A PROOF SYSTEM FOR COMMUNICATING SEQUENTIAL PROCESSES

K.R. Apt

N. Francez

W.P. de Roever

RUU-CS-79-8

August 1979

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6
Postbus 80.012
3508 TA Utrecht
Telefoon 030–531454
The Netherlands

A PROOF SYSTEM FOR COMMUNICATING SEQUENTIAL PROCESSES

Krzysztof R. Apt
Faculty of Economics, University of Rotterdam, P.O. Box 1738
3000 DR Rotterdam, the Netherlands

Nissim Francez
Dept. of Computer Science, The Technion, Haifa, Israel

Willem P. de Roever
Dept. of Computer Science, University of Utrecht, P.O. Box 80.012
3508 TA Utrecht, the Netherlands

Technical Report RUU-CS-79-8

August 1979

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
the Netherlands

This technical report is a joint publication of the departments of computer science of the Technion, Haifa, Israel, and the universities of Rotterdam and Utrecht, the Netherlands.

A PROOF SYSTEM FOR COMMUNICATING SEQUENTIAL PROCESSES (*)

by

Krzysztof R. Apt (1)

Nissim Francez (2)

Willem P. de Roever (3)

ABSTRACT: An axiomatic proof system is presented for proving partial correctness and absence of deadlock (and abortion) of communicating sequential processes. The key (meta) rule introduces *cooperation between proofs*, a kind of dual to Owicki and Gries' notion of interference freedom. CSP's new convention for distributed termination of loops is incorporated. Applications of the method involve correctness proofs for two algorithms, one for distributed partitioning of sets, the other for distributed computation of the greatest common divisor of n numbers.

Keywords and phrases: Hoare-style proof rules, global invariants, cooperating proofs, CSP, communicating processes, concurrency, absence of deadlock.

AMS Subject Classification (1970): 68 A 05.

(1) Faculty of Economics, University of Rotterdam, P. O. Box 1738, 3000 DR Rotterdam, the Netherlands.

(2) Dept. of Computer Science, The Technion, Haifa, Israel.

(3) Dept. of Computer Science, University of Utrecht, P. O. Box 80.012 3508 TA Utrecht, the Netherlands.

(*) This work was supported by NSF grant No. MCS 78-673 during the authors' stay at USC in Los Angeles.

1. INTRODUCTION AND PRELIMINARIES

1.1 Introduction

This paper presents a proof system for CSP, a language for Communicating Sequential Processes due to Hoare [H 2]. This system deals with proofs of partial correctness and of deadlock freedom; proofs of soundness and relative completeness will be published separately by the first author.

Just as CSP sheds new light on synchronization and message passing both by its communication primitives and by the operations upon them, so new insights are needed to obtain a proof system for this language. In particular the following properties of CSP have to be taken care of:

- CSP stresses *simultaneity* rather than mutual exclusion as synchronization mechanism by using simultaneous communication as the only means of synchronization.
- The two communication primitives of CSP, input and output commands, can function as choice mechanism by acting as guards in (possibly nondeterministic) guarded choices and repetitions.
- CSP focusses on terminating concurrent computations by introducing a distributed termination convention for input/output guarded repetitions.

Correspondingly, to deal with these properties, we introduce:

- A (meta) rule to establish *joint cooperation between isolated proofs* for CSP's sequential components.

In these separate proofs each statement is preceded and followed by a pre - and post-assertion. These assertions satisfy the axioms and proof rules introduced for the purely sequential constructs of CSP. However, when viewed in the isolation of its sequential component, the post-assertion of an input command cannot be validated since the assertions of its corresponding output command occur in another sequential component. Now proofs cooperate if, taken together, they validate the assertions of the i/o commands mentioned in the isolated proofs. A global invariant is needed to determine which pairs of input and output commands correspond, i.e., are synchronized during execution.

~ A simple mechanism for expressing termination of repetitive commands, generalizing the expression of the termination criterion "negation of all the boolean guards" to distributed termination of CSP processes.

This termination criterion is needed for proof of absence of deadlock and abortion; it generalizes the notion of blocking [OG2] to an environment in which some processes, which are intended to terminate, fail to communicate.

The distinction between cooperation versus combat acted as an almost philosophical guideline in our efforts. Cooperation via resources versus mutual exclusion of critical regions; synchronized communication by means of CSP's communication primitives between a specified pair of processes versus asynchronous interaction by means of shared variables; even purely local variables versus globally shared variables. All these are opposing notions taken from the area of concurrent languages which accentuate in proof theory the problem of finding the missing counterpart of interference freedom [OG2] between proofs: *cooperation* between proofs.

This proof system derives from various related work:

~ Owicki's and Lamport's landmark in the proof theory of concurrent processes [OG1, OG2, L]. We benefitted also from relative completeness proofs due to Owicki and to Mazurkiewicz [O , M].

~ A still enduring effort spearheaded by Hoare to establish a firm semantic basis for CSP, in which the second and third authors participated, resulting in a denotational semantics [FLHR]. In a later stage this semantics was simplified using a generalization of Dijkstra's weakest precondition operator as a descriptive tool to obtain a characterization of the semantics of terminating programs in CSP [ADF], which brought the semantics closer to a proof system.

~ The concept of assumption/commitment pairs (interface predicates) as introduced by Francez & Pnueli [FP] to characterize the assumptions which a process has to make about the behaviour of its concurrently computing environment in order to enable it "to function properly", so as to justify in its turn the claims made by that environment upon its behaviour; thus, assumption/commitment pairs are assertions which express the cooperation between a process and its environment.

