A STUDY IN DISTRIBUTED SYSTEMS AND DUTCH PATRIOTISM

Rob Gerth

Willem P. de Roever

Marly Roncken

RUU-CS-82-10

juni 1982

Rijksuniversiteit Utrecht

Vakgroep informatica

Princetonplein 5
Postbus 80.002
3508 TA Utrecht
Telefoon 030–53 1454
The Netherlands

A STUDY IN DISTRIBUTED SYSTEMS AND DUTCH PATRIOTISM

Rob Gerth

Willem P. de Roever

Marly Roncken

Technical Report RUU-CS-82-10

juni 1982

Department of Computer Science

University of Utrecht

P.O. Box 80.002, 3508 TA Utrecht

the Netherlands

A study in Distributed Systems and Dutch Patriotism

by

Rob Gerth[1]

Willem P. de Roever[2]

Marly Roncken

Department of Computer Science, University of Utrecht

P.O.Box 80.002, 3508 TA Utrecht, the Netherlands

ABSTRACT

One of the customary ways to introduce process-communication in parallel programming languages, is by means of procedure-calls: each process has a fixed set of entry-procedures and other process may issue external requests for one of these procedures. The calling process then waits until the process being called, after accepting the request, has finished executing the body of the entry-procedure.

This form of communication was first advocated in [Brinch Hansen, 1975] and has been incorporated in many languages since. Notably in Concurrent Pascal [Brinch Hansen, 1975], Distributed Processes [Brinch Hansen, 1978], Mesa [Lampson et al, 1980], Modula2 [Wirth, 1980] and ADA [Ichbiah et al, 1979].

Up to now, there has been no attempt to construct a proof theory for languages based on this form of communication, which not only is sound, but also (relatively) complete. The current paper is a first attempt at such a theory.

We have chosen Distributed Processes (DP) as a vehicle for our study, because it is the simplest of the above languages, exhibiting the essential characteristics. Moreover, DP is one of the languages which directly influenced the design of ADA.

Technically, one might view DP as an intermediate between CSP [Hoare, 1978] and Owicki's language in [Owicki et al, 1976]: the external request mechanism in DP can be seen as a CSP-handshake; each DP-process however, can logically be divided into a (varying) number of sub-processes which are working in a shared variable environment. These similarities are reflected in the proof system by the introduction, from [Apt et al, 1980], of a general invariant and a cooperation test to describe the process-interactions, and, from [Owicki et al, 1976], of a process invariant for each of the DP-processes and an interference freedom test (IFT) to describe the execution within each of the processes.

These notions needed however adaption; especially in the case of the IFT, because the number of sub-processes within each DP-process varies dynamically, due to communication with the process. To capture this, the novel notion of calling chain assertion is introduced , which characterizes the processes which are waiting for some specific communication -activity to finish.

In the paper we present a proof system for partial correctness properties for DP and use it to prove correct a complicated algorithm which is a solution to a problem inspired by the Dutch national flag [Dijkstra, 1977].

---

1 This author is supported by the Netherlands Organization for the Advancement of Pure Research (ZWO).

2 The research reported in this paper benefitted from several visits of the second author to academic institutions in Israel, sponsored by the Netherlands organization for the Advancement of Pure Research (Z.W.O.).

1. INTRODUCTION

We present an overview of a Hoare-like proof system for the language Distributed Processes (DP). DP has been introduced in [Brinch Hansen, 1978] and is a generalisation of the Concurrent-Pascal-Monitor concept. In the context of DP, we study the proof theory of concurrently executing processes which communicate in a monitor-based fashion.

Communication between two processes is established when one process accepts a call to one of its **entry-procedures** (common procedure in DP), issued by the other process. Issuing such a call, raises the (monitor-) lock of the caller until the call is processed, thus inhibiting other communication-activities to be established. Accepting such a call, raises the lock of the callee until either the operation, which has been started by the call finishes or the lock is released voluntarily.

Previous papers dealing with the subject, [Hoare, 1974; Howard, 1976], were limited to a discussion of proof-methodologies and heuristics. The current research aims at a relatively complete proof system in the sense that we want to be able to prove any operationally meaningfull assertion about the computation of a concurrent program. The soundness and completeness proofs for the proof system are projected to be the subject of the first author's dissertation.

This requirement leads to the essential notion of **calling-chain-assertions.** These assertions may be interpreted as characterizing the fact that a process (or a chain of processes, as execution of a call may cause other calls to be issued) is waiting for some communication activity to finish.

Proofs of the individual processes, which make up a DP-program, depend on each other because communication between the processes will in general affect the validity of these proofs. Therefore a **general invariant**, GI, is introduced which expresses this interdependence by keeping track of these communication activities. Moreover, a **process invariant**, PI, is introduced separately for each process; this invariant generalizes the known notion of **monitor invariant**, e.g. [Hoare, 1974], and characterizes the state of a process when synchronisation actions, like process-communication, occur within that process. To prove the invariance of PI and GI we have to combine (and adapt) (1) the notion of **interference freedom test** introduced in [Owicki and Gries, 1976] in their proof system for languages in which processes communicate by sharing variables, with (2) the notion of **cooperation test** introduced in [Apt et al, 1980] in their proof system for CSP [Hoare, 1978] in which processes have disjoint state-spaces and communicate through handshaking.

The major difficulty in applying Owicki´s interference freedom test is, that contrary to Owicki´s case, in DP the interference conditions which have to be checked are not determined syntactically but rather semantically. The calling-chain-assertions, referred to above, are used to determine which conditions need to be checked.

The remainder of this summary is organized as follows. Section 2 contains the proof-theoretical description of the interplay between process-communication and the execution of an individual DP-process. In section 3 some proof rules are presented; only the <u>when</u>-rule and the <u>call</u>-rule are discussed in some depth, because these are the essential rules of DP. Section 4 contains a small example proof. Section 5 introduces the various consistency-checks which must be imposed upon the separate process-proofs. Finally in section 6 the use of the complete proof system is illustrated by proving correct a subtle algorithm of Dijkstra's. This algorithm is a solution of a problem inspired by the Dutch national flag [Dijkstra, 1977]. For ease of reference, two appendices contain respectively a brief (but complete) description of Brinch Hansen's language, and a summary of the proof rules and terms. In the sequel we will assume the reader to be familiar with the basic ideas of DP.

2. <u>THE PROOF SYSTEM</u>

One of the milestones in the theory of program proving is undoubtedly the axiomatic proof method using Hoare-assertions [Hoare, 1969]. His method has received much attention, and for most of the sequential programming language constructs, proof rules have been proposed [de Bakker, 1980]. Parallelism however proved be more evasive. The first break-through was achieved in [Owicki and Gries, 1976] who contrived a proof system for languages in which process-communication takes place by sharing variables and the only non-interruptable (i.e. atomic) action within a process is the memory reference. On the other side of the language-spectrum, one finds CSP [Hoare, 1978] in which processes have disjoint state-spaces, where communication is established through an explicit handshaking-mechanism and in which, consequently, atomic actions extend from one handshake to another. A proof system for CSP is presented in [Apt et al, 1980].

DP occupies a midway position, in that it incorporates aspects of both of the above languages: the external request mechanism of DP may be thought of as a CSP-handshake; within each process however, one can envisage the initial statement together with the procedure incarnations, activated by external requests, as a set of sub-processes working in a shared variable environment and whose operations are interleaved.
These similarities are reflected in the proof system by the introduction of Apt´s **cooperation test** and Owicki´s **interference freedom test**.

2.1. Partial correctness and concurrency

At the moment, the proof system deals with partial correctness proofs.
In order to get a meaningfull application of Hoare´s {p}S{q}-formalism, termination of a DP-program must be defined:

> A DP-program has **terminated** iff all processes making up the program have terminated their initial statements.

Notice that by this definition, a program which deadlocks, has not terminated.
However in a concurrent context, termination is not always what we want. Some buffer-process between a disk and a lineprinter, for instance, may be required to be active all the time and we should be able to prove that this process always functions correctly. For such properties, Hoare´s method has to be extended along the lines of [Lamport, 1980] so that so called **safety properties** may be proved which state that something bad cannot happen. In this context, partial correctness is a safety property, stating that the program cannot terminate with incorrect output.
This extension however does not alter those parts of the proof system which are presented here, except for the parallel composition rule.

2.2. Introduction of general invariant, auxiliary variables, bracketed sections and process invariant

The main characteristic of DP, upon which the proof system is built, is the **synchronisation-at-waiting-points**. In particular it is essential to acquire a clear picture of the way in which the interplay between external requests and the interleaving of operations in each of the individual processes (cfr. appendix 1 section 1.1) is described in the proof system. This will be explained using a picture :

Fig. 1 shows the execution of 4 processes, The horizontal lines represent the time-axes for each of the respective processes, P_1, P_2, P_3 and P_4, along which execution proceeds. During execution of P_1, an external request is made (remember that this is an atomic action as far as P_1 is concerned). Execution of P_2 proceeds until a waiting point is encountered, denoted by the (leftmost vertical wriggles), at which synchronisation takes place. P_2 continues by honouring P_1´s request and starts executing a new incarnation of the procedure-body of pr. During this execution, other waiting points are encountered and external requests are made (notice that the first call apparently is not part of this incarnation, while the second one is). Finally execution arives at the end of the procedure-body, another waiting point, after which P_1 and P_2 proceed independently again. (it may be instructive to construct a program (-skeleton) that exhibits this behaviour).

The very act of honouring an external request will affect P_2´s state, because a fresh procedure incarnation has to be created. Moreover, when parameter-passing occurs, external information from the caller (P_1) will be injected into the callee (P_2) by the value-transfer at the beginning of procedure execution and from P_2 to P_1 by the assignment to the variables of the call at the end of procedure execution.

It should be clear that the proof system must somehow capture these process-interactions, if we want to prove assertions about DP-programs. Therefore the **general invariant**, GI, is introduced, which will keep track of the changes in the process-states, created by these external requests. To be able to express properties about each of the process-states, a new class, AV, of

auxiliary variables is introduced; the need for such a class in proof systems for paralell programs is a well-known phenomenon.

As the variables used within GI will have to be updated with every external request, we cannot expect GI to be valid throughout the whole program, which contradicts the property of being an invariant. This is remedied by the introduction of **bracketed sections**, <S>, associated with each call, in which GI need not hold, [Apt et al, 1980]. Assignments to the variables of GI will be confined to these bracketed sections only (overlay 1). Consequently, we only need to check whether GI is left invariant by execution of each bracketed section.

There arises however a problem: according to the above discussion each call should somehow be inside a bracketed section. Consider the call in P_1 (fig. 1). Honouring this call by P_2 results in two more calls being made. For these calls, GI has to hold again before entering the bracketed sections surrounding them. But these latter calls apparently take place **within** the bracketed section in P_1, in which validity of GI is not assured. This seemingly contradictory situation is remedied by noticing that the validity of GI is required **only when parameter passing occurs**, because it is only at these positions that external information is injected into the processes. Hence updating the variables of GI may be restricted to these points. For each call, there are at most two places where parameter-passing occurs: at the beginning and at the end of procedure execution. So we can refine the notion of **bracketed section** (as suggested in overlay 2) by associating with the opening-bracket in P_1, the leftmost closing-bracket in P_2, and with the closing-bracket in P_1 the rightmost opening-bracket in P_2(*).
In other words, each procedure body is extended by a pre- and postlude in which the GI-variables are updated.

At a waiting point, external requests may be accepted, resulting in the execution of some procedure body. In order to characterize the (local) state in which execution of this procedure body starts, for each process P_i a **monitor invariant**

(*) By a similar argument one can show that bracketed sections should not contain when-statements (i.e. possible waiting points) either

MI_i is introduced which is required to hold at each waiting point (such invariants have initially be introduced in [Hoare, 1974]). The monitor invariant MI_i will in fact be part of a **process invariant** PI_i (overlay 3).
However, PI contains more. Firstly, as with monitor-entries, we associate with each common procedure a pre- and post-assertion specifying the input-output behaviour for use in the call-rule (cfr. section 3.2). Secondly, when writing the proofs for the initial statement and the procedure-bodies of a DP-process, we would like to construct them as were they proofs in a sequential language and defer the problems, associated with the fact that interleaving takes place, to a later stage (as it does in [Owicki and Gries, 1976], this will lead to an interference freedom test). Interleaving at the when-statements is semantically determined, so besides the normal sequential assertion, there is a special one associated with these points, used in case the when-statement serves as a waiting point and interleaving takes place. Thirdly, the fact that the initial statement of some process has terminated, is significant; from this time on the process will not originate communication activities anymore with other processes, i.e. from now on, if the process issues an external request, it will be because some other process called one of its common procedures. Hence, to this point, we will attach a special assertion too. Finally, PI will record the initial state in which execution of the corresponding process starts.

Thus, each PI consists of four separate parts:

(1) the monitor-invariant (MI),

(2) the pre- and post-assertions of the common procedures,

(3) the (special) assertions associated with the when-statements and the point after the initial statement,

(4) the assertion associated with the point preceding the initial statement.

The assertions in the last two categories are attached to specific locations. So, to specify PI, a notation is needed to specify **syntactical** locations. First, for each syntactical occurence of some statement S, a unique **name** 'S' is introduced to distinguish between the occurences. These names will not be defined formally and can be labels, for instance.

The following standard names will be used:

'W' or 'W_i'	the name of some occurrence of a when-statement,
P.pr	the body of the procedure pr in process P without its pre- and postlude,
P.pr.body	the body of the procedure pr in process P without its prelude and
$Init_i$	the initial statement of process P_i.

Sometimes we will omit references to the process (and use Init or pr) if it is clear or immaterial which process is meant.

With this notation, the following **location specifiers** can be defined

at('S')	denoting the point just before 'S' and
after('S')	denoting the point immediately following 'S'

Finally, notation is introduced to define or refer to "location specific" assertions (loc denotes some location specifier):

loc:p	associating the assertion p with the location loc
PI:loc	denoting those parts of PI that hold at loc

With this notation, PI_k of some process P_k can be specified as:

$$PI_k \equiv \bigwedge_{pr_i} ((at(pr_i):pre_i) \wedge (after(pr_i):post_i)) \wedge \bigwedge_{'W_i'} (at('W_i'):p_i \wedge MI_k)) \wedge$$
$$(at(Init_k):q_1) \wedge (after(Init_k):(q_2 \wedge MI_k))$$

where the first disjunction ranges over all common procedures in P_k and the second one over all when-statements in P_k.

