyset$
$\wedge\ (|\text{away[this]}| > 0 \rightarrow v \leq \min(\text{away[this]}))$
$\wedge\ \bigwedge_{i=1}^{n-1} (i \neq \text{pred} \rightarrow \text{away}[i] = \text{away}[i+1] \amalg \text{kept}[i+1])$,

cyinv $\equiv$ $\text{PI} \wedge \neg(\text{here.length} = 2 \vee (\text{here.length} = 0 \wedge \text{rest} > 0))$,

pre1 $\equiv$ $\text{here.length} = 1 \wedge \text{rest} \geq 0 \wedge \text{rest} = |\text{away}| \wedge \text{kept} = \text{con(here)} \amalg \{\text{temp}\}$
$\wedge\ \text{here}[1] \leq \min(\text{away} \amalg \{\text{temp}\})$,

post2 $\equiv$ $\text{here.length} = 0 \wedge \text{rest} \geq 0 \wedge \text{rest} = |\text{away}| \wedge \text{kept} = \{\text{temp}\}$
$\wedge\ (\text{rest} > 0 \rightarrow \text{temp} \leq \min(\text{away}))$.

We want to prove:

{aux = away[0] = away[1] $\amalg$ kept[1]} <u>call</u> sort[1]'.get(elem) {elem = min(aux)},

i.e., the element output by sort[1]'.get is the least of all elements still in the sorting process. To prove this we shall have to prove the correctness of the assertions in sort[1]', and therefore in sort[2]', and so on.

We prove by induction the correctness of the assertions in sort[i]' for i from n downto 1, assuming sort[n+1]' will not be called (this is the case iff rest[1] $\leq$ n-1, which we shall not prove now). The assumption tells us sort[n]' will never go into the cycle. So, the correctness of the initialization part of sort[n]' is trivial.

To prove the correctness of the procedures we have to know the meaning of here.put and here.get. Obviously here.put stands for here := here^[c] and here.get for v := here[1]; here := []. Now {p} here.put(c) {q} and {p} here.get(v) {q}, with the p and q defined above, follow by A1, R2 and R3. Note the importance of here[1] $\leq$ min(away) in PI in the deduction of outg.

Because in both cases the precondition is PI, the first and last premisses of the <u>when</u>-rule are equivalent and we just proved them; the second premiss is a mere triviality, so by the <u>when</u>-rule we prove the part between > and < of the procedures. The parts before > and after < are easy.

Note that in this proof we did not use the induction hypothesis, so the proof holds for all processes.

Now we shall prove the correctness of the proof outlines for the initial statements of sort[i]' for i < n, assuming we have a complete proof for sort[i+1]'. By the usual rules the part before the cycle is correct.

To prove the cycle, we have to prove, by R9':

{PI $\wedge$ here.length = 2} <u>if</u> ... > {cyinv},

{PI $\wedge$ here.length = 0 $\wedge$ rest > 0} <<u>call</u> ...[temp] {cyinv},

{cyinv $\wedge$ here.length = 2} if ... {cyinv},

{cyinv $\wedge$ here.length = 0 $\wedge$ rest > 0} <<u>call</u> ...[temp] {cyinv},

cyinv $\wedge \neg$(here.length = 2 $\vee$ (here.length = 0 $\wedge$ rest > 0)) $\rightarrow$ PI.

Because cyinv $\rightarrow$ PI it is sufficient to prove the first two clauses. The difficult parts are the bracketed sections for which we give more elaborated proof outlines (the other parts are left to the reader):

a) {pre1}

<{pre1 $\wedge$ GI} <u>call</u> sort[succ]'.put(temp) {out1}

rest := rest + 1; away := away $\amalg$ {temp}; kept := kept - {temp}

{post1 $\wedge$ GI} > {post1},

with post1 $\equiv$ cyinv,

pre1 is defined as above,

out1 $\equiv$ here.length = 1 $\wedge$ rest $\geq$ 0 $\wedge$ rest = |away| $\wedge$ kept = con(here) $\amalg$ {temp} $\wedge$

$\wedge$ here[1] $\leq$ min(away $\amalg$ {temp}) $\wedge$ away[this] $\amalg$ {temp} = away[succ] $\amalg$

$\amalg$ kept[succ] $\wedge$ $\bigwedge_{i=1}^{n-1}$ (i $\neq$ this $\rightarrow$ away[i] = away[i+1] $\amalg$ kept[i+1]).

To prove {pre1 $\wedge$ GI} <u>call</u> sort[succ]'.put(temp) {out1}.
We must prove(R12):

{pre1 $\wedge$ GI $\wedge$ PI_{succ}} c[this]:= temp {GI $\wedge$ PI_{succ} $\wedge$ pre1},

{PI_{succ}} Body.put {$\text{postp}_{\text{succ}}$} and (i)

{$\text{postp}_{\text{succ}}$ $\wedge$ pre1 $\wedge$ GI} kept[succ]:= kept[succ] $\amalg$ {c[this]} {out1 $\wedge$ PI_{succ}}.