While writing up this paper we learned about related work by Carl Hauser (in preparation) and Chandy & Misra [CM].

This paper is organized as follows:
Section 1.2 contains a definition of the kernel of CSP with which we deal in this paper. The fragment incorporates guards consisting of *pairs* of a boolean expression and an input/output command. Section 2 contains the proof system and is the heart of the paper. Section 3 contains two detailed case studies of correctness proofs ~ the one of a distributed partition algorithm due to E. Feijen and described in Dijkstra [D2] (our proof differs from that of Dijkstra), and the second of an algorithm for the distributed computation of the greatest common divisor of n natural numbers taken from Francez and Rodeh [FR]. Section 4 generalizes the proof system to freedom of deadlock and abortion and contains an example. In the last section we try to assess our method by comparing it in more technical terms with related proof systems for other concurrent languages. In particular, Dijkstra's account [D1] of the Gries-Owicki theory is of relevance there.

ACKNOWLEDGEMENT: We express our gratitude to Hisa Nomura for her extra efforts in typing the manuscript.

1.2 Preliminaries

Syntax and informal meaning of the fragment of CSP considered in this paper are described by way of example:

$$P :: [P_1 \| P_2 \| P_3], \text{ where:}$$

$$P_1 :: A_1;\ [P_2\ ?\ x \rightarrow S_1 \ \square\ b_1,\ P_3\ !\ y \rightarrow S_2\ ];$$
$$*[b_2,\ P_2\ !\ u \rightarrow S_3 \ \square\ b_3,\ P_3\ ?\ u \rightarrow S_4],$$

$$P_2 :: *[P_1\ ?\ s \rightarrow S_5 \ \square\ P_1\ !\ t \rightarrow S_6 \ \square\ P_3\ ?\ s \rightarrow S_7],$$

$$P_3 :: P_1\ ?\ z\ ;\ *[b_1 \rightarrow S_8 \ \square\ b_2 \rightarrow S_9\ ].$$

~ "$\|$" denotes parallel composition; different processes in the same parallel command have disjoint sets of local variables.

~ A_i's denote elementary operations such as assignment or skip.

~ S_i's are unspecified (for abbreviating the example) program sections.

~ $P_j\ ?\ x$ (in P_i) denotes an input command, expressing an input request of P_i from P_j. Such a command is to be executed only when P_j is ready to execute a corresponding output command $P_i\ !\ y$, meaning a request to output the value of y to P_i. The combined effect of executing both commands is that of assigning the value of y to x. Either i/o command waits until a corresponding one is ready.

~ "$\square$" denotes the guard separator. Guards may be boolean (b_i's) passable when true, or i/o commands passable when a corresponding i/o command in the process addressed is ready, or a combination of both, passable when its boolean part is true and the process addressed is ready.

~ A guarded selection aborts in case all its guards are false.

~ A guard is false in one of the following cases:

i) it is a boolean expressions evaluating to false;

ii) it is an i/o command for which the process addressed has terminated;

iii) it is a combination of a boolean expression and an i/o command and either the boolean expression is false, or the process addressed in the i/o command has terminated.

~ "*" denotes a repetitive construct. Repetition continues as long as there exists a passable guard, and terminates when all guards are false.

~ ";" denotes sequential composition.

Guarded commands (i.e., selection or repetition) introduce the possibility that more than one matching pair of i/o commands occurs; e.g., in the example above the first communication of P_1 can be either with P_2 or with P_3, but not with both; this is another source of nondeterminism.

Finally, for simplicity, we consider in this paper only guarded commands of which the guards either contain all an i/o command, or are all boolean. This restriction is non-essential for the purpose of this paper in that our proof system can be easily extended to cover the case just excluded.

For the full details concerning CSP, see [H2].

2. THE PROOF SYSTEM

We intend to reason about CSP programs in a manner analogous to the work of Owicki and Gries [OG2]- first we present proofs for processes in separation and then we deduce properties of complete programs by comparing the proofs for the component processes. Therefore, we have to provide axioms and proof rules for all possible constructs of a process. One of the essential properties of CSP programs is that the meaning of processes viewed in isolation is inherently incomplete when compared with their meaning in the context of a complete program. This phenomenon is also present in a less obvious way in the case of the languages considered in [OG1] and [OG2], where the constructs await b then S and with r when b do S are meaningful, essentially, only in the context of parallel composition. Therefore, the axioms and proof rules dealing with the constructs pertinent to CSP do not capture a complete meaning of these constructs viewed separately.

The main novel contribution of this work is in our opinion the proposal of tying separate proofs together into a meaningful whole; this proposal, the test for *cooperation* between proofs, will be discussed shortly.

We adopt the following axioms and proof rules (α_i stand for i/o commands):

A1. input

$$\{p\}\ P_i\ ?\ x\ \{q\}$$

This axiom may look strange since it allows to deduce any post-assertion q of the input command whatsoever. However, any q thus introduced will be later (when proofs are tested for cooperation) checked against some post-assertion regarding corresponding output statements. An arbitrary q will in general fail to pass the cooperation test.

A2. output

$$\{p\}\ P_i ! y\ \{p\}$$

This axiom conveys the information that an output statement has no side effect.