Then PI_k:after($Init_k$) denotes $MI_k \wedge q_2$,

PI_k:at('W_3') denotes $MI_k \wedge p_3$ etc.

PI is written as some formula of a predicate logic, i.e., as an ordinary assertion, although PI is **not** an assertion; only expressions like PI:at(pr_3) can be (part) of an assertion. As there is small chance of confusion, we prefer this somewhat inexact notation to a formally correct one.

Our location specifiers look exactly the same as Lamport's location predicates ([Lamport, 1980]). Semantically however, they are quite different. Lamport's predicates have an operational meaning; for instance, the location predicate at('S') is a valid assertion, being true iff control is just before the statement denoted by 'S'. In contrast, our location specifier at('S') is not part of our assertion language. It can only be used in terms such as PI:at('S') or at('S'):p and is thus purely a notational device.

2.3. Free variables and assertions

In DP, each process has its own memory which cannot be accessed directly by the other processes of the DP-program. Within the proof system we have to work with variable(-names). So the proof-theoretical pendant of the above restriction is: Processes have mutually disjoint name-spaces.

However this restriction is slightly too strong and we define:

> own(P_i) - the set of variables which are subject to change in process P_i.
>
> a variable, x, may appear (free) in process P_i or in an assertion belonging to the proof of P_i iff $x \notin \text{own}(P_j)$, $j \neq i$, i.e. iff x is not changed by any other process.

In the sequel, free(·,·,..., ·) will denote the set of all variables appearing free in the processes, assertions or expressions in the argument list.

3. THE PROOF RULES

Only the rules that are characteristic for DP are discussed in some depth; for a complete list see appendix 2.

3.1. Rule for guarded regions: the when-rule

$$\frac{\begin{array}{l}\{p\wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p\wedge b_n\}\ S_n\ \{q\},\\ p\wedge\neg\bigvee_{i=1}^{n} b_i \rightarrow \text{PI:at('S')},\\ \{b_1\wedge\text{PI:at('S')}\}\ S_1\ \{q\},\ \ldots,\ \{b_n\wedge\text{PI:at('S')}\}\ S_n\ \{q\}\end{array}}{\{p\}\ \underline{\text{when}}\ b_1 : S_1 | \ldots | b_n : S_n\ \underline{\text{end}}\ \{q\}}$$

This rule is of interest, because the <u>when</u>-statement may serve as a waiting point: when all the guards of this statement evaluate to false, the process may resume execution at an earlier waiting point, or honour a new external request. In the discussion in section 2.2 we suggested that at these times PI takes over;

this is expressed by the second premiss. Here, ´S´ denotes this particular occurence of the when-statement. So PI:at(´S´) specializes to that part of PI applicable to S. Whenever the statement is resumed again, anything may have happened to the variables in assertion p and, consequently, p need not hold anymore. However now one of the b_i´s is true and we just passed a waiting point, otherwise control could not have switched back to this when-statement. Therefore, the monitor part of PI holds and, assuming that the possible 'special' assertion attached to 'S' holds too, we have arrived at a situation equivalent to a guarded conditional with pre-assertion PI:at('S'); whence the third premiss. (Notice, that to assure the validity of such a special assertion when execution resumes at the corresponding when-statement, an interference freedom test is needed to show that the assertions are invariant over operations which can be interleaved.) The first premiss covers the case that the when-statement does not act as a waiting point, i.e. at least one boolean guard b_i is true on arrival.

3.2. External request rule (*)

$$\frac{\begin{array}{l}\{p \wedge MI_j \wedge GI\}\ S_1;T_1[\cdot]\ \{PI_j{:}at(P_j.pr)[\cdot] \wedge p_2 \wedge GI\},\\ \{PI_j{:}at(P_j.pr)\}\ T\ \{PI_j{:}after(P_j.pr)\},\\ \{p_2 \wedge PI_j{:}after(P_j.pr)[\cdot] \wedge GI\}\ T_2[\cdot];S_2\ \{q \wedge MI_j \wedge GI\}\end{array}}{\{p\}\ <S_1;\ \underline{call}\ P_j.pr(\overline{t},\overline{y});\ S_2>\ \{q\}}$$

where (1) P_i contains the bracketed section,

(2) P_j the declaration $\underline{proc}\ pr(\overline{u},\overline{v})\ \underline{begin}\ T_1;>\ T\ <;T_2\ \underline{end}$ $(i \neq j)$,

(3) $[\cdot] = [\overline{t}/\overline{u},\ \overline{y}/\overline{v}]$,

(4) $free(p,q) \subseteq free(P_i)$

(5) $free(p_2) \subseteq free(P_i) \backslash \overline{y}$

(6) $free(PI_j{:}at(pr), PI_j{:}after(pr)) \subseteq free(P_j)$

There are two aspects to an external request: (1) the execution of the procedure-body and, as far as the caller is concerned, there is no difference with a procedure-call in a sequential language and (2) the process-communication as embodied in the updating of the GI-variables in the bracketed section surrounding the call. We discuss the latter aspect first.

(*) The barred parameters and variables in this section denote parameter and variable lists

In section 2.2 we have argued that ensuring the invariance of GI over the bracketed section containing the call is not enough. We had to construct with every call, two new bracketed sections (in our context $<S_1;T_1>$ and $<T_2;S_2>$) and prove invariance of GI over those. These tests are embodied in the first and third premiss of the rule(*).

The PI_j-terms denote the pre-assertion of the common procedure being called and the post-assertion as discussed in section 2.2. Notice however, that they are the pre- and post-assertions of the procedure body without its pre- and postlude.

Next the "procedure-call aspect" of an external request.

In order to render the sequential aspects of a call as simple as possible, the following restrictions are introduced:

(1) at procedure-entry, the variables, $\bar{y}$, of the call are assigned to the output parameters $\bar{v}$,

(2) the y_i's are pairwise disjoint,

(3) $\mathrm{free}(\bar{t}) \cap \bar{u} = \emptyset$,

(4) $(s \in \mathrm{free}(\bar{t}) \cap \bar{y}) \wedge (s \notin \bar{u} \cup \bar{v}) \rightarrow s \notin$free("body of P_j.pr")

Restriction (2) is fairly natural for DP because the order in which the variables of a call are assigned their values at the end of procedure-execution is undefined. Restriction (4) is implied by the constraint on the variables of a DP-process (cfr. section 2.3).

If (2), (3) and (4) hold, one can show that

$S[\bar{t}/\bar{u}, \bar{y}/\bar{v}]$ = $\underline{\text{begin}}$ $\underline{\text{new}}$ $\bar{u},\bar{v}$; $\bar{u}:=\bar{t}$; $\bar{v}:=\bar{y}$; S; $\bar{y}:=\bar{v}$ $\underline{\text{end}}$

(*) Connaisseurs of the CSP-proof system in [Apt et al, 1980] will notice the similarity of these premisses with the cooperation test of matching bracketed sections. This similarity will be expounded upon in section 5

Hence, because of (1), the value-assignments to the formal parameters on a procedure-call can be replaced by a substitution of variable-names as in the premisses of the external-request-rule (**).

The further workings of the rule are simple now:
The procedure-body is proven correct only once and this gives the input-output behaviour of the common procedure for any legal input; this is the second premiss, $\{p_1\}\ T\ \{q_1\}$, of the rule. In the first premiss we must show that the input is legal by deriving the post-assertion $p_1[\cdot]$. If the input is legal, $q_1[\cdot]$ specifies the procedure-output. The intermediate assertion, p_2, is needed to retain information about variables in P_i other than the actual parameters (these variables cannot be changed by the call). This information cannot be placed in p_1 because p_1 is an assertion from the proof of the procedure-body within the other process.

3.3. Parallel composition rule

$$\frac{\{PI_i:at(Init_i)\}Init_i\{PI_i:after(Init_i)\}\text{ and }PI_i\text{ is interference free, }i=1..n}{\{\bigwedge_{i=1}^{n} PI_i:at(Init_i)\wedge GI\}\ [P_1\|\ldots\|P_n]\ \{\bigwedge_{i=1}^{n} PI_i:after(Init_i)\wedge GI\}}$$

where (1) PI_i denotes the process invariant of process P_i,
(2) $Init_i$ is the initial statement of process P_i

This rule should be clear, except for the interference freedom which is explained in section 4.

(**) The above restrictions can be removed, resulting in a more complicated external-request-rule. However, the other parts of the proof system are not affected; i.e. these restrictions do not influence the concurrency features of DP

4. EXAMPLE

We now give an example proof, illustrating the use of the proof system. In the course of the proof, it will be seen that the system, as presented up to now, is still incomplete.

Two monkeys are obliged to write the word 'HUMBUG' in order to get a banana. They can do this by requesting process 'writer' to type one of the syllables 'HUM' or 'BUG' which in cooperation and depending on the typing order may lead to the desired result. The two monkeys are represented by a so-called *process-array* monkey[2], which is nothing but a short-hand notation for having 2 identical processes in the DP-program.

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

process writer
   b : bool; x : int; word : seq[2]seq[3]char; message : seq[16]char
   proc type begin
     if true : word := word^<HUM>; b := false
      | true : word := word^<BUG>; b := ¬b
     end; x := x + 1
   end
   begin
     b := false; x := 0; word := <<>, <>>; message := <>;
     when x = 2 : if b : call monkey[1].eat(banana);
                         call monkey[2].eat(banana)
                   | ¬b : message =  NO_BANANA'S_TODAY
                   end
     end
end
```

This example exhibits synchronization by means of a when-statement. The single guard x = 2 of the when-statement in process 'writer', is initially made false; this implies that the when-statement will act as a waiting point. As a consequence, the outstanding requests for procedure 'type' will be honoured. As each execution of procedure 'type' increases the value of x by 1, both requests must be honoured before execution of the when-statement can be resumed.

Let the processes be numbered as follows:

$P_1 \equiv$ monkey[1], $P_s \equiv$ monkey[2], $P_3 \equiv$ writer. We will prove:

{true}[$P_1 \parallel P_2 \parallel P_3$]{word = <HUMBUG> $\bar{\vee}$ message = <NO_BANANA'S_TODAY>},

where '$\bar{\vee}$' denotes the exclusive-OR operator.

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

$PI_i \equiv MI_i \equiv h_i = 1$ for $i = 1,2$ and

$PI_3 \equiv MI_3 \wedge (at(\ell_1) : message = \langle\rangle) \wedge (at(Init_3):h_3=0) \wedge (after(Init_3):POST)$

$\wedge\ (at(type):MI_3) \wedge (after(type):MI'_3 \wedge x = h_3+1 \wedge len(word) = h_3+1)$

$MI_3 \equiv MI'_3 \wedge h_3 = x \wedge h_3 = len(word)$, where 'len' denotes the syllable-length

$MI'_3 \equiv (x = 0 \rightarrow \neg b \vee word = \langle\rangle) \wedge (x = 1 \rightarrow (b \wedge \langle BUG\rangle \in word) \vee (\neg b \wedge \langle HUM\rangle \in word))$

$\wedge\ (x = 2 \rightarrow \neg b\ \bar{\vee}\ word = \langle HUMBUG\rangle)$

$POST \equiv word = \langle HUMBUG\rangle\ \bar{\vee}\ message = \langle NO\_BANANA'S\_TODAY\rangle$

$GI \equiv h_3 = h_1 + h_2 \wedge h_1 \leq 1 \wedge h_2 \leq 1$

```
process monkey[j]'
$h_j$ : int
proc eat(x) {$MI_j$} begin >"eat x"< end {$MI_j$}
{$h_j = 0$} begin {$h_j = 0$} <call writer'.type; $h_j$ := 1> {$h_j=1$} end {$PI_j$:after('$Init'_j$')}