But this is not strong enough, so we add a new auxiliary variable freeze to hold the value of temp. The clauses now become:

{pre1 $\wedge$ GI $\wedge$ PI_{succ} $\wedge$ freeze = temp} c[this]:= temp
{GI $\wedge$ PI_{succ} $\wedge$ c[this] = freeze $\wedge$ pre1 $\wedge$ freeze = temp},

{PI_{succ} $\wedge$ c[this] = freeze} Body.put {$\text{postput}_{\text{succ}}$ $\wedge$ c[this] = freeze},

{$\text{postput}_{\text{succ}}$ $\wedge$ c[this] = freeze $\wedge$ pre1 $\wedge$ freeze = temp $\wedge$ GI}
kept[succ]:= kept[succ] $\amalg$ {c[this]} {out1 $\wedge$ PI_{succ}}.

The first and last clause hold by A1 and R3; the second clause without freeze (as in (i)) was correct by our induction hypothesis. But now a problem occurs for which we need the addition of at(S) in the <u>when</u>-rule: if we want to deduce c[this] = freeze we have to know PI_{succ} $\wedge$ b $\rightarrow$ c[this] = freeze, but that would be impossible without the use of at(S). We now use a new PI which is PI $\wedge$ (at(S) $\rightarrow$ c[pred] = freeze) where S is the <u>when</u>-statement in <u>proc</u> put. By choosing

p1 $\equiv$ PI_{succ} $\wedge$ c[this] = freeze,

p2 $\equiv$ pre1 $\wedge$ freeze = temp, and

p3 $\equiv$ $\text{postp}_{\text{succ}}$ $\wedge$ c[this] = freeze, in the application of the <u>when</u>-rule we obtain:

$\{pre1 \wedge GI \wedge freeze = temp\}$ call sort[succ]'.put(temp) $\{out1\}$.

By substituting temp for freeze we get the statement to be proved (R6 and R3).

The remainder of this proof outline follows by application of assignment axiom (A1), consequencerule (R3), compositionrule (R2) and finally bracketed section rule (R11).

b) The other proof outline is:

$\{pre2\}$

$<\{pre2 \wedge GI\}$ <u>call</u> sort[succ]'.get(temp) $\{out2\}$

rest := rest - 1; away := away - $\{temp\}$; kept := kept$\perp\!\!\!\perp$ $\{temp\}$

$\{post2 \wedge GI\} > \{post2\}$,

where post2 is defined above,

$pre2 \equiv PI \wedge here.length = 0 \wedge rest > 0$,

$out2 \equiv here.length = 0 \wedge rest > 0 \wedge rest = |away| \wedge kept = con(here) \wedge$

$\wedge\ temp \le min(away) \wedge away[this] = away[succ] \perp\!\!\!\perp \{temp\} \wedge$

$\wedge\ kept[succ] = \emptyset \wedge \bigwedge_{i=1}^{n-1} (i \ne this \rightarrow away[i] = kept[i+1] \perp\!\!\!\perp away[i+1])$.

To prove $\{pre2 \wedge GI\}$ <u>call</u> sort[succ]'.get(temp) $\{out2\}$, we have to prove, by R12:

$\{pre2 \wedge GI \wedge PI_{succ}\}$ <u>skip</u> $\{pre2 \wedge GI \wedge PI_{succ}\}$,

$\{PI_{succ}\}$ Body.get $\{postg_{succ}\}$,

$\{postg_{succ} \wedge pre2 \wedge GI\}$ kept[succ] := kept[succ] - $\{v[this]\}$;

temp := v[this] $\{out2 \wedge PI_{succ}\}$.

The first clause is trivial; the second one was proved (induction hypothesis) in the proof of sort[succ]', and the third one follows by A1, R2 and R3. By these same rules one can prove the remainder of the bracketed section. Application of the bracketed sectionrule (R11) yields what we wanted to prove.

Now we have proved the premiss, belonging to this application of R9', and, thereby, we proved the initial statements of the sorting processes correct.

The proof of

$\{aux = away[0] = away[1] \perp\!\!\!\perp kept[1]\}$ <u>call</u> sort[1].get(elem) $\{elem = min(aux)\}$

is analogous to the proof of the second bracketed section, and justifies the heading of this paragraph.

Q.E.D.

7. CONCLUSIONS

We have developed a deductive system for proving properties of DP programs.