R1. i/o guarded selection

$$\frac{\{p \wedge b_i\}\ \alpha_i\ \{r_i\},\ \{r_i\}\ S_i\ \{q\}, i=1,\ldots,m}{\{p\}[\square\ (i=1,\ldots,m)\ b_i,\ \alpha_i \longrightarrow S_i]\{q\}}$$

The meaning of this rule is that the post-assertion of an i/o directed selection must be established along each possibly selected path. We discuss later the problem of paths never selected.

R 2. i/o guarded repetition

$$\frac{\{p \wedge b_i\}\ \alpha_i\ \{r_i\},\ \{r_i\}\ S_i\ \{p\}, i=1,\ldots,m}{\{p\} * [\square\ (i=1,\ldots,m)\ b_i,\ \alpha_i \rightarrow S_i\ ]\{p\}}.$$

This rule will be strengthened in the sequel by taking into account the exit conditions of the loop.

(Our intended style of presentation is deliberately incremental in order to obtain a natural flow of the argument).

The other axioms and proof rules regarding purely sequential constructs are standard and therefore omitted.

Using these axioms and proof rules we can establish proofs for formulae of the form $\{p\}\ P_i\ \{q\}$ where P_i is a process. Each such proof can be represented, as in [OG2] by a <u>proof outline</u> in which each substatement S of P_i is preceded and followed by a corresponding assertion, pre(S) and post(S), respectively. The subsequent discussion will always refer to proofs presented in such a form.

We now present a first formulation of a proof rule (or rather a meta-rule) which can be used to deduce a property of $[P_1 \| \ldots \| P_n]$ using the proofs concerning programs P_i, $i=1,\ldots,n$. This rule has the following form:

$$\frac{\text{proofs of } \{p_i\}\ P_i\ \{q_i\} \text{ cooperate, } i=1, \ldots, n}{\{p_1 \wedge \ldots \wedge p_n\}[P_1 \| \ldots \| P_n]\{q_1 \wedge \ldots \wedge q_n\}}$$

Intuitively, proofs cooperate if they help each other to validate the post-assertions of the i/o statements mentioned in those proofs. More formally this property is expressed, as follows:

The proofs of $\{p_i\}\ P_i\ \{q_i\}$ $i=1,\ldots,n$ cooperate if

i) The assertions used in the proof of $\{p_i\}\ P_i\ \{q_i\}$ have no variables subject to change in P_j for $i \neq j$;

ii) $\{pre_1 \wedge pre_2\}\ P_j ? x \| P_i ! y\ \{post_1 \wedge post_2\}$ holds whenever $\{pre_1\}\ P_j ? x\ \{post_1\}$ and $\{pre_2\}\ P_i ! y\ \{post_2\}$ are taken from the proofs of $\{p_i\}\ P_i\ \{q_i\}$ and $\{p_j\}\ P_j\ \{q_j\}$, respectively.*

We shall need the following axioms to establish cooperation:

A3. communication

$$\{true\}\ P_i ? x \| P_j ! y\ \{x=y\}$$

provided $P_i ? x$ and $P_j ! y$ are taken from P_j and P_i, respectively.

A4. preservation

$$\{p\}\ S\ \{p\}$$

provided no free variable of p is subject to change in S.

Note that A2 is subsumed by A4. We shall also need the following proof rule.

---

* Such pairs of i/o instructions will be said to be *syntactically matching*.

R3. substitution

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

provided z does not appear free in S and q.

Example 1. Using the system above we can prove

$\{true\}\ [P_1 \| P_2 \| P_3]\ \{x=u\}$,

where $P_1 :: P_2\ !\ x$,

$P_2 :: P_1\ ?\ y;\ P_3\ !\ y$

$P_3 :: P_2\ ?\ u$

Here are the proof outlines:

$\{x=z\}\ P_2 !\ x\ \{x=z\}$,

$\{true\}\ P_1\ ?\ y\ \{y=z\};\ P_3\ !\ y\ \{y=z\}$,

$\{true\}\ P_2\ ?\ u\ \{u=z\}$.

The proofs clearly cooperate - for example

$\{x=z\}\ P_2 !\ x \| P_1 ?\ y\ \{x=z \wedge y=z\}$ can be derived -

so we get $\{x=z\}\ [P_1 \| P_2 \| P_3]\ \{x=z \wedge y=z \wedge u=z\}$. Now by applying the consequence rule we get $\{x=z\}\ [P_1 \| P_2 \| P_3]\ \{x=u\}$ from which the claim follows by applying the substitution rule.

This approach fails when dealing with programs in which some output commands to not match with any input command.

Example 2. Let

$P_1 :: P_2 ! \ 0$

$P_2 :: [P_1 ? x \rightarrow \text{skip} \square P_3 ! \ y \rightarrow \text{skip} \square P_3 ? \ y \rightarrow \text{skip}]$

$P_3 :: \text{skip}$

Clearly, $\{ \text{true} \} \ [P_1 \| P_2 \| P_3 \ \{x=0\}$ holds. However this cannot be proved in the above system, for any such proof would require to establish both $\{ \text{true} \} \ P_3 ! \ y \ \{x=0\}$ and $\{ \text{true} \} \ P_3 ? \ y \ \{x=0\}$. The latter formula is an instance of the input axiom but the former one cannot be derived in the system. ∎

We remedy this difficulty by introducing the following, rather astonishing, new output axiom:

A2'. output

$\{p\} \ P_i ! \ y \ \{q\}$

At this moment the reader might wonder: "Does not the combination of axioms A1 and A2', i.e., of $\{p\} \ P_i ? x \ \{q\}$ and $\{p\} \ P_j ! \ y \ \{q\}$ together allow us to deduce $\{p\} \ P_i \ ? x \| P_j ! \ y \ \{q\}$ for arbitrary p and q "? That this is not the case follows from the cooperation test. Using A3, the axiom of communication, and A4, the axiom of preservation, only formulae of the form $\{r\} \ P_i \ ? \ x \| P_j ! \ y \ \{x=y \wedge r\}$ can be derived, where x is not free in r, and any use of the substitution or consequence rule can only weaken the conclusion. We hope that these remarks indicate to what extent the choice of p and q above is restricted by requiring cooperation.