process writer'
...; $h_3$ : int
proc type; {$MI_3$} begin>
   {$MI_3$} if true : {$MI_3$} word := word^<HUM>; b := false
                      {$MI'_3 \wedge h_3 = x \wedge len(word) = h_3 + 1 \wedge \neg b \wedge \langle HUM\rangle \in word$}
             | true : {$MI_3$} word := word^<BUG>; b := $\neg$b;
                      {$len(word) = h_3 + 1 \wedge x = h_3 \wedge$
                      $(x = 0 \rightarrow b \wedge \langle BUG\rangle \in word) \wedge$
                      $(x = 1 \rightarrow (\neg b \wedge word = \langle BUGBUG\rangle) \vee (b \wedge word = \langle HUMBUG\rangle))$
           end;
           x := x + 1; {$MI'_3 \wedge x = h_3 + 1 \wedge len(word) = h_3 + 1$}
   <$h_3$ := $h_3$ + 1 end {$MI_3$}
   {$h_3 = 0$} begin b := false; x := 0; message := <>;word := <<>,<>>;{$MI_3 \wedge x \neq 2 \wedge message=\langle\rangle$}
   {$PI_3$:at($\ell_1$)} $\ell_1$: when x = 2;          {$(\neg b\ \bar{\vee}\ word = \langle HUMBUG\rangle) \wedge message = \langle\rangle$}
                    if b : {word = <HUMBUG>} <call monkey[1].eat(banana)>;
                           <call monkey[2].eat(banana)> {$word = \langle HUMBUG\rangle \wedge message = \langle\rangle$}
                    | $\neg$b : {word $\neq$ <HUMBUG>} message := <NO_BANANA'S_TODAY>
                           {$word \neq \langle HUMBUG\rangle \wedge message = \langle NO\_BANANA'S\_TODAY\rangle$}
                    end {POST}
                  end {POST}
        end {$PI_3$ :after($Init_3$)}
```

We only prove the call to procedure 'type' and the when-statement in 'writer'. The rest is trivial or follows directly from the proof-outline (using the assignment and skip-axioms, composition, consequence and conjunction-rules).

Consider the call in P_1'. We have to prove $\{h_1 = 0\}$ <call writer.type; $h_1 := 1$> $\{h_1=1\}$

By the external request-rule we have to prove

$\{h_1 = 0 \wedge MI_3 \wedge GI\}$ skip $\{h_1 = 0 \wedge MI_3 \wedge GI\}$

$\{MI_3\}$ "body of type" $\{MI_3' \wedge x = h_3 + 1 \wedge len(word) = h_3 + 1\}$

$\{MI_3' \wedge x = h_3 + 1 \wedge len(word) = h_3 + 1 \wedge h_1 = 0 \wedge GI\}$ $h_3 := h_3 + 1;\ h_1 := 1$

$\{h_2 = h_3 - 1 \wedge h_2 \leq 1 \wedge h_1 = 1 \wedge MI_3\}$.

All this follows readily from the definitions, the proof-outline of P_3' and the assignment and skip-axioms.

We next consider the when-statement.

As $(PI_3 : at(\ell_1) \wedge x \neq 2 \wedge x = 2) \leftrightarrow$ false and $PI_3 : at(\ell_1) \wedge x \neq 2 \rightarrow PI_3$, we obtain by the usual rules the validity of the first two premisses of the when-rule. As to the third premiss, this is shown in the proof-outline, whence application of the when-rule is allowed. Thus we proved $\{h_j = 0\}$ $Init_j$ $\{PI_j \wedge after('Init_j')\}$ for j = 1,2,3

Application of the parallel-composition-rule, consequence and conjunction-rules gives $\{h_1 = h_2 = h_3 = 0 \wedge GI\}$ $[P_1' \parallel P_2' \parallel P_3']$ $\{\bigwedge_{j=1}^{3} (PI_j \wedge after('Init_j')) \wedge GI\}$.

Finally, using the AV-rule and substituting 0 for h_1, h_2 and h_3 delivers the desired formula {true} $[P_1 \parallel P_2 \parallel P_3]$ {word = <HUMBUG> $\bar{\vee}$ message = <NO_BANANA'S_TODAY>}.

In this proof, we have essentially used the validity of the PI_j's at all waiting points, because PI_j is attached to each of them.

The validity of PI_j:after('Init$_j'$') for j=1,2 is guaranteed, because its variables are not used in procedure 'eat'.

As for the first waiting point ℓ_1 in process P_3, the additional assertion, message=<>, attached to it contains no variables used in the procedure-body of 'type' and the rest of PI_3, MI_3, is shown invariant in the proof outline of procedure 'type'.

The other waiting point, after('Init$_3'$'), is not so easy. In fact, PI_3:after('Init$_3'$') will be violated by the execution of (an incarnation of) 'type'.

Fortunately, no such incarnations can exist after termination of Init_3, because there are precisely two monkeys alive, each of them able to call procedure 'type' only once, and they have already done so (x=2).

The above informal reasoning showed that execution of 'type' did not interfere with the validity of PI_3:at(ℓ_1), whence this assertion can safely be assumed whenever execution of the <u>when</u>-statement is resumed; likewise for PI_3:after('Init_3''). In the next section, these interference freedom conditions will be discussed in general.

5. INVARIANCE PROOFS

In the proof system, the invariance of GI throughout the program (except for the bracketed sections) and the validity of the PI_i's at all waiting points are essentially used. However, these claims should be substantiated.
As to GI, by the discussion in section 3.2 it should be clear that the invariance test is 'hardwired' into the proof system by application of the external-request-rule.

For those acquainted with the CSP-proof system in [Apt et al, 1980], it may be instructive to see that this invariance test is a CSP-cooperation test in disguise.
Consider a typical call (in P_i) to some common procedure (in P_j) with intermittent assertions :

$$\begin{array}{l} \{p\}\ \langle S_1;\ \underline{\mathrm{call}}\ P_j.pr(\bar{x},\bar{y});\ S_2\rangle\ \{q\} \\ \underline{\mathrm{proc}}\ pr(\bar{u}\#\bar{v})\ \{MI_j\}\ \underline{\mathrm{begin}}\ T_1;\rangle\{PI_j{:}at(pr)\}\ T\ \{PI_j{:}after(pr)\}\langle;T_2\ \underline{\mathrm{end}}\ \{MI_j\} \end{array} \qquad (1)$$

The first and third premiss of the external-request-rule tell use to prove:

$$\begin{array}{l} \{p\wedge MI_j\wedge GI\}\ S_1;T_1[\bullet]\ \{PI_j{:}at(pr)[\bullet]\wedge p_2\wedge GI\} \\ \{p_2\wedge PI_j{:}after(pr)[\bullet]\wedge GI\}\ T_2[\bullet];\ S_2\ \{q\wedge MI_j\wedge GI\} \end{array} \qquad (2)$$

But remembering the equivalence result of variable-substitution and assignment of section 3.2 (2) may be rewritten as:

$$\begin{array}{l} \{p\wedge MI_j\wedge GI\}\ S_1;\ \bar{u}:=\bar{x};\ \bar{v}:=\bar{y};\ T_1\ \{PI_j{:}at(pr)\wedge p_2\wedge GI\} \\ \\ \{p_2\wedge PI_j{:}after(pr)\wedge GI\}\ T_2;\ \bar{y}:=\bar{v};\ S_2\ \{q\wedge MI_j\wedge GI\} \end{array} \qquad (3)$$

Now write (1) in CSP-notation (remember that "p_2" belongs to the proof of the calling process:

$$P_i::\ldots\{p\}\ \langle S_1;\ P_j!(\bar{x},\bar{y})\rangle\ \{p_2\}\langle P_j?\bar{y};\ S_2\rangle\ \{q\}$$
$$P_j::\ldots\{MI_j\}\ \langle P_i?(\bar{u},\bar{v});\ T_1\rangle\{PI_j:at(pr)\}\ T\ \{PI_j:after(pr)\}\langle T_2;\ P_i!\bar{v}\rangle\{MI_j\} \qquad (4)$$

Thus we see that (3) states nothing but the cooperation test of the matching bracketed sections in (4), as $P_j!a \| P_i?z \equiv z:=a$

Next we turn to the interference freedom test of the process invariant. This will be illustrated using the program of fig.2, for which we will attempt to prove the invariance of the process invariant, PI_2, of process P_2.

```
process $P_0$
begin<$\{q_1\}$call $P_1.pr$>end

process $P_1$
proc pr begin<$\{q_2\}$call $P_2.pr_1$>end
begin skip end


$\{\neg b\}\ S_1\ \{c \wedge \neg b\}$
$\{true\}\ T\ \{b\}$
none of the $S_i$'s and T contain
when-statements.
$q_3 \equiv PI_2:at(\ell_1)$
```