Unfortunately the system up to now is not complete w.r.t. proofs of properties about programs not free from deadlock w.r.t. external requests. This will be shown by the following program:

```
process P1;
x : int
proc pr1 begin skip end
begin x := 0; call P2.pr2(x) end

process P2;
proc pr2(#y : int) if true : y := 37; call P1.pr1
                   | true : y := 73
                   end
begin skip end
```

The program will deadlock, when the first alternative of the if-statement within the procedure pr2 is chosen on execution of P_1's request (cf. 2.5). Consequently, after termination of the program $x = 73$ holds. However we are not able to prove exactly this post-assertion, because the assertion depends on information about (durable) suspension. In the present system, we can prove $x = 37 \vee x = 73$. The proof is easy and therefore omitted. We are still working on this problem and, at present, we are indeed able to prove the program above w.r.t. the post-condition $x = 73$. But more study is needed for the analogous case with guarded regions.

Besides, this phenomenen raises exciting possibilities for the implementation of recursion within a fixed network of distributed processes. Namely, by allowing inter-process communication in the requesting process, during its suspension. Also, in this direction, research is going on, in order to find the appropriate version of Scott's induction rule.

Another complication concerns the index mechanism of an array of processes. With the proof system, so far, we cannot handle cases in which the existence of requested processes in the array, i.e., their index being between the array bounds, is of importance to the proof.

The "germ" of our proof system is the separation of proofs of the various processes. The proof of a single process, i.e., program module in the terminology of [7], depends on assertions that cannot be modified by the concurrent actions of other processes. To this end, private auxiliary variables (to store the history of the process, as far as concerned), process invariants, a global invariant, and bracketed sections (to indicate elementary, i.e., atomic actions) are introduced.

Faith in this method was gained, in particular, through Owicki's paper [7]. In this paper, Owicki presents an analogous (w.r.t. auxiliary variables) method for proving shared data types, which are similar to the common procedures in DP. For each program module, so called, "private" variables are included, to store its sharing history w.r.t. a particular data type. To each data type a, so called, "global" variable is added, to store its sharing history w.r.t. the program modules. In this way, "private" and "global" variables can be charged only when sharing of this particular data type takes place. Thus, our auxiliary variables (AV) are similar to Owicki's "global" and "private" variables.
Our particular set-up is motivated by our wish to prove at a later stage relative completeness of our proof system, using techniques from K.R. Apt's manuscript of a completeness proof for the CSP-proof system of [2].

In this paper global program control is reproduced with the help of auxiliary variables and the global invariant, GI, while local process control is modelled by the use of variables and auxiliary variables private to the process and by the process invariant PI. While GI registers the communications between the processes, PI functions as go-between for the variables of the process and the variables in GI. In section 4.1, we showed that PI alone was not sufficient, whereas certainly GI cannot take the role of PI in itself, because we want to keep the proofs of the processes private w.r.t. the variables used in them; besides, their extensions do not coincide (e.g. PI need not hold at the beginning of a process). The arguments leading to the present system have all been motivated within the article.

Finally we want to thank Krzysztof Apt for the opportunity to present and discuss this proof system in its various stages, and, especially, Nissim Francez for the discussion of this material during his visit at the University of Utrecht in August 1980, during which the outline of the system came in sight. We are grateful to various members of the Department of Computer Science of the Mathematical Centre in Amsterdam for the opportunity to present and discuss an earlier stage of the system.

REFERENCES

[1] Apt, K.R., Recursive assertions and parallel programs, to be published in Acta Informatica.

[2] Apt, K.R., N. Francez and W.P. de Roever, A Proof System for Communicating Sequential Processes. ACM Trans. Program. Lang.-syst. 2, 3(July 1980),359-385.

[3] Bakker, J.W. de, Mathematical Theory of Program Correctness, Series in Computer Science, Prentice-Hall International, 1980.

[4] Brinch Hansen, P., Distributed Processes: A Concurrent Programming Concept., Comm. ACM 21, 11 (November 1978), 934-941.

[5] Lamport, L., The Hoare's Logic of Concurrent Programs. Acta Inf. 14 (1980), 21-37.

[6] Levin, G.M., A Proof Technique for Communicating Sequential Processes (with an example), Techn. Rep., Computer Science Dep., Cornell University, 1979.

[7] Owicki, S., Specifications and proofs for abstract data types in concurrent programs, Rep. TR 133, CRC Stanford University, 1977.

[8] Owicki, S. and D. Gries, Verifying properties of parallel programs: An Axiomatic Approach, Comm. ACM 19, 5 (May 1976) 279-285.

[9] Owicki, S. and D. Gries, An axiomatic Proof Technique for Parallel Programs I. Acta Inf. 6 (1976) 319-340.

[10] Welsh, J., A.M. Lister and E.J. Salzman, A comparison of two notations for Process Communication. Department of Computer Science, University of Queensland, St. Lucia, Qld 4067.