Next we solve the following problem:

The cooperation test between proofs requires to compare all i/o pairs which syntactically match, even though sometimes communication will never take place. A simple example follows where we run into difficulties because of this very reason:

Example 3. Let

$P_1 :: [P_2 ? x \rightarrow \text{skip} \; \square \; P_2 ! 0 \rightarrow P_2 ? x; x:=x+1]$

$P_2 :: [P_1 ! 2 \rightarrow \text{skip} \; \square \; P_1 ? z \rightarrow P_1 ! 1]$

Clearly $\{true\} \; [P_1 \| P_2] \; \{x=2\}$ holds. To prove this we are forced to use x=2 as the post-assertion of the first occurrence of $P_2 ? x$ in P_1. This assertion, however, will not pass the test for cooperation since it cannot be validated when $P_2 ? x$ is compared with $P_1 ! 1$ (; the point being that this pair also syntactically matches, although it will not be synchronized during execution). $\square$

In general, syntactic matching of a pair of i/o instructions does not imply yet that this communication will ever be taken, i.e., imply their *semantic* match. In order to take care that semantically not matching pairs of i/o instructions do not fail the cooperation test as above, we introduce a global invariant I which will determine semantic matches, and which may carry other global information needed for the proof. However, in order to express semantic matching in general one needs variables which are not necessarily the ones referred to in the i/o instructions themselves (and, as is well known, needn't be program variables either; in general auxiliary variables are needed).

For example, consider the following program sections:

$\ldots P_2 ? x; i:=i+1 \ldots \| \ldots P_1 ! y; j:=j+1 \ldots$

where i and j count the number of communications actually occurring in each process, and let therefore the criterion for semantic matching be i=j. However, i=j is no global *invariant* since the two assignments will not necessarily be executed simultaneously.

To resolve these difficulties we must reduce the number of places where the global invariant should hold. This will be done by introducing brackets, the purpose of which is to delimit program sections within which the invariant need not necessarily hold.

This phenomenon is similar to the one concerning resource invariants of Hoare (see [H1]) where the global invariant does not need to hold within the critical sections. An analogous problem arises when dealing with monitor invariants (see [HW]).

Regarding the program sections just considered the bracketing will be

$$\ldots < P_2 ?x; i:=i+1> \ldots \| \ldots < P_1 ! y; j:=j+1> \ldots$$

so that i=j will hold outside the brackets.

Definition. A process P_i is bracketed if the brackets "<" and ">" are interspersed in its text, so that for each program section <S> (to be called a *bracketed section*), S is of one of the following forms:

i) $S_1; \alpha; S_2$

or

ii) $\alpha \rightarrow S_1$,

and S_1 and S_2 do not contain any i/o statements.

□

Definition. If S is bracketed then *bracket* (S) denotes the set of all variables appearing in some bracketed section <S> of S.

□

With each proof of $\{p\}$ $[P_1 \| \dots \| P_n]$ $\{q\}$ we now associate a global invariant I, and appropriate brackets. Therefore the proof rule concerning parallel composition becomes as follows (in second approximation):

R.3 parallel composition

$$\frac{\text{proofs of } \{p_i\}\ P_i\ \{q_i\} \text{ cooperate, } i=1, \dots, n}{\{p_1 \wedge \dots \wedge p_n \wedge I\}\ [P_1 \| \dots \| P_n]\ \{q_1 \wedge \dots \wedge q_n \wedge I\}}$$

provided

$$\text{free } (I) \subseteq \text{bracket } ([P_1 \| \dots \| P_n]) .$$

We have now to define precisely when proofs cooperate. Assume a given bracketing of $[P_1 \| \dots \| P_n]$ (to which we referred in the clause concerning *free* (I).

Definition. Let $<S_1>$ and $<S_2>$ denote two bracketed sections from P_i and P_j $(i \neq j)$. We say that S_1 and S_2 *match* if S_1 and S_2 contain matching i/o commands.

Definition. The proofs of the $\{p_i\}$ P_i $\{q_i\}$, $i=1,\dots,n$, *cooperate* if

(i) the assertions used in the proof of $\{p_i\}$ P_i $\{q_i\}$ have no free variables subject to change in P_j $(i \neq j)$.

(ii) $\{pre(S_1) \wedge pre(S_2) \wedge I\}$ $S_1 \| S_2$ $\{post(S_1) \wedge post(S_2) \wedge I\}$ holds for all matching pairs of bracketed sections $<S_1>$ and $<S_2>$.

The following additional proof rules are used to establish cooperation:

R4. formation

$$\frac{\{p\}\ S_1;S_3\ \{p_1\},\ \{p_1\}\ \alpha \parallel \bar{\alpha}\ \{p_2\},\ \{p_2\}\ S_2;S_4\ \{q\}}{\{p\}(S_1;\alpha;S_2) \parallel (S_3;\bar{\alpha};S_4)\ \{q\}}$$

provided α and $\bar{\alpha}$ match, and S_1, S_2, S_3, S_4 do not contain any i/o commands.