```
process $P_2$
b,c,d,e : bool
proc $pr_1$ begin $\{p_1\}S_1$;  $\ell_2$: when b:$S_2$ end
proc $pr_2$ begin $\{p_2\}$
if d : $S_3$
| $\neg$d : $S_4$; $\ell_3$ : when e : $S_5$ end end end
begin b:=c:=false
    $\ell_1$:when c : T end
end
```

figure 2

The first waiting point is the statement, labeled ℓ_1. According to the when-rule, we need the validity of $q_3 \equiv PI_2{:}at(\ell_1)$. But, ℓ_1 being a waiting point, S_1 will be executed first and hence we are forced to show that this leaves q_3 valid: $\{p_1 \wedge q_3\}\ S_1\ \{q_3\}$. As S_1 is the sole statement that can be executed at this point, this is the only check to be made. In general, this implies that we have to introduce an **interference freedom** test as in [Owicki and Gries, 1976]. However, DP is a more complicated language than those that Owicki was concerned with. Firstly, had the guard, b, been true after execution of S_1, we would have had to check whether $\{p_1 \wedge q_3\}\ S_1;S_2\ \{q_3\}$. So in general, the statement-sequences over which to check the invariance of q_3 cannot be determined statically, since it depends on whether when-statements act as waiting points or not.
Secondly, in the interference freedom test we have to take the environment into account: we do have to check the invariance of q_3 over S_1, as there is a call pending, but not over the body of pr_2. Likewise, at the waiting point after('Init$_2$'), we need only take S_2 into account, as there cannot be any other calls pending. Hence we must express whether some procedure incarnation can exist.

Let pr.inc denote some incarnation of the common procedure pr, declared within some process P_m. This incarnation is called into existence by the acceptance of some external request, generated by the execution of a call-statement within another process. This call-statement on its turn may be part of a procedure body. Hence, a procedure-incarnation is (uniquely) characterized by a chain, $C_1, C_2, \ldots, C_k$, of call-statements: C_k calling the procedure P_m.pr; C_i $(1 \leq i \leq k)$ within the body of the procedure called by C_{i-1} and C_1 within the initial statement of some process (of course each C_j must be within a different process).

at('C_1'), at('C_2'), ..., at('C_k') is called the (complete) **calling-chain** determining or determined by pr.inc.

The question whether some incarnation can exist has thus been translated into the question whether there is an execution of the DP-program such that (some of) the constituent processes are (simultaneously) suspended at the locations specified by the calling chain determining the incarnation.

Hence an assertion must be found, characterizing this situation: the **calling-chain-assertion**.

If 'S' is some assertion within the program P, the pre-assertion of 'S', pre('S'), in a proof-outline of P, expresses in general that there is a computation-history leaving control in process P just before 'S'. A first approximation to the calling-chain-assertion might therefore be:

$$\text{pre}('C_1') \wedge \text{pre}('C_2') \wedge \ldots \wedge \text{pre}('C_k'), \qquad (\$)$$

i.e., the conjunction of the pre-assertions of the call's. As in the situation we are attempting to characterize, the call-statements in the calling-chain are actually being executed , the assertions in the corresponding calling chain assertion should indeed be the pre-assertions of the call-statement (i.e. the "p_2-assertion" of the external request rule), rather than the pre-assertions of the surrounding bracketed section.

There are however three refinements that must be incorporated in the definition. Firstly, the pre-assertions all encode **local** computation-histories, which therefore should be checked for compatibility. It should be checked whether the local histories are compatible with the global calling-history which is encoded in GI. Secondly, in the proof-outlines, no specific incarnations are considered, hence the assertions interspersed in the procedure bodies and in particular the pre-assertions of the various call-statements are canonical with respect to the various incarnations. In the calling chain assertions however, we do consider specific incarnations, so the assertions making up the calling chain assertion must be modified by substituting the formal parameters of each common procedure by the actual parameters of the corresponding call-statement (bearing in mind that the actual parameters of a call 'C_i' ($1<i\leq k$) may contain formal parameters of the procedure containing 'C_i', called by 'C_{i-1}'). Thirdly, GI is needed to check compatibility. However the conjunction of ($) with GI will in general be false, because ($) captures the situation that all processes on the chain are waiting for the call-statements to finish, but does not specify whether control is inside bracketed sections in which GI need not hold. This implies that the calling chain assertion must be extended to specify where control resides in the procedure-incarnation defined by the calling chain.

Hence:

Let at('C_1'), ..., at('C_k') be the calling chain defining the incarnation pr.inc of the common procedure pr and let p be some assertion within pr. Then the **calling chain assertion** chain(pr.inc,p) is given by:

$$\mathrm{pre}('C_1')\wedge\mathrm{pre}('C_2')[\bar{x}_1/\bar{u}_1]\wedge\mathrm{pre}('C_3')[\bar{x}_2/\bar{u}_2][\bar{x}_1/\bar{u}_1]\wedge\ldots\wedge$$
$$\mathrm{pre}('C_k')[\bar{x}_{k-1}/\bar{u}_{k-1}]\ldots[\bar{x}_1/\bar{u}_1]\wedge p[\bar{x}_k/\bar{u}_k]\ldots[\bar{x}_1/\bar{u}_1]\wedge GI$$

where (1) $\bar{x}_i$ denotes the actual parameter list of the call-statement 'C_i' and
(2) $\bar{u}_i$ the formal parameter list of the procedure called by 'C_i'.

With these calling-chain-assertions at hand, the DP interference freedom tests can be formulated more precisely.
In [Owicki and Gries, 1976] parallelism is introduced by the <u>cobegin</u>-statement:
<u>cobegin</u> $S_1 \parallel \ldots \parallel S_n$ <u>end</u>.
Execution of this statement causes the statements $S_1, S_2, \ldots, S_n$ to be executed in parallel. They require however, roughly speaking, that each assignment-statement is executed as an indivisible, atomic action. Hence, the possible execution-sequences of the cobegin-statement are all interleavings of the atomic statements within $S_1, \ldots, S_n$. Consequently, the interference freedom condition (basically) is, that any assertion, p, from the proof-outline of S_i must be left invariant by execution of any of the atomic actions, T, part of some statement, S_j, which can be executed in parallel with S_i (i.e. part of those S_j with $j \neq i$).

Stated in these general terms, the same principle applies to DP, provided the two key-notions, atomic actions and concurrently executing statements, can be rendered meaningful in a DP-context.
Starting with the last notion, it is clear that a syntactic constraint does not suffice: within the same DP-process, two statements, S_1 and S_2, which may be executed concurrently (t.i. whose operations may be interleaved), notation: $S_1 \parallel S_2$, presuppose at least one external request to the process. This fact is expressed by the validity of some calling-chain-assertion "tracing out" a computation leaving control at('S_1') or at('S_2'). To express this an auxiliary predicate, act(ℓ), is introduced, for any ℓ specifying a waiting point:

$\mathrm{act}(\ell)$ = PI : ℓ, if ℓ is a location within or after the initial statement,

= false , if there are no calling chains leading to the procedure containing ℓ,

= $\bigwedge_{\mathrm{pr.inc}}$ chain(pr.inc,PI:ℓ), if ℓ is in the body of procedure pr.

In this last clause, the conjunction ranges over all incarnations of pr.
Using this predicate, we can define

$\ell_1 \| \ell_2$ where (1) neither ℓ_1 nor ℓ_2 is of the form after(pr),

(2) ℓ_1 and ℓ_2 are not both of the form at(pr), and

(3) ℓ_1 and ℓ_2 are not both within the initial statement,

as follows

$\ell_1 \| \ell_2$ iff $\mathrm{act}(\ell_1) \wedge \mathrm{act}(\ell_2)$ holds.

Notice that this predicate is defined for location specifiers instead of statement names.
Validity of $\ell_1 \| \ell_2$ in some state, is interpreted as implying that there is a computation such that control is both at(ℓ_1) and at(ℓ_2). If $\ell_1 \equiv$ at('W'), (W≡<u>when</u> b:S <u>end</u>), then $(\ell_1 \| \ell_2) \wedge b$ denotes that in the above situation 'W' can be interleaved at ℓ_2.

Next, what are the atomic statements in DP?
From the semantics of DP, the points at which interleaving takes place, i.e. the points at which another operation can be started or resumed, are the waiting points. A complication arises because some of the waiting points are semantically defined, so we must carefully phrase what we mean by atomicity:

T is an atomic statement of the initial statement or procedure body S iff

(1) T is either a prefix of S (t.i. at('T') ≡ at('S')) or T starts with a <u>when</u>-statement

(2) T is either a postfix of S (t.i. after('T') ≡ after('S')) or T is followed in S by a <u>when</u>-statement

(3) no <u>when</u>-statement other than the one possibly following T acts as a waiting point.

Notice that the first 2 requirements are of a syntactical nature, but that the third requirement is a semantical one.

This definition simply states that for 'T' to be an atomic statement of a process P, (1)at('T') must be a point at which the process can start execution, or resume execution when coming from a waiting point, (2) after('T') must be a point at which P starts or resumes another operation and (3) control must not leave 'T' in between.

Before we can formulate the interference freedom conditions, there are some technicalities to be dispensed with. Firstly, up to now we have implicitly assumed that interference freedom need not be checked for each incarnation separately (cf. the definition of $\ell_1 \| \ell_2$ above,, in which we do not ask for the existence of a particular incarnation). This is indeed correct as the discussion in section 3.2 indicated the possibility of specifying the input-output behaviour of some procedure in a canonical way, i.e., indicated the possibility of giving proof outlines for the procedure bodies. A problem arises here, however, as the formal parameters of a procedure are **local** to that procedure, so that there are name-clashes to be avoided.

Secondly, the proof outline of a procedure-body does not extend to the prelude of that body and GI does not hold in the prelude, so that we can't use the $\|$-predicate there. However, the assertion to be proven invariant over the procedure-body may well contain variables which are changed by the prelude so it must be taken into account.

Regarding the first problem, it suffices to substitute fresh variable-names for the possible formal parameters in an assertion. As to the second problem, we introduce strongest post-conditions (w.r.t. partial correctness). I.e., if assertion p is to be invariant over some procedure body $T_1;>T;<T_2$ ($\{p\}T_1;T;T_2\{p\}$) we will check instead whether $\{p\rightarrow T_1\}T;T_2\{p\}$, where $p\rightarrow T_1$ is the strongest assertion s.t. $\{p\}T_1\{p\rightarrow T_1\}$. Using such conditions does not complicate matters unduly, because a prelude typically consists of one or two assignments.

The following definition formalizes these ideas:

Let p be an assertion within the proof outline of process P. Let $\vec{x}$ be the (possibly empty) set of formal parameters within p and $\vec{y}$ a set of fresh variables (of the same cardinality of $\vec{x}$). Furthermore, let 'S' denote some statement of P.

Then $\overline{p} \equiv p[\vec{y}/\vec{x}]$ and $p('S') \equiv \overline{p} \rightarrow R$ where

$R \equiv T_1$, if 'S' denotes a statement in a procedure body $T_1;>T<;T_2$ and

$T;T_2 \equiv S$, or $T;T_2 \equiv S;\ T'$ or $T;T_2 \equiv T';S$ for some statement T',

<u>skip</u>, otherwise.

Now let ℓ be some location specifier at('W') or after(Init), 'T' a statement which obeys the syntactical constraints of our definition of atomicity, and atom('T') the assertion enforcing the semantic constraints. Then the interference freedom of 'T' w.r.t. PI : ℓ is expressed as

$$\{PI{:}\ell('T') \wedge PI{:}at('T') \wedge atom('T') \wedge act(\ell)('T') \wedge act(at('T'))\}T\{\overline{PI{:}\ell}\}$$

If this is applied to the interference freedom test of q_3 in the above example, the following conditions must be checked:

(1) $\{q_3 \wedge p_1 \wedge (S_1 \rightarrow \neg b) \wedge q_1 \wedge q_2 \wedge GI\}\ S_1\ \{q_3\}$(*)

(2) $\{q_3 \wedge p_1 \wedge (S_1 \rightarrow b) \wedge q_1 \wedge q_2 \wedge GI\}\ S_1$ <u>when</u> $b{:}S_2$ <u>end</u> $\{q_3\}$

(3) $\{q_3 \wedge PI_2 : at(\ell_2) \wedge b \wedge q_1 \wedge q_2 \wedge GI\}$ <u>when</u> $b{:}S_2$ <u>end</u> $\{q_3\}$

(4) $\{q_3 \wedge p_2 \wedge (\neg d \supset (S_4 \rightarrow e)) \wedge false\}$ <u>if</u> ... <u>end</u> $\{q_3\}$

(5) $\{q_3 \wedge p_2 \wedge \neg d \wedge (S_4 \rightarrow \neg e) \wedge false\}\ S_4\ \{q_3\}$

(6) $\{q_3 \wedge PI_2 : at(\ell_3) \wedge e \wedge false\}$ <u>when</u> $e{:}S_5$ <u>end</u> $\{q_3\}$

Of course, the last three tests are trivially satisfied, but they are interesting because they clearly show that the atomic statements do not follow the phrase-structure of the language: One branch of the <u>if</u>-statement in the body of pr_3 (fig. 2) contains a <u>when</u>-statement, and it is necessary to "break open" this <u>if</u>-statement to consider each branch separately (tests (5) and (6)). The assertion atom(...), must force the correct branch to be taken in such cases.

We see that a direct analogue of Owicki's test (i.e., listing all atomic statements separately) is somewhat awkward here. In fact, the situation is worse because in the

----

(*) $R \rightarrow s$ denotes the weakest pre-condition (w.r.t. partial correctness) s', such that {s'} R {s} holds. This dynamic logic approach was suggested by Amir Pnueli.

presence of loops, there may even be an infinity of atomic statements. So, we look for an alternative formulation in which interference freedom w.r.t. some assertion PI : ℓ is expressed for all atomic actions within a procedure body or initial statement S in a single step. This will appear to be straightforward.

Consider a proof for

$$\{PI{:}\ell('S')\wedge PI{:}at('S')\wedge act(\ell)('S')\wedge act(at('S'))\}S\{\overline{PI{:}\ell}\}.$$

This takes care of the invariance of PI : ℓ over S. However S may contain atomic statements too. Consider such an atomic statement T within S, starting with resp. followed (in S) by when-statements W_1 resp. W_2. If T can be interleaved at(ℓ), i.e., if W_1 and W_2 can become blocked at(ℓ), PI : ℓ must be invariant over T. The pre-assertions of W_1 and W_2, in case they blocked, are PI : at(W_1) and PI : at(W_2). This suggest changing PI (for this particular proof) such that PI : at(W_i) $\rightarrow$ PI : ℓ; whence the following definition of the auxiliary predicate, INV(p,'S'), expressing the interference freedom of 'S' w.r.t. the assertion p.

Let 'S' be some statement and p an assertion associated with a statement denoted by the location specifier ℓ. Then INV(p,'S') is true iff a (valid) proof can be constructed for $\{p('S')\wedge PI{:}at('S')\wedge act(\ell)('S')\wedge act(at('S'))\}S\{\overline{p}\}$ such that the process invariant PI', used when applying the when-rule in this proof, is the following

$$PI'\equiv\bigwedge_{'W_i'\text{ in }'S'} at('W_i'){:}(\overline{p}\wedge PI{:}at('W_i')\wedge act(\ell)\wedge act(at('W_i')))$$

Using this predicate, our final interference freedom test is straightforwardly formulated as:

Interference freedom test:

Let process P have procedures $pr_1,\ldots,pr_n$ and a process invariant PI. Assume we have a proof outline (w.r.t. PI and GI) for P. For any 'S' in P_1 let pre('S') denote the (sequential) pre-assertion of 'S' in P's proof-outline. Suppose we have a predicate INV(p,'S') asserting the invariance of an assertion p over all atomic statements within 'S'. Then PI is defined to be **interference free** w.r.t. a proof outline of P iff

(1) for each 'term' in PI of the form $at('W_i'){:}p_i$ the formula

$INV(p_i,pr_1.body)\wedge\ldots\wedge INV(p_i,pr_n.body)$ holds,

(2) for each term in PI of the form $at('W_i'){:}p_i$, $'W_i'$ within some procedure-body

$INV(p_i, Init_p)$ holds, and

(3) for the possible term $after(init_p):q$ in PI, $INV(q,pr_1.body) \wedge \ldots \wedge INV(q,pr_n.body)$ holds.

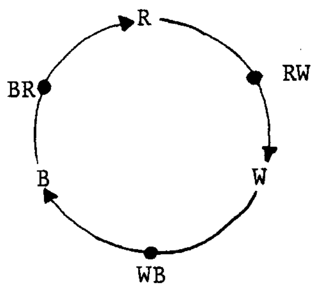
The assertion associated with each when-statement in 'S' by PI' in the INV predicate is rather formidable. On the other hand, consider such a when-statement 'W_i'. By the ordinary proof outline, we know that PI : at('W_i') holds when control arrives at('W_i'). The disjunction $act(\ell) \wedge act(at('W_i'))$ remains valid because of the substitution of the formal parameters in the calling chain assertion. So effectively, only validity of $\bar{p}$ has to be proven.

They are, however, all needed because every time we must check if an interleaving is possible. If it is not possible, PI' : at('W_i') reduces to <u>false</u> and the test is trivialized. Notice that this does <u>not</u> imply that subsequent tests are trivialized too, because the when-rule rule forces us to use PI' : at('W_j') as a pre-assertion for a subsequent statement 'W_j'.

6. THE DUTCH NATIONAL TORUS.

Edsger W. Dijkstra designed the following problem, "inspired by the Dutch National Flag" [Dijkstra, 1977]:

Consider a program comprising 3 + 3 processes in a cyclic arrangement; three main processes, called R(ed), W(hite) and B(lue), and three buffer processes RW, WB and BR, in between:



Each main process contains initially a number of red, white and blue pebbles. Its goal is to collect from the system all pebbles of its own colour, thus constructing a neatly coloured Dutch National Torus. Communication takes place along the directed channels as indicated in the figure. In particular, process B cannot communicate directly with process W to transmit its white pebbles. Moreover, per communication (at most) one pebble can be transmitted.

Below, we will give a DP version of Dijkstra's solution. Readers who want to see more of the original draft are referred to [Dijkstra, 1977].

The solution, as proposed in Dijkstra, implemented in DP.

The program contains three main processes R(ed), W(hite) and B(lue) respectively, and three buffer processes RW, WB and BR, in between.

Each buffer process is synchronized with its left neighbour for the input of pebbles. After each single input the process is synchronized with the main process at its righthand side, to get rid of the pebble again. The private variable y takes care of buffering the pebble. The main process on the left can only have its pebble buffered in y when y contains no other pebble; this is controlled by means of a <u>when</u>-statement. Each main process transmits the value ω to its right buffer process when it has transmitted all foreign coloured pebbles, to signal that the buffer process may terminate. A specific buffer process BR is as follows:

```
process BR
y:pebble; trm:bool;
proc p(x:pebble)
   when y=ε:y:=x end
begin
   y:=ε; trm:=false
do ¬trm:
   when y≠ε:call R.p(y); trm:=y=ω; y:=ε end
end
```

Each main process has three bags of pebble, r(red), w(hite) and b(lue) initialized to resp. r_0, w_0 and b_0 (the pebbles themselves are denoted by r, w and b). Its goal is to collect from the system all pebbles of its own colour (in the correspondingly coloured bag). Its foreign pebbles it transmits, one at a time, via the buffer process on its right, thereby first emptying the bag that contains the pebbles with the larger travelling distance. When it has no more foreign pebbles left, it gives a signal ω, via the same buffer process (this introduces a starting problem, which will be discussed on the basis of the program text). Communication takes place in a lock-stepped fashion: after the process has sent a pebble to its right buffer process, it waits until its left buffer process has a new pebble to be submitted. When no more pebbles are to be submitted, expressed by the variable "term", the main process just empties its foreign coloured bags of pebble. This emptying takes place in a smooth way, by the set up of the communication: the right buffer process only finishes the request when it has already sent the former pebble to the next main process. Thus, no pebble vanishes. A variable "acc" keeps count of the last communication action of the process: acceptance or output of a pebble. Hence, our translation of a particular main process R (if g is a bag of pebbles, the cardinality of g is denoted by #g):