R5. arrow

$$\frac{\{p\}\ (\alpha;S) \parallel S_1\ \{q\}}{\{p\}\ (\alpha \rightarrow S) \parallel S_1\ \{q\}}$$

R4 and R5 reduce the proof of cooperation to sequential reasoning, except for an appeal to the communication axiom. In this sequential reasoning, assertions appearing within brackets can be used.

Finally, we use auxiliary variables whenever needed. These are variables which do not affect program control during execution, and are added only for expressing assertions and invariants, which cannot be expressed in terms of the program variables alone. We use rule R6, due to Gries & Owicki [OG2] for deleting assignments to auxiliary variables.

R6. auxiliary variables

Let AV be a set of variables such that $x \in AV \Rightarrow x$ appears in S' only in assignments $y:=t$, where $y \in AV$. Then if p and q are assertions which do not contain free any variables from AV, and if S is obtained from S' by deleting all assignments to variables in AV,

$$\frac{\{p\}\ S'\ \{q\}}{\{p\}\ S\ \{q\}} \quad .$$

Example 4. We now show how to verify the program from example 3. Two auxiliary variables i and j are needed. We give proof outlines for the already bracketed program S.

$\{i=0 \wedge j=0\}$

$[\{i=0\}$

$[\langle P_2 ? x \{x=2\} \rightarrow i:=1\rangle \{x=2 \wedge i=1\};$ skip $\{x=2\}$

$\square$

$\langle P_2 ! 0 \{true\} \rightarrow i:=1\rangle \{i=1\}; \langle P_2 ? x \{x=1\}; i:=2\rangle \{x=1 \wedge i=2\};$

$x:=x+1 \;\{x=2\}$

$]\{x=2\}$

$\|$

$[\{j=0\}$

$\langle P_1 ! 2 \{true\} \rightarrow j:=1\rangle \{j=1\};$ skip $\{true\}$

$\square$

$\langle P_1 ? z \{z=0\} \rightarrow j:=1\rangle \{z=0 \wedge j=1\}; \langle P_1 ! 1 \{true\}; j:=2\rangle \{j=2\}$

$]\{true\}$

$]$

$\{x=2\}$

We choose $I \equiv (i=j)$. Cooperation is easily established. Note that that $i=0 \wedge (z=0 \wedge j=1) \wedge I \equiv$ false, so the bracketed sections containing $P_2 ? x$ and $P_1 ! 1$ pass the cooperation test trivially. Hence by the parallel composition rule, and rule R6,

$$\{i=0 \wedge j=0\} \; [P_1 \| P_2 \quad] \; \{x=2\}$$

holds. Applying the substitution rule we finally get

$$\{true\} \; [P_1 \| P_2 \quad] \; \{x=2\} \; .$$

The last problem that remains to be solved is that of i/o guarded repetitions. Rule R2 does not provide any means to deduce that upon exit of the loop $*[\Box\ (i=1, \ldots, m)\ b_i, \alpha_i \rightarrow S_i]$ some of b_i 's may be false. In particular, this rule is apparently insufficient to prove $\{true\}\ [P_1 \| P_2]\ \{b\}$ with

$$P_1 :: *[b, P_2 ? x \rightarrow b:=false]$$

and

$$P_2 :: skip,$$

and also insufficient to prove $\{true\}\ [P_1 \| P_2]\ \{\neg b\}$,

with P_1 as above, and $P_2 :: P_1 ! y$.

Since the ultimate rule for the i/o guarded repetition is also necessary for proving deadlock freedom, we postpone its formulation and discussion until **section 4.**

3. CASE STUDIES

3.1 Partitioning a set

Given two disjoint sets of integers S and T; $S \cup T$ has to be partitioned into two subsets S^∇ and T^∇ s.t. $|S|=|S^\nabla|$, $|T|=|T^\nabla|$, and every element of S^∇ is smaller than any element of T^∇. The program P and its correctness proof are inspired upon Dijkstra [D2]; however the proof presented here differs from Dijkstra's one. $P::[P_1 \| P_2]$, as given below, and $S \neq \emptyset$.

```
P1:: mx:=max(S);                       | P2 :: P1 ? y; T:=T∪{y} ;
     P2 ! mx; S:=S-{mx};               |      mn:=min(T);
     P2 ? x ; S:=S∪{x} ;               |      P1 ! mn; T:=T-{mn} ;
     mx:=max(S) ;                      |
    *[mx >x → P2 ! mx; S:=S-{mx} ;     |     *[P1 ? y→ T:=T∪{y} ;
              P2 ? x ; S:=S∪{x} ;      |               mn:=min(T);
              mx:= max(S)              |               P1 ! mn; T:=T-{mn}
     ]                                 |      ]
```

Intuitively, these programs execute the following loop: Let S and T denote set variables; then processes P_1 and P_2 exchange the current maximum of S, max(S), with the current minimum of T, min (T), until max (S) in P_1 equals the value last received from P_2.

The correctness proof of P requires the introduction of two auxiliary variables ℓ_1 in P_1 and ℓ_2 in P_2, to enable expression of the global invariant GI; ℓ_i counts the number of communications performed by P_i.