```
process R
r,w,b:bag_of_pebble; acc,term:bool
proc p(x:pebble)
   if x=ω: term:=true
      x≠ω: "place x in appropriate bag"; acc:=true
   end
begin
   r,w,b:=r0,w0,b0; acc,term:=true, false;
   do #b>0:
       b-:=b; acc:=false; call RW.p(b);
       when termvacc:skip end
   end;
   do #w>0:
       w-:=w; acc:=false; call RW.p(w);
       when termvacc:skip end
   end;
   call RW.p(ω)
end
```

The starting problem.

Unfortunately, the program lined out above does not fullfill its purpose for every input. Consider an initial division of pebbles such that process R has no white and blue pebbles. Execution in R starts with Init_R, where the loop statements are skipped because their guards are false. Consequently, Init_R terminates before external requests to procedure p_R can be honoured. Such a request may introduce a white pebble in R (as process B initially may have a white pebble). This pebble will never be sent to its collector process W, since sending only takes place in Init_R, which has terminated.

Clearly, the problem is that Init_R is allowed to terminate without checking whether it has to pass on pebbles from its left neighbour. This suggests, inserting just before the two guarded loops, the statement:

if $\#b_0+\#w_0=0$: when termvacc:skip end | $\#b_0+\#w_0\neq 0$:skip end end.

An easier solution, not modifying the elegant algorithm, is demanding that each main process contains at least one foreign pebble. This is what Dijkstra proposed and we shall follow in his footsteps.

Correctness proof of the Dutch National Torus.

First some conventions and notation are introduced.

(1) To avoid name-clashes, the variables of each process are indexed by the process' name. However, this index will not be written if it is clear from the context, which process a variable belongs to.

(2) The notation "#b(h)" is shorthand for "the number of blue pebbles in h" (where h is a bag, sequence, set etc.), "$\#(b_R+w_R)$" for "the number of blue and white pebbles in R" and so on.

(3) "hd(h)" denotes the head element of the sequence h; "tl(h)" denotes the rest of the sequence.

(4) "$r^* w^+ (\varepsilon+\omega)$" denotes a sequence of zero or more red pebbles, followed by at least one white pebble and possibly closed-off by an ω. (ε denotes the empty sequence).

(5) "h^k" denotes the concatenation of the two sequences h and k.

We want to prove the following Hoare formula:

$$\{\#(b_{OR}+w_{OR})>0 \wedge \#(r_{OW}+b_{OW})>0 \wedge \#(r_{OB}+w_{OB})>0 \wedge \bigwedge_{(k,K)\in\{(r,R),(w,W),(b,B)\}} \#k_{OK} \geq 0\}$$

$$[R \| RW \| W \| WB \| B \| BR]$$

$$\{\bigwedge_{(k,K)\in\{(r,R),(w,W),(b,B)\}} (\#k_K = \#(k_{OR}+k_{OW}+k_{OB})) \wedge \#(w_R+b_R)+\#(r_W+b_W)+\#(r_B+w_B)=0\}$$

To express the necessary assertions, some auxiliary variables are introduced. For each process we introduce variables h and $\overline{h}$ which keep track of the pebbels having been received resp. dispatched by the process. Using these, the fact that no pebble is lost during execution can be expressed. Next, consider a buffer process. With a little thought, it becomes clear that such a buffer has a maximal lag of 2 between the pebbles received and the pebbles already sent off. Therefore, an auxiliary variable z is introduced, remembering the pebble received last, as long as this pebble has not yet been put into the buffer-variable y. This is linked with the fact that the (last) call to the buffer, having transmitted the pebbly now in z, has not been completed yet. To express this link, each main process gets an auxiliary boolean l, expressing the fact that no call (originating from this process) is in progress.

The general invariant now becomes:

$$GI \equiv \bigwedge_{KL\in\{RW,WB,BR\}} (\bar{h}_{KL}=h_L \wedge \bar{h}_K=h_{KL} \wedge (z_{KL}=\varepsilon \supset 1_K))$$

Proof outline of process BR.

The following monitor and process invariants are used

$MI_{BR} \equiv h_{BR}=\bar{h}_{BR}\,\hat{}\,y_{BR}\,\hat{}\,z_{BR} \wedge (trm_{BR} \rightarrow hd(\bar{h}_{BR})=\omega) \wedge h_{BR} \in w^* r^* (\varepsilon+\omega) \wedge (\omega \in h_{BR} \rightarrow \# h_{BR} \geq 2)$

$PI_{BR} \equiv$ $at(BR.p):(z_{BR}=x_{BR} \wedge MI_{BR}) \wedge after(BR.p):\ MI_{BR}[\varepsilon/z_{BR}] \wedge$

$at(Init_{BR}):(z_{BR}=\varepsilon \wedge h_{BR}=\bar{h}_{BR}=\varepsilon) \wedge after(Init_{BR}):(MI_{BR} \wedge trm_{BR} \wedge y_{BR}=\varepsilon) \wedge$

$at('W'):(z_{BR}=x_{BR} \wedge MI_{BR})$

The first term of MI_{BR} just expresses that BR is a buffer (taking account of the lag). The second term is needed to prove that no calls occur once the value ω has been been received. The last two terms describe the behaviour of processes B (as seen from BR). This is needed to prove the calls to process R correct.

Here follows the proof outline:

```
process BR
   y,z:pebble; h,h̄:seq of pebble; trm:bool
   proc p(x:pebble) {MI}
   begin h^:=x; z:=x>{z=x∧MI}
     'W':when y=ε:{MI∧z=x∧y=ε}
             {MI[ε/y, x/z]} y:=x
           end {MI[ε/z]}
   <z:=ε end {MI}

   {z=ε∧h=h̄=ε}
   begin y:=ε; trm:=false {MI∧y=ε}
     do   trm:{MI∧y=ε}
          when y≠ε:{MI∧y≠ε}
             <h̄^:=y; {MI[ε/y, y=ω/trm]} call R,p(y)>
             trm:=(y=ω); y:=ε
          end {MI∧y=ε}
     end
   end {MI∧y=ε∧trm}
```

The proof outlines and invariants for the other buffers are similar and will not be spelled out.

<u>Proof outline of process R.</u>

The following monitor invariant is used:

$$MI_R \equiv \bar{h}_R \in b^* w^* (\varepsilon+\omega) \wedge ((term_R \vee acc_R) \rightarrow \# h_R \geqq \min(1, \# \bar{h}_R)) \wedge (\neg acc_R \rightarrow \# h_R \geqq \min(1, \# \bar{h}_R - 1)) \wedge$$
$$((acc_R \wedge \# w_R = 0 \wedge \# h_R \geqq 1) \rightarrow hd(h_R) \neq w) \wedge (term_R \rightarrow hd(h_R) = \omega) \wedge \#(b_{0R} + w_{0R}) > 0 \wedge$$
$$\# b(\bar{h}_R) + \# b_R = \# b_{0R} \wedge \# w(\bar{h}_R) + \# w_R = \# w_{0R} + \# w(h_R) \wedge \# r_R = \# r_{0R} + \# r(h_R)$$

The first term describes the sequence of pebbles that the process transmits. The next two terms are necessary to prove that after termination of the process, at least one pebble has been received. Together with the fourth and fifth term, they are used to prove that after termination no white pebbles can be received. The last three terms are important and describe the relationship between the pebbles within the bags and the pebbles received resp. transmitted at any time.

The process invariant is as follows:

$$PI_R \equiv at(R.p):(hd(h_R) = x_R \wedge MI_R[tl(h_R)/h_R] \wedge h_R \in w^* r^* (\varepsilon+\omega) \wedge \# h_R \geqq \min(1, \# \bar{h}_R)) \wedge after(R.p): MI_R \wedge$$
$$at(Init_R):(\# b_{0R} + \# w_{0R} > 0 \wedge \# b_{0R} \geqq 0 \wedge \# w_{0R} \geqq 0 \wedge l_R \wedge h_R = \bar{h}_R = \varepsilon) \wedge$$
$$after(Init_R):(MI_R \wedge \# b_R + \# w_R = 0 \wedge hd(\bar{h}_R) = \omega \wedge (term \vee acc) \wedge l_R \wedge \# \bar{h}_R \geqq 2) \wedge$$
$$at('W_1'):(MI_R \wedge \bar{h}_R \in b^* \wedge l_R \wedge LI_{R1}) \wedge at('W_2'):(MI_R + \# b_R = 0 \wedge \bar{h}_R \in b^* w^* \wedge l_R \wedge (\# b_{0R} > 0 \rightarrow b \in \bar{h}_R) \wedge LI_{R2})$$

In the proof outline, the following two loop invariants are used:

$$LI_{R1} \equiv \exists k (\# b_R = \# b_{0R} - k \wedge (k > 0 \rightarrow b \in \bar{h}_R))$$
$$LI_{R2} \equiv \exists k (\# w_R = \# w_{0R} - k \wedge (k > 0 \rightarrow w \in \bar{h}_R))$$

These are needed to show that at least one foreign pebble will be transmitted by the process.

Now follows the proof outline of process R (the other proof outlines and corresponding invariants are analogous):

```
process R
  r,w,b:bag of pebble; inc,term,l:bool; h,$\bar{h}$:seq of pebble
  proc p(x:pebble) $\{MI\}$
  begin h^:=x;>$\{hd(h)=x\wedge MI[tl(h)/h]\wedge h\in w^{*}r^{*}(\varepsilon+\omega)\wedge\#h\geq\min(1,\#\bar{h})\}$
    if x=ω:term:=true | x≠w:"place in appropriate bag"; acc:=true end
  $\{MI\}$<end $\{MI\}$