The purposes of GI are:

1) to determine which syntactically matching bracketed sections are executed indeed (by requiring $\ell_1=\ell_2$);

2) to guarantee the partitioning property;

3) to tie the local reasoning required for processes P_1 and P_2 in isolation together so as to permit derivation of $\max(S) < \min(T)$ upon (joint) loop exit; to express the global conditions on S and T needed for the local reasoning about P_1 and P_2 (in testing for cooperation).

In the annotated versions of P_1 and P_2, P_1' and P_2', the following is added to their "bare" text:

1) Assigments to the auxiliary variables ℓ_1, ℓ_2.

2) The pre- and post-conditions required for a proof, modulo deletion of conditions which were mentioned earlier in the annotated text and remained invariant, or were not relevant at earlier points.

3) Bracketed sections of instructions which, from the point of view of the proof, are considered as units for the proof of cooperation. Note that the global invariant GI requires $S \cap T = \emptyset$, and that $S:=S-\{mn\}$ and $T:=T \cup \{y\}$ are not synchronized. Thus within these units GI may be violated indeed, but <u>not</u> outside these units.

Annotated text of P_1:

$\{|S|=n_1 > 0 \wedge S=S_0 \wedge \max(S) \in S \wedge \ell_1=0\}$ $mx := \max(S);$

$\{mx \in S \wedge |S| = n_1 \wedge \ell_1 = 0\}$

$< P_2 ! \, mx; \; \ell_1 := \ell_1 + 1; \; \{mx \in S\} \; S := S - \{mx\} >;$

$\{|S| = n_1 - 1 \wedge \ell_1 = 1\}$

$< P_2 ? \, x; \; \ell_1 := \ell_1 + 1; \; \{x \notin S\} \; S := S \cup \{x\} >;$

$\{|S| = n_1 \wedge x \in S \wedge \ell_1 = 2\}$

$mx := \max(S);$

LI_1: $\{|S| = n_1 \wedge mx = \max(S) \wedge x \leq \max(S) \wedge \text{even}(\ell_1) \wedge \ell_1 \geq 2\}$

$* [\, mx > x \rightarrow \{mx \in S \wedge LI_1\} < P_2 ! \, mx; \; \ell_1 := \ell_1 + 1; \; \{mx \in S\}$

$S := S - \{mx\} >;$

$\{|S| = n_1 - 1 \wedge \text{odd}(\ell_1) \wedge \ell_1 \geq 2\}$

$< P_2 ? \, x; \; \ell_1 := \ell_1 + 1 \; \{x \notin S\}; \; S := S \cup \{x\} >;$

$\{|S| = n_1 \wedge x \in S \wedge \text{even}(\ell_1)\}$

$mx := \max(S)$

LI_1: $\{|S| = n_1 \wedge x \in S \wedge mx = \max(S) \wedge \text{even}(\ell_1) \wedge \ell_1 \geq 2\}$

]

$\{\max(S) = x \wedge |S| = n_1 \wedge \text{even}(\ell_1)\}$

Annotated text of P_2:

$\{ |T| = n_2 \geq 0 \wedge T=T_0 \wedge \ell_2=0 \}$

$<P_1 ? y; \ell_2 := \ell_2+1; \{y \notin T\}\ T:=T \cup \{y\}>;$

$\{ |T| = n_2+1 \wedge \ell_2 = 1\}\ mn:=min(T);$

$\{ |T| = n_2+1 \wedge mn=min(T) \wedge \ell_2=1\}$

$<P_1 ! mn; \ell_2:=\ell_2+1; \{mn \in T\}\ T:=T-\{mn\}>;$

LI_2: $\{ |T| = n_2 \wedge mn < min(T) \wedge even(\ell_2) \wedge \ell_2 \geq 2\}$

$*[< P_1 ? y \rightarrow \ell_2:=\ell_2+1;\ T:=T \cup \{y\}>;$

$\{ |T| = n_2+1 \wedge odd(\ell_2)\} mn:=min(T);$

$\{ |T| = n_2+1 \wedge mn=min(T) \wedge odd(\ell_2) \wedge \ell_2 \geq 2\}$

$< P_1 ! mn; \ell_2:=\ell_2+1; T:=T-\{mn\}>$

LI_2: $\{ |T| = n_2 \wedge mn < min(T) \wedge even(\ell_2) \wedge \ell_2 \geq 2\}$

]

$\{ |T| = n_2 \wedge mn < min(T)\}$

The global invariant GI:

$$GI \equiv S \cap T=\emptyset \wedge S \cup T=S_0 \cup T_0 \wedge \ell_1=\ell_2 \wedge (even(\ell_1) \wedge \ell_1 \geq 2 \rightarrow x < min(T))$$

For the sake of the proof we assume that

$$min(\emptyset) = +\infty.$$

We restrict ourselves to proving cooperation between proofs for the first bracketed section of P_1 and of P_2, and for the second bracketed section of P_1 and P_2; the customary kind of sequential reasoning is omitted. Proofs for cooperation for the third bracketed sections and the fourth ones are actually identical, and omitted. Proofs for syntactically matching but semantically non-matching sections are trivial; e.g., the first section of P_1 and the third one of P_3 are trivially cooperating since $\neg$GI holds (in this case $\neg(\ell_1=0 \wedge \ell_2 \geq 2 \wedge \ell_1=\ell_2)$). Note also how the input and output axioms are used to insert the occurrences of $\{mx \in S\}$, $\{x \notin S\}$, $\{y \notin T\}$ and $\{mn \in T\}$ in the annotated program; the choice of these assertions will be justified in the cooperation proofs.