  $\{\#(r_0+w_0)>0\wedge l=true\wedge h=\bar{h}=\varepsilon\}$
  begin r,w,b:=$r_0,w_0,b_0$; acc,term:=true, false; $\{MI\wedge\bar{h}\in b^{*}\wedge l\wedge LI_1\wedge(term\vee acc)\}$
    do #b>0:b-:=b; acc:=false; $\{MI[\bar{h}\hat{}b/\bar{h}]\wedge\bar{h}\in b^{*}\wedge l\wedge LI_1[\#b-1/\#b]\wedge\neg acc\}$
         <l:=false; $\bar{h}$^:=b; $\{MI\wedge\bar{h}\in b^{+}\wedge\neg l\wedge LI_1\}$ call RW.p(b); l:=true>;
         $\{MI\wedge\bar{h}\in b^{+}\wedge l\wedge LI_1\}$ 'W$_1$':when term∨acc:skip end
    end; $\{MI\wedge\#b=0\wedge\bar{h}\in b^{*}w^{*}\wedge l\wedge(\#b_0>0\rightarrow b\in\bar{h})\wedge LI_2\wedge(term\vee acc)\}$
    do #w>0:w-:=w; acc:=false; $\{MI[\bar{h}\hat{}w/\bar{h}]\wedge\bar{h}\in b^{*}w^{*}\wedge l\wedge LI_2[\#w-1/\#w]\wedge\#b=0\wedge(\#b_0>0\rightarrow b\in\bar{h})\}$
       <l:=false; $\bar{h}$^:=w; $\{MI\wedge\bar{h}\in b^{*}w^{+}\wedge\neg l\wedge LI_2\wedge\#b=0\wedge(\#b_0>0\rightarrow b\in\bar{h})\}$ call RW.p(w); l:=true>;
       $\{MI\wedge\bar{h}\in b^{*}w^{+}\wedge l\wedge LI_1\wedge\#b=0\wedge(\#b_0>0\rightarrow b\in\bar{h})\}$ 'W$_2$':when term∨acc:skip end
    end; $\{MI\wedge\#(b+w)=0\wedge\bar{h}\in b^{*}w^{*}\wedge l\wedge\#\bar{h}\geq 1\wedge(term\vee acc)\}$
    <l:=false; $\bar{h}$^:=ω; call RW.p(ω); l:=true>
  end $\{MI\wedge\#(b+w)=0\wedge l\wedge hd(\bar{h})=\omega\wedge\bar{h}\geq 2\wedge(term\vee acc)\}$
```

Before the parallel composition rule can be applied, we need to check for cooperation (i.e. we need to check whether the external request rule has been properly used) and for interference freedom. First the

Cooperations tests:

We only check the call in process BR and the first call in process R; the other ones are analogously proven. For each call we have to prove the three premisses of the external request rule

A. call R.p(y) in BR:

1. $\{MI_{BR}\wedge y_{BR}\neq\varepsilon\wedge MI_R\wedge GI\}$

   $\bar{h}_{BR}\hat{}:=y_{BR};\ h_R\hat{}:=y_{BR}$

   $\{MI_{BR}[\varepsilon/y_{BR},\ y_{BR}=\omega/trm_{BR}]\wedge hd(h_R)=y_{BR}\wedge MI_R[tl(h_R)/h_R]\wedge h_R\in w^{*}r^{*}(\varepsilon+\omega)\wedge$
   $\#h_R\geq\min(1,\#h_R)\wedge GI\}$

   the relevant part of GI is $\bar{h}_{BR}=h_R$, so GI is trivially invariant;

   the transformation of the BR-assertions is easily worked out, similar for the rest.

2. Canonical proof of the body of R.p: see proof outline

3. $\{MI_{BR}[\varepsilon/y_{BR}=\omega/trm_{BR}]\wedge MI_R\wedge GI\}$

   skip

   { idem }

   evident

B. <u>call</u> RW.p(b) in R:

1. $\{MI_R[\overline{h}_R\hat{\ }b/\overline{h}_R]\wedge\overline{h}_R\in b^*\wedge l_R\wedge LI_{R1}[\#b_R-1/\#b_R]\wedge\neg acc\wedge MI_{RW}\wedge GI\}$

   $l_R:=\underline{false};\ \overline{h}_R\hat{\ }:=b;\ h_{RW}\hat{\ }:=b;\ z_{RW}:=b$

   $\{MI_R\wedge\overline{h}_R\in b^+\wedge\neg l_R\wedge LI_{R1}\wedge\neg acc\wedge MI_{RW}\wedge z_{RW}=b\wedge GI\}$

   relevant part of GI:$\overline{h}_R=h_{RW}\wedge(z_{RW}=\varepsilon\rightarrow l_R)$ which is clearly invariant; transformation of R-assertions: clear

   RW-assertions: we only need to check the invariance of MI_{RW}:

   beforehand $z_{RW}=\varepsilon\wedge h_{RW}=\overline{h}_{RW}\hat{\ }y_{RW}$, hence afterwards: $h_{RW}=\overline{h}_{RW}\hat{\ }y_{RW}\hat{\ }z_{RW}$;

   also $\overline{h}_R=h_{RW}$(GI) and $\overline{h}_R\in b^+$ afterwards, so $h_{RW}\in b^+$ afterwards,

   hence $h_{RW}\in b^*\mathit{w}^*(\varepsilon+\omega)\wedge\omega\notin h_{RW}$; trm_{RW} and $\overline{h}_{RW}$ are left unchanged

2. Canonical proof of the body of R.p: see proof outline

3. $\{MI_R\wedge\overline{h}_R\in b^+\wedge\neg l_R\wedge LI_{R1}\wedge\neg acc\wedge MI_{RW}[\varepsilon/z_{RW}]\wedge GI\}$

   $z_{RW}:=\varepsilon;\ l_R:=\underline{true}$

   $\{MI_R\wedge\overline{h}_R\in b^+\wedge l_R\wedge LI_{R1}\wedge\neg acc\wedge MI_{RW}\wedge GI\}$

   cf. the definition of GI

<u>Interference freedom tests</u>.

As expected, only the buffer process BR and the main process R are checked.

A. process BR

1. PI:at('W'). Invariance over $Init_{BR}$ is trivial, because

   FV(PI:at('W'))∩FV(Init)=∅

   Invariance over BR.p is trivial too on syntactic grounds: no two calling chains can coexist.

2. PI:after(Init). We show that no incarnation of BR.p can exist once

   BR's initial statement is terminated. The only global information that is needed, is that the actual parameter of a call to BR.p is either w, r or ω. Hence

   $act(at(BR.p))\wedge act(after(Init_{BR}))(BR.p)\rightarrow$
   $MI\wedge z\in(\mathit{w}+\mathit{r}+\omega)\wedge trm\wedge y=\varepsilon\rightarrow$
   $Z\in(\mathit{w}+\mathit{r}+\omega)\wedge h=\overline{h}\hat{\ }z\wedge hd(\overline{h})=\omega\wedge h\in\mathit{w}^*\mathit{r}^*(\varepsilon+\omega)\rightarrow false.$

   As $at(BR.p)\equiv at('W_{BR}')$, this takes care of the second test.

B. process R

1. PI:at('W_i') (i=1,2). The invariance checks over R.p are trivial and are left to the reader. Here too, no global information is needed.

2. PI:after(Init). To check invariance over R.p no global information is needed.

   Hence we prove

   $\{hd(h)=x\wedge MI[tl(h)/h]\wedge h\in\omega^*\mathit{r}^*(\varepsilon+\omega)\wedge\#b+\#w=0\wedge hd(\bar{h})=\omega\wedge(term\vee acc)\wedge l\wedge\#\bar{h}\geq 2\}$

   $\underline{if}$ $x=\omega$:term:=$\underline{true}$|$x\neq\omega$:"place in appropriate bag"; acc:=$\underline{true}$ $\underline{end}$

   $\{\#b+\#w=0\wedge hd(\bar{h})=\omega\wedge(term\vee acc)\wedge l\wedge\#\bar{h}\geq 2\}$.

   Only "$\#b+\#w=0$" and "$term\vee acc$" can be affected by execution of R.p. Invariance of the latter term is trivial, so we concentrate on the first one.

   First assume term holds:

   In that case $MI[tl(h)/h]\wedge\#h\geq 2\rightarrow hd(tl(h))=\omega$, so the pre-assertion implies $hd(tl(h))=\omega\wedge h\in\omega^*\mathit{r}^*(\varepsilon+\omega)$ which is false.

   Assume $\neg$term holds:

   Then acc must hold. As $\#w=0$, $MI[tl(h)/h]\wedge\#h\geq 2\rightarrow hd(tl(h))\in\mathit{r}+\omega$.

   The pre-assertion implies that $h\in\omega^*\mathit{r}^*(\varepsilon+\omega)$, hence $hd(h)\in\mathit{r}+\omega$.

   So after termination, we still have $\#b+\#w=0$.

This concludes the interference freedom tests and the parallel composition rule may safely be applied:

$$\{\#(b_{OR}+w_{OR})>0\wedge\#(r_{OW}+b_{OW})+\#(r_{OB}+w_{OB})>0\wedge\bigwedge_{(k,K)\in\{(r,R),(w,W),(b,B)\}}\#k_{OK}\geq 0\wedge GI\}$$

$$[R\|RW\|\ldots\|B\|BR]$$

$$\{\bigwedge_{(k,K)\in\{(r,R),(w,W),(b,B)\}}(\#k_K=\#(k_{OR}+k_{OW}+k_{OB}))\wedge\#(w_R+b_R)+\#(r_W+b_W)+\#(r_B+w_B)=0\wedge GI\}$$

Derivation of the required post-assertion:

$(\bigwedge_{K\in\{R,W,B\}}PI_K:after(Init_K))\rightarrow\#(w_R+b_R)=0\wedge\#(b_W+r_W)=0\wedge\#(w_B+r_B)=0$,

whence the second part of the post-assertion.

Of the first part of the post-assertion, we will only show $\#r_R=\#r_{OR}+\#r_{OW}+\#r_{OB}$:

$MI_R\rightarrow\#r_R=\#(r_{OR}+r(h_R))$ (1)

we show that $\#r(h_R)=\#(r_{OW}+r_{OB})$: (2)

$GI\rightarrow h_R=\bar{h}_{BR}\wedge h_{BR}=\bar{h}_B\wedge(l_B\rightarrow z_{BR}=\varepsilon)$ (3)

$PI_B:after(Init_B)\rightarrow l_B$ (4)

$PI_{BR}:after(Init_{BR})\rightarrow y_{BR}=\varepsilon\wedge MI_{BR}$ (5)

from (3), (4) and (5) follows $h_R=\bar{h}_B$, hence $\#r(h_R)=\#r(\bar{h}_B)$

PI_B:after($Init_B$)$\rightarrow MI_B \wedge \#r_B=0$, or $\#r(\bar{h}_B)=\#(r_{0B}+r(h_B))$

which turns the last result into $\#r(h_R)=\#(r_{0B}+r(h_B))$

Remains to show $\#r(h_B)=\#r_{0W}$:

a similar reasoning as in (3), (4), (5) for the processes W and WB shows that

$\#r(h_B)=\#r(\bar{h}_W)$.

PI_W:after($Init_W$)$\rightarrow MI_W \wedge \#r_W=0$, or $\#r(\bar{h}_W)=\#r_{0W}$, whence $\#r(h_B)=\#r_{0W}$

Finally $(1)\wedge(2)\rightarrow \#r_R=\#r_{0R}+\#r_{0W}+\#r_{0B}$.

Using the consequence, substitution and AV-rule, allows us to remove all unnecessary information and the auxiliary variables, thus concluding the proof.

If this proof is compared with the original proof in [Dijkstra, 197], the reader undoubtedly will notice the greater length and complexity of our proof. He should however bear in mind that ours is completely formal, whereas Dijkstra's proof is based on informal (albeit sound) reasoning.

REFERENCES

[Apt et al, 1980]
Apt, K.R., N.Francez, W.P. de Roever, A Proof System for Comunicating Sequential Processes, TOPLAS 2-3, p359-385, 1980.

[Brinch Hansen, 1975]
Brinch Hansen, P., The programming language Concurrent Pascal, IEEE TSE 1, p99-207, 1975.

[Brinch Hansen, 1978]
Brinch Hansen, P., Distributed Processes: A Concurrent Programming Concept, CACM 21-11, p.934-941, 1978.

[de Bakker, 1980]
de Bakker, J., Mathematical Theory of Program Correctness, Prentice Hall, 1980

[Dijkstra, 1976]
Dijkstra, E.W., A Discipline of Programming, Prentice Hall, 1976.

[Dijkstra, 1977]
Dijkstra, E.W., An elephant inspired by the Dutch national flag, EWD 608, 1977.

[Hoare, 1969]
Hoare, C.A.R., An Axiomatic Basis for Computer Programming CACM 12-10 p.576-583, 1969.

[Hoare, 1974]
Hoare C.A.R., Monitors, an Operating System Structuring Concept, CACM 17-10, p.549-557, 1974.

[Hoare, 1978]
Hoare C.A.R., Communicating Sequential Processes, CACM 21-8, p.666-677, 1978

[Ichbiah et al, 1979]
Ichbiah, J. et al, Rationale for the design of the ADA programming language, SIGPLAN notices 14(6) part B, 1979.

[Howard, 1976]
Howard, J.H., Proving Monitors, CACM 19-5, p.273-279, 1976.

[Lamport, 1980]
Lamport, L., The Hoare´s Logic of Concurrent Programs, Acta Inf. 14, p.21-37, 1980.

[Lampson et al, 1980]
Lampson, B.W., D.D. Redell, Experience with Processes and Monitors in MESA, CACM 23-2, p105-117, 1980.

[Owicki and Gries, 1976]
Owicki, S., D. Gries, An Axiomatic Proof Technique for Parallel Programs I, Acta Inf. 6, p.319-340, 1976

[Roncken et al, 1981]
Roncken, M., N. van Diepen, M. Kramer, W.P. de Roever, A Proof System for Brinch Hansen´s Distributed Processes, Technical Report RUU-CS-81-5, Dept. of C.S., University of Utrecht, 1981

[Wirth, 1980]
Wirth, N., Modula2, Report 36, Institut für Informatik, ETH, Zürich, 1980.

1. A SUMMARY OF BRINCH HANSEN'S LANGUAGE DISTRIBUTED PROCESSES (DP)

A DP-program consists of a fixed number of persistent sequential processes that are executed concurrently. We assume the processes to be numbered, hence a DP-program is denoted as $[P_1 \| \ldots \| P_n]$, n>2. A process does not contain parallel statements and can access its own private variables only (i.e. there are no shared variables in DP). The only form of process communication is calling a **common procedure** declared within another process. Such an **external request** is honoured when the other process arrives at a **waiting point**, waiting for some condition to become true.
The syntax of a DP-process is as follows:

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

1.1. Process execution

A process starts by executing its initial statement. This continues until the statement either terminates or waits for a condition to become true. Then another operation may be started by honouring an external request. In that case a (fresh) **procedure incarnation** is created and executed. When this operation terminates or waits, the process will either begin yet another operation (by honouring an external request) or will resume an earlier operation (as result of a condition having become true). This (coroutine-like) interleaving of the initial statement and the procedure incarnations continues forever. A process continues to execute operations except when all its operations are delayed at waiting points or when it makes an external request. In the first case the process remains idle until it receives an external request; in the second case, the process is idle until the callee has completed the requested operation. Apart from this nothing is assumed about the order in which a process performs its operations.

1.2. Common procedures

Syntax of a common procedure (*)

```
proc<name> (<input parameter list>#<output parameter list>)
begin <statement> end
```

External request of P to a common procedure qr declared in process Q :

```
call Q.qr(<expression list>,<variable list>),
```

where the variables in the expression and the variable lists must be private to P.

The parameter transfer mechanism is the following: Before executing the procedure-body, the expression-values of the call are assigned to the input parameters. Afterwards, the output-values are assigned to the variables of the call.

As in the above situation, execution of the calling process, P, is suspended until the callee, Q, has completed the request, recursion (be it direct or indirect) is disallowed as it would lead to deadlock (**).

(*) In DP, as defined in [Brinch Hansen, 1978], procedures may have local variables. We have disallowed this in order to concentrate on the concurency-features of DP, without being distracted too much by sequential language aspects already dealt with elsewhere, cfr. [de Bakker, 1980]. See also section 3.2 p.9

(**) Although we might have allowed direct recursion, as a process calling one of its own common procedures does not result in process-communication

1.3. Sequential language constructs

The sequential constructs of DP are the Dijkstra **guarded commands** (Dijkstra, 1976]:

(1) **guarded conditional:** if $b_1 : S_1 | \ldots | b_n : S_n$ end

meaning: arbitrarily take one of the S_i's whose boolean guard b_i evaluates to true and execute it; abort if all guards evaluate to false.

(2) **guarded iteration:** do $b_1 : S_1 | \ldots | b_n : S_n$ end

meaning: while $b_1 \vee \ldots \vee b_n$ do if $b_1 : S_1 | \ldots | b_n : S_n$ end od

1.4. Synchronisation at waiting points

Synchronisation is established by so called **guarded regions:**

when-statement : when $b_1 : S_1 | \ldots | b_n : S_n$ end

with meaning : **wait** until at least one of the guards b_i is true, then select one of these arbitrarily and execute the corresponding statement S_i.

cycle-statement : cycle $b_1 : S_1 | \ldots | b_n : S_n$ end

with meaning : do true : when $b_1 : S_1 | \ldots | b_n : S_n$ end

We now define the notion of **synchronisation** taking place at a **waiting point.**
By **synchronisation** we mean either:

(1) the act of honouring an external request from another process, or
(2) the act of resuming execution of a guarded region (having been a waiting point earlier on) as result of a condition having become true. (This resembles a (nondeterministic) coroutine-mode of operation.)

The choice of one of these synchronisation-acts at each waiting point is completely nondeterministic, as is the possible choice between guarded regions that can be resumed

Control is at a **waiting point** within a process

(1) at a guarded region, if all of its guards are false on arrival,
(2) upon termination of its initial statement
(3) upon completion of an external request to one of its common procedures

1.5. Data-types

The only data-types we allow are the **boolean** and **integer** types.

1. SUMMARY OF THE PROOF SYSTEM

This appendix contains the formal definitions of the proof-theoretical terms introduced in the abstract and a list of the axioms and proof rules making up the DP-proof system.

1.1. Proof-theoretical terms

Bracketed section

A **bracketed section** within a DP-program $[P_1 \| \ldots \| P_n]$ is of the form $<S>$, $<S_1;T_1>$ or $<T_2;S_2>$, where

(1) S within process P_i is of the form $S_1;$ call $P_j.pr(x,y);S_2$ $(j \neq i)$,

(2) pr in process P_i is declared as proc $pr(u \# v)$ begin $T_1;> T <;T_2$ end,

(3) S_1, T_1, T_2 and S_2 do not contain any call-statements or when-statements.