Proof of cooperation between first bracketed sections:

One has $pre_1 \equiv mx \in S \wedge |S|=n_1 \wedge \ell_1=0$, and

$$pre_2 \equiv |T|=n_2 \wedge T=T_0 \wedge \ell_2=0.$$

Also, $post_1 \equiv |S|=n_1-1 \wedge \ell_1=1$, $post_2 \equiv |T|=n_2+1 \wedge \ell_2=1$.

We have to prove: $\{pre_1 \wedge pre_2 \wedge GI\}$

$$P_2!\,mx;\ \ell_1:=\ell_1+1;\ S:=S-\{mx\} \quad \| \quad P_1?\,y;\ \ell_2:=\ell_2+1;\ T:=T\cup\{y\}$$

$$\{post_1 \wedge post_2 \wedge GI\}.$$

By the communication axiom and preservation axioms,

$$\{pre_1 \wedge pre_2 \wedge GI\}\ \ P_2!\,mx \,\|\, P_1?\,y\ \ \{mx=y \wedge pre_1 \wedge pre_2 \wedge GI\}.$$

Pre-condition of section $\ell_1:=\ell_1+1;\ S:=S-\{mx\};\ \ell_2:=\ell_2+1;\ T:=T\cup\{y\}$ w.r.t. post-condition $post_1 \wedge post_2 \wedge GI$ is

$$\ell_1=\ell_2=0 \wedge y \notin T \wedge |T|=n_2 \wedge mx \in S \wedge |S|=n_1 \wedge S\cap T=\emptyset \wedge S\cup T=S_0\cup T_0,$$

which is implied by $\{mx=y \wedge pre_1 \wedge pre_2 \wedge GI\}$.

Therefore the formation rule yields the result since

$$\{pre_1 \wedge pre_2 \wedge GI\} \;\; P_2 ! \, mx \,\|\, P_1 ? \, y \;\; \{mx=y \wedge pre_1 \wedge pre_2 \wedge GI\}$$

and

$$\{mx=y \wedge pre_1 \wedge pre_2 \wedge GI\} \;\; \ell_1:=\ell_1+1;\ S:=S-\{mx\};\ \ell_2:=\ell_2+1;\ T:=T\cup\{y\}$$
$$\{post_1 \wedge post_2 \wedge GI\} \text{ holds.}$$

Proof of cooperation between second bracketed sections:

One has $pre_1' \equiv |S|=n_1-1 \wedge \ell_1=1$, and $pre_2' \equiv |T|=n_2+1 \wedge mn=\min(T) \wedge \ell_2=1$; and $post_1' \equiv |S|=n_1 \wedge x\in S \wedge \ell_1=2$, $post_2' \equiv |T|=n_2 \wedge mn<\min(T) \wedge \text{even}(\ell_2)$
$\wedge\ \ell_2 \geq 2$.

We have to prove: $\{pre_1' \wedge pre_2' \wedge GI\}$

$$P_2 ? \, x;\ \ell_1:=\ell_1+1;\ S:=S\cup\{x\} \quad \| \quad P_1 ! \, mn;\ \ell_2:=\ell_2+1;\ T:=T-\{mn\}$$
$$\{post_1' \wedge post_2' \wedge GI\} \ .$$

By the communication axiom and preservation axiom

$$\{pre_1' \wedge pre_2' \wedge GI\} \;\; P_2 ? \, x \,\|\, P_1 ! \, mn \;\; \{mn=x \wedge pre_1' \wedge pre_2' \wedge GI\},$$

since odd (ℓ_1).

Now observe that

$$\{mn=x \wedge pre_1' \wedge pre_2' \wedge GI\}$$
$$\ell_1:=\ell_1+1;\ S:=S\cup\{x\};\ \ell_2:=\ell_2+1;\ T:=T-\{mn\}$$
$$\{post_1' \wedge post_2' \wedge GI\}$$

holds.

Note that $x<\min(T)$ in the post-assertion follows from the fact that

$$mn=x \wedge mn=\min(T) \rightarrow x<\min(T-\{mn\}).$$

Therefore the formation rule yields the result.

Applying the rule of parallel programs we get

$$\{ |S| = n_1 > 0 \wedge S = S_0 \wedge |T| = n_2 \geq 0 \wedge T = T_0 \wedge \ell_1 = 0 \wedge \ell_2 = 0 \wedge GI\}$$

$$[P'_1 \parallel P'_2]$$

$$\{ LI_1 \wedge LI_2 \wedge GI\}$$

where $P'_1 \wedge$ and P'_2 are the modified versions of P_1 and P_2.

From this we obtain

$$\{ |S| = n_1 > 0 \wedge S = S_0 \wedge |T| = n_2 \geq 0 \wedge T = T_0 \wedge S \cap T = \emptyset\}$$

$$\ell_1 := 0;\ \ell_2 := 0;\ [P'_1 \parallel P'_2]$$

$$\{ |S| = n_1 \wedge |T| = n_2 \wedge S \cap T = \emptyset \wedge S \cup T = S_0 \cup T_0 \wedge \max(S) < \min(T)\}$$

Now by dropping the assignments to ℓ_1 and ℓ_2 we get the desired formula.