Auxiliary variables (AV)

Auxiliary variables differ from (the normal) program variables and formal parameters. They may only appear in assignments of the form $x:=t$, with $x \in AV$. Each auxiliary variable may be assigned to in **one** process only.

General invariant

GI contains no free variables subject to change outside bracketed sections

Process invariants

The process invariant, PI_i, of a process P_i may only contain variables, or auxiliary variables free in P_i and location predicates referencing statements of P_i.

1.2. Proof rules and axioms

Axioms

assignment :

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

skip :

$$\{p\}\ \underline{skip}\ \{p\}$$

Rules

composition :

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

consequence :

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

conjunction :

$$\frac{\{p\}\ S\ \{r\},\ \{q\}\ S\ \{r\}}{\{p\ \ q\}\ S\ \{r\}}$$

substitution :

$$\frac{\{p\}\ S\ \{q\}}{\{p[z/x]\}\ S\ \{q\}},\quad \text{provided } x \notin free(S,q)$$

if :

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

do :

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

when :

$$\frac{\begin{array}{l}\{p\wedge b_1\}\ S_1\ \{q\},\ \ldots,\ \{p\wedge b_n\}\ S_n\ \{q\},\\ p\wedge\neg \bigvee_{i=1}^{n} b_i \rightarrow PI{:}at('S'),\\ \{b_1\wedge PI{:}at('S')\}\ S_1\ \{q\},\ \ldots,\ \{b_n\wedge PI{:}at('S')\}\ S_n\ \{q\}\end{array}}{\{p\}\ \underline{\text{when}}\ b_1 : S_1 | \ldots | b_n : S_n\ \underline{\text{end}}\ \{q\}}$$

external-request-rule (*) :

$$\frac{\begin{array}{l}\{p\wedge MI_j\wedge GI\}\ S_1;T_1[\bullet]\ \{PI_j{:}at(pr)[\bullet]\wedge p_2\wedge GI\},\\ \{PI_j{:}at(pr)\}\ T\ \{PI_j{:}after(pr)\},\\ \{p_2\wedge PI_j{:}after(pr)[\bullet]\wedge GI\}\ T_2[\bullet];S_2\ \{q\wedge MI_j\wedge GI\}\end{array}}{\{p\}\ \langle S_1;\ \underline{\text{call}}\ P_j.pr(\bar{t},\bar{y});\ S_2\rangle\ \{q\}}$$

where (1) P_i contains the bracketed section,

(2) P_j the declaration $\underline{\text{proc}}\ pr(\bar{u},\bar{v})\ \underline{\text{begin}}\ T_1;\rangle\ T\ \langle;T_2\ \underline{\text{end}}$ $(i\neq j)$,

(3) $[\bullet] = [\bar{t}/\bar{u},\ \bar{y}/\bar{v}]$,

(4) $free(p,q)\subseteq free(P_i)$

(5) $free(p_2)\subseteq free(P_i)\setminus\bar{y}$

(6) $free(PI_j{:}at(pr),PI_j{:}after(pr)) \subseteq free(P_j)$

parallel composition :

$$\frac{\{PI_i{:}at(Init_i)\}\ Init_i\ \{PI_i{:}after(Init_i)\}\ \text{and}\ PI_i\ \text{is interference free},\ i=1..n}{\{\bigwedge_{i=1}^{n} PI_i{:}at(Init_i)\wedge GI\}\ [P_1\|\ldots\|P_n]\ \{\bigwedge_{i=1}^{n} PI_i{:}after(Init_i)\wedge GI\}}$$

where (1) PI_i denotes the process invariant of process P_i,

(2) $Init_i$ is the initial statement of process P_i

AV :

$$\frac{\{p\}\ S'\ \{q\}}{\{p\}\ S\ \{q\}},$$

provided $free(q)\cap AV=\emptyset$ and S is obtained from S´ by deleting all assignments of the form $x:=t$ for $x\in AV$

(*) The barred parameters and variables denote parameter and variable lists

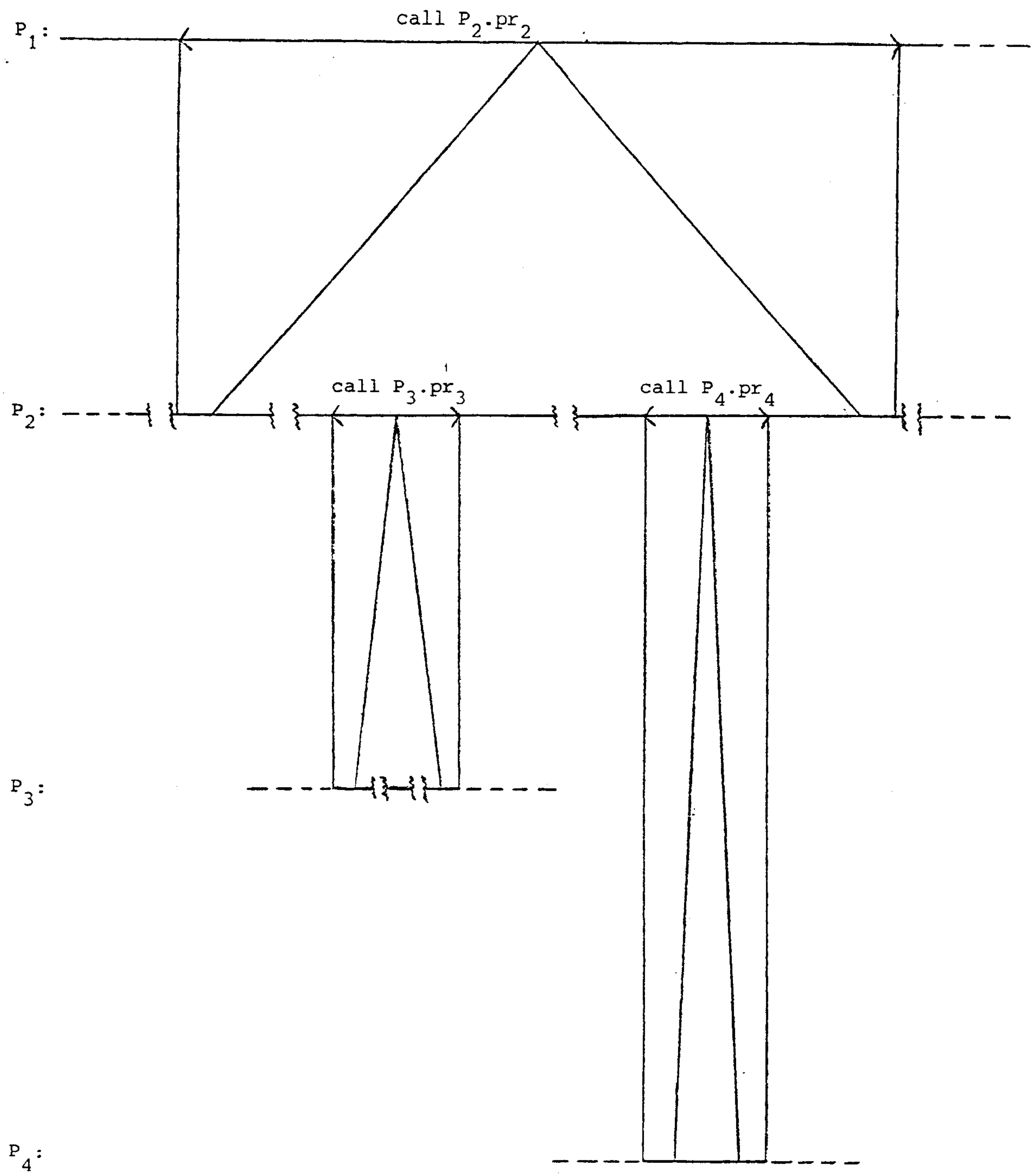


figure 1

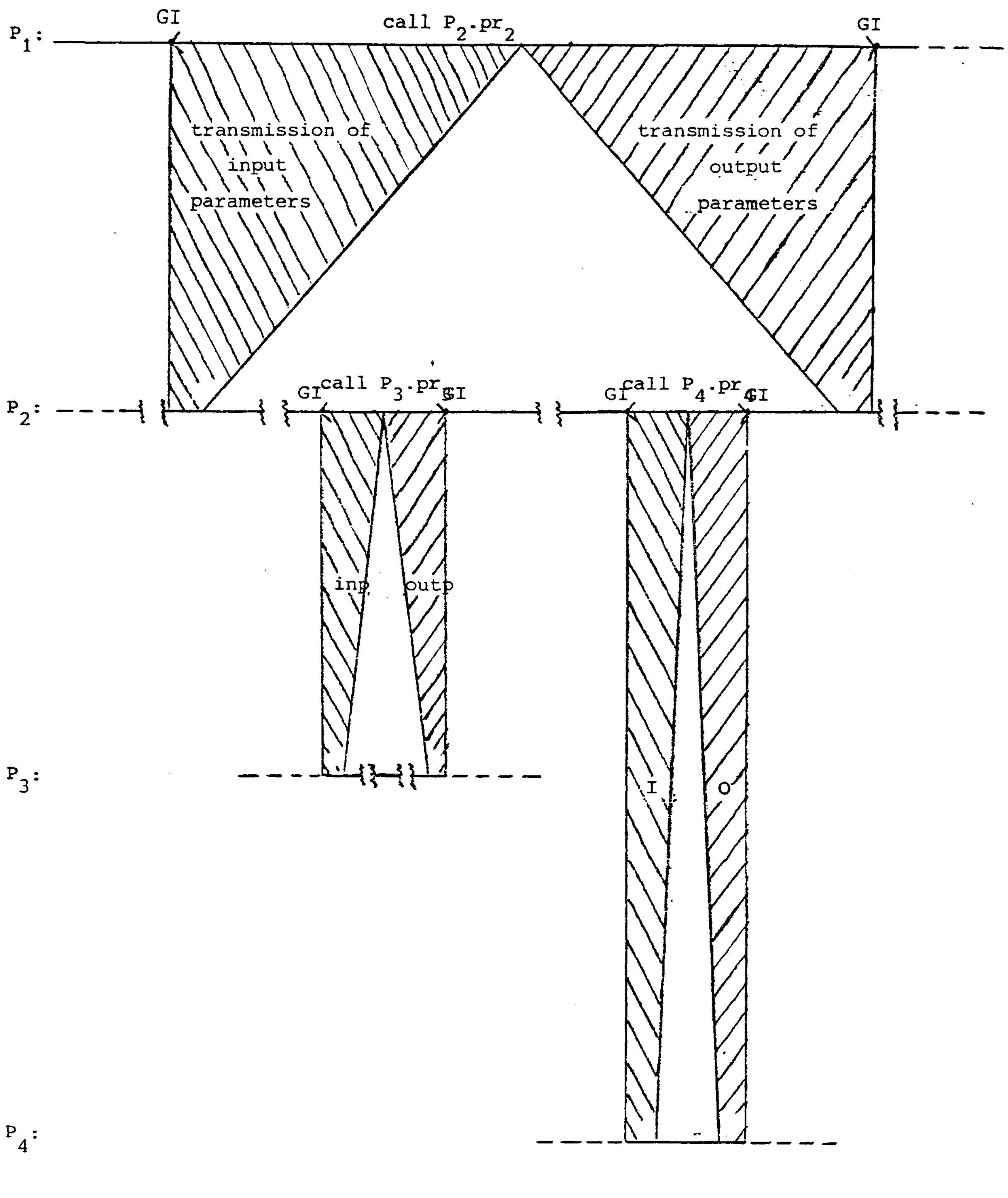


figure 1

overlay 1

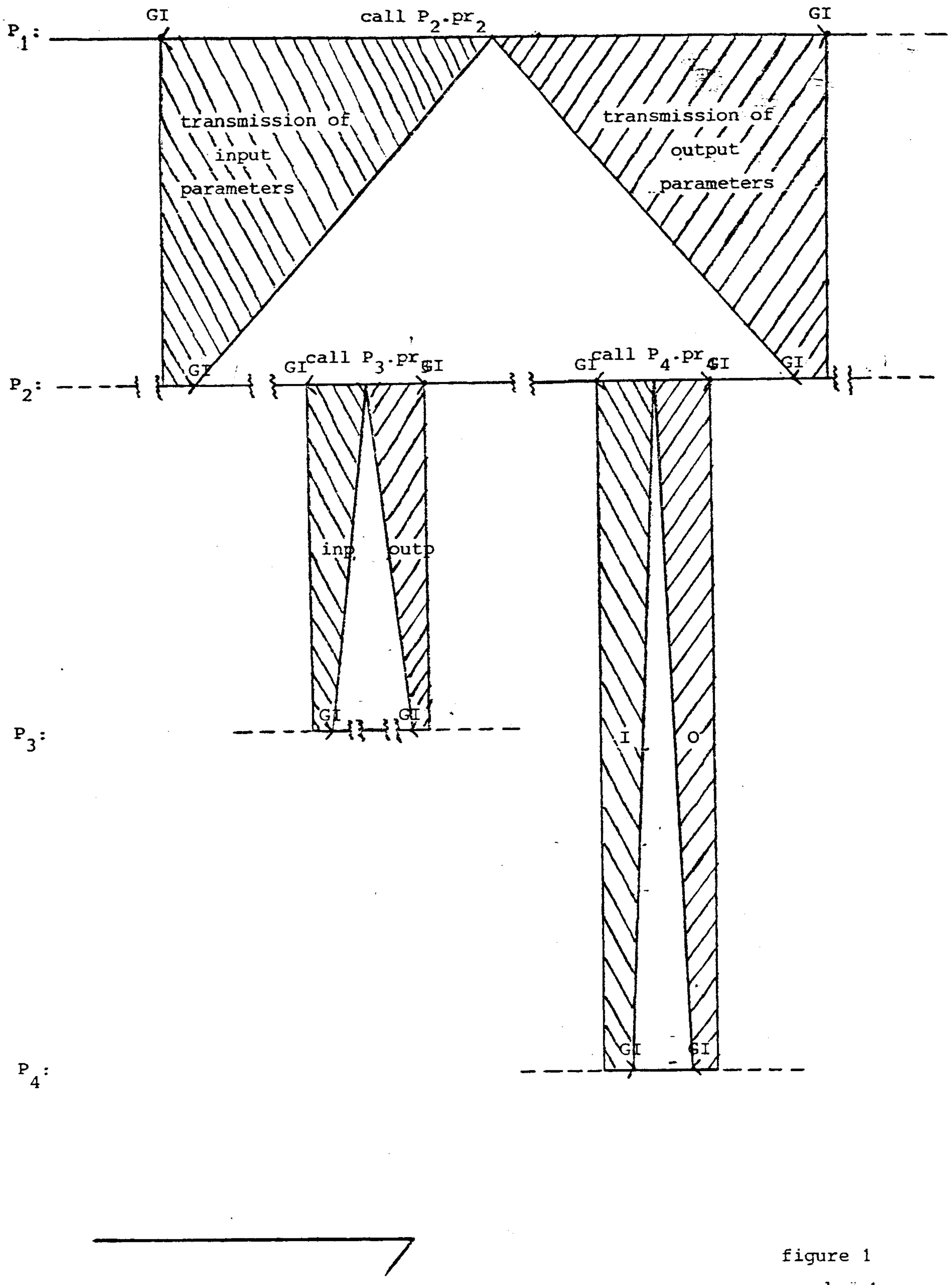


figure 1
overlay 1
overlay 2

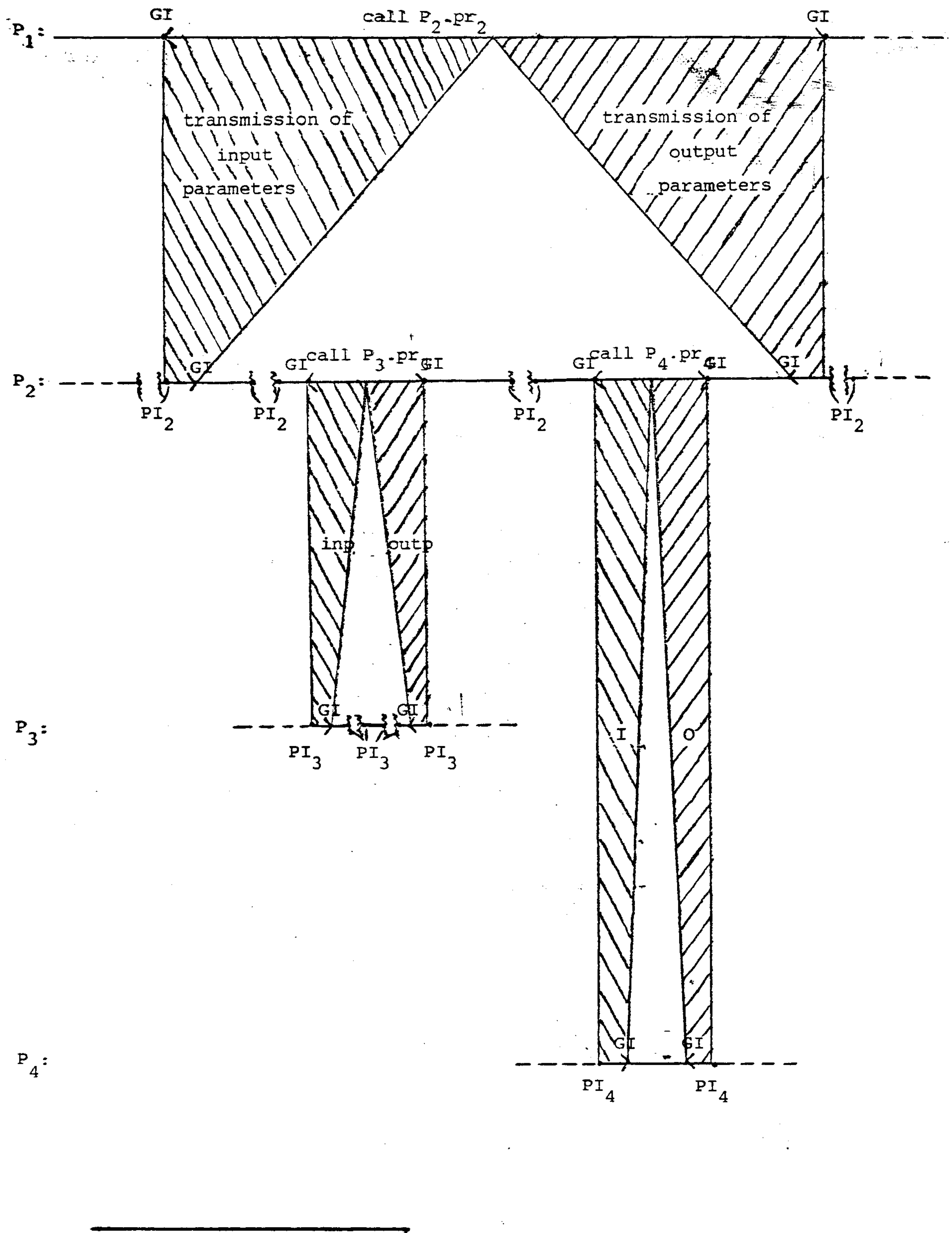


figure 1

overlay 1

overlay 2

overlay 3

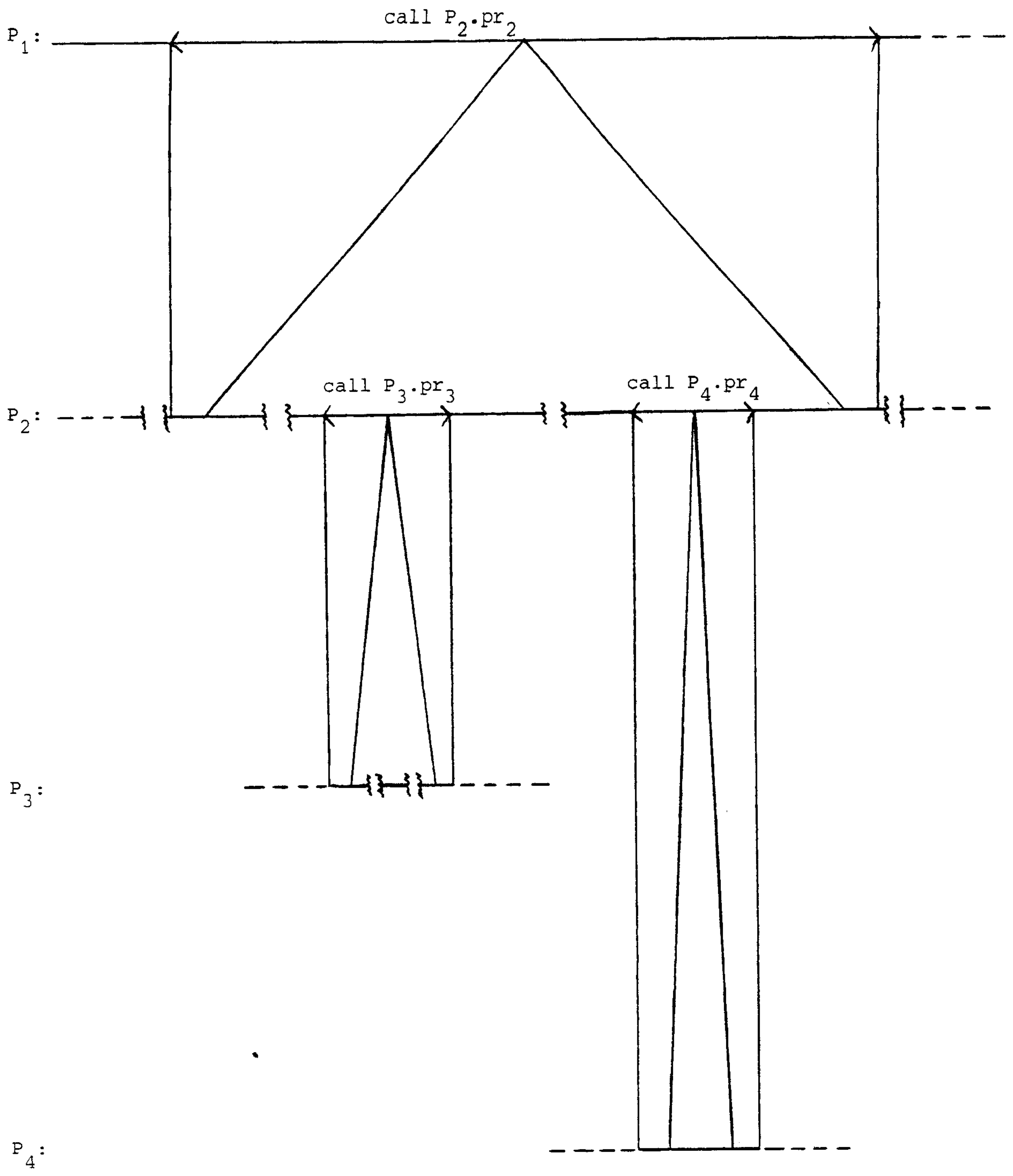
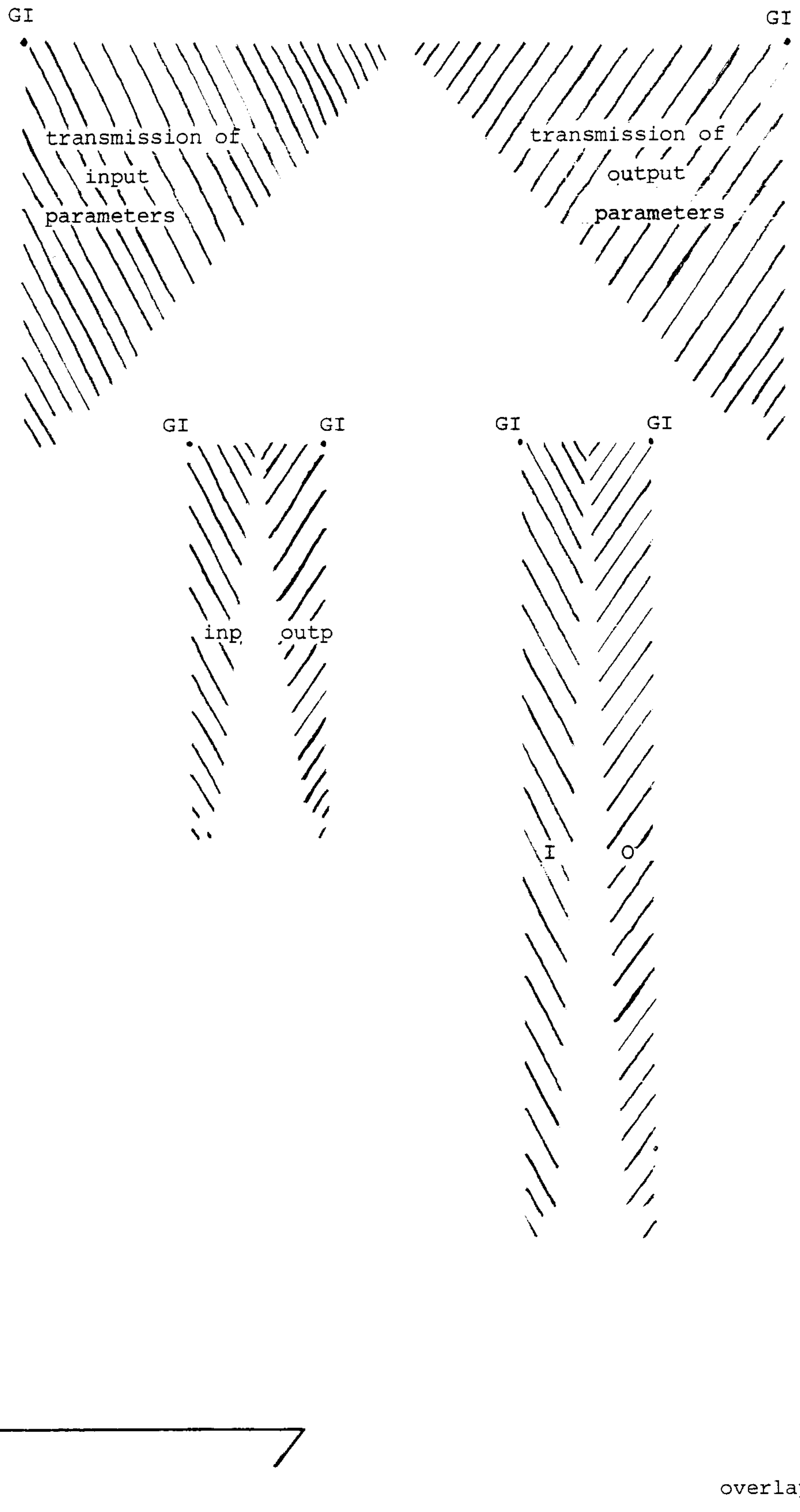


figure 1


GI
GI
transmission of
input
parameters
transmission of
output
parameters
GI
GI
GI
GI
inp
outp
I
O

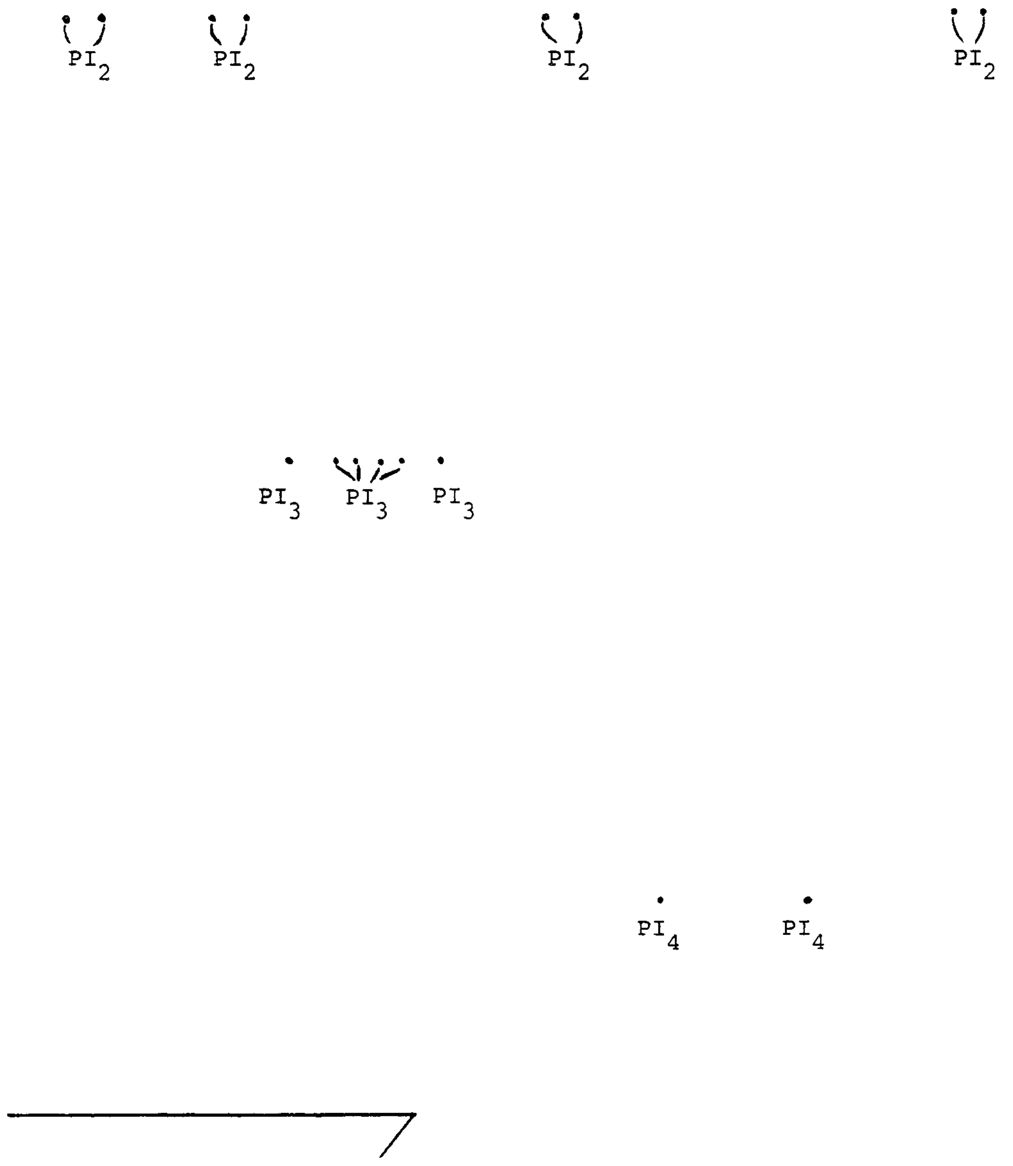
overlay 1

GI GI

GI GI

GI GI

overlay 2



overlay 3