3.2 Distributed computation of the greatest common divisor of n numbers

As another example, we shall consider a program P which computes gcd $(\sigma_1,\ldots,\sigma_n)$, $\sigma_i>0$, $i=1,\ldots,n$, a variant of a program first presented in [FR]. This program has the property that when all processes reached a final state and have computed the gcd, the program is blocked in a deadlock state, since no process "knows" that all other processes are in final states. The interest in such programs arises because of two facts:

1. It may be easier to write such a program then the corresponding program that will terminate when all processes reached final states.
2. There exists an automatic transformation transforming every such blocked program into an equivalent terminating program. See [F, FR] for details about this transformation.

Using such an example, we are also able to show that our deductive system can deal with more general invariance (or safety in the terminology of [L]) then just partial correctness.

The program P consists of n parallel processes arranged in a ring configuration, where each processes P_i communicates with its own immediate neighbours P_{i-1}, P_{i+1} ('+' and '-' are interpreted cyclically in $\{1,\ldots,n\}$). Each process has a local variable x_i which initially has the value σ_i. Each process sends its own x_i to each immediate neighbour, and uses flags rsl (ready to send left) and rsr (ready to send right) to avoid sending x_i again before it was modified. Other alternatives of P_i are to receive a copy of x_{i-1} in y, or a copy of x_{i+1} in z. Upon receiving such number from a neighbourprocess, the number is compared to x_i. If x_i is smaller, then it is updated according to Euclid's rule, and the rsl, rsr flags are set on. Otherwise nothing happens. Two auxiliary variables rcvl (received from left) rcvr (received from right) are included for the sake of the proof.

Since the program deadlocks upon reaching the final state, no post-condition is claimed for the whole program. Rather, we shall show how to express in the formalism the claim about the state at the instant of blocking.

In the annotation LI_i is the loop variant of P_i which serves also as the pre-condition and post-condition for the body of the main loop.

Following is the annotated text for P_i

$\{ x_i = \sigma_i > 0 \wedge rsl_i \wedge rsr_i \}$

$*[\{ LI_i \}$

$\langle rsl_i, P_{i-1} ! x_i \rightarrow rsl_i := false; rcvl_i := false \{ LI_i \}$

$\square$

$\langle rsr_i, P_{i+1} ! x_i \rightarrow rsr_i := false; rcvr_i := false \rangle \{ LI_i \}$

$\square$

$\langle P_{i-1} ? y_i \rightarrow rcvl_i := true;$

$[y_i \geq x_i \rightarrow skip$

$\square$

$y_i < x_i \rightarrow [y_i | x_i \rightarrow x_i := y_i$

$\square$

$y_i \nmid x_i \rightarrow x_i := x_i \bmod y_i$

$] ; \{ LI_i \} \; rsr_i := true; rsl_i := true$

$] \rangle \{ LI_i \}$

□

$\langle P_{i+1} ? z_i \rightarrow$ $rcvr_i$:=true;

$[z_i \geq x_i \rightarrow$ skip

□

$z_i < x_i \rightarrow [z_i | x_i \rightarrow x_i := z_i$

□

$z_i \nmid x_i \rightarrow x_i := x_i \bmod z_i$

]; $\{LI_i\}$ rsr_i:=true; rsl:=true

]$\rangle$ $\{LI_i\}$

]

The global invariant, GI, is the following:

$$GI \equiv \bigwedge_{i=1}^{n} [\neg rsl_i \rightarrow (z_{i-1} = x_i \wedge rcvr_{i-1})$$

$$\wedge$$

$$\neg rsr_i \rightarrow (y_{i+1} = x_i \wedge rcvl_{i+1})]$$

$$\wedge$$

$$gcd(x_1, \ldots, x_n) = gcd(\sigma_1, \ldots, \sigma_n)$$

GI establishes the correct sending and receiving relationship between any triple P_{i-1}, P_i, P_{i+1}, and also that all changes in the x_i's preserve gcd $(\sigma_1 \ldots, \sigma_n)$.

The loop invariant LI_i is expressed in terms of local variables (of P_i) only, and describes the sequential behaviour of the loop body

$$LI_i \equiv (\neg rsl_i \wedge rcvl_i \rightarrow y_i \geq x_i)$$

$$\wedge$$

$$(\neg rsr_i \wedge rcvr_i \rightarrow z_i \geq x_i)$$

The instant where a process is about to execute the loop body and find itself blocked is characterized by

$BL_i \equiv (LI_i \wedge \neg rsl_i \wedge \neg rsr_i)$.

Therefore, we have to prove the following property:

(*) $(GI \wedge \bigwedge_{i=1}^{n} BL_i) \rightarrow (\bigwedge_{i=1}^{n} x_i = gcd(\sigma_1, \ldots, \sigma_n))$

(*) implies that the conclusion indeed holds at the instant of total blocking if it occurs.

Proof of (*): Suppose that $GI \wedge \bigwedge_{i=1}^{n} BL_i$ holds.

From $GI \wedge \bigwedge_{i=1}^{n} (\neg rsl_i \wedge \neg rsr_i)$ we infer

(1) $\bigwedge_{i=1}^{n} (x_i = z_{i-1} = y_{i+1}) \wedge rcvr_i \wedge rcvl_i$.

From $\bigwedge_{i=1}^{n} (LI_i \wedge \neg rsl_i \wedge \neg rsr_i \wedge rcvl_i \wedge rcvr_i)$ we infer

(2) $\bigwedge_{i=1}^{n} (y_i \geq x_i \wedge z_i \geq x_i)$.

Using (1) and (2) we get

$x_i \leq z_i = x_{i+1}$

and

$x_{i+1} \leq y_{i+1} = x_i$ which, together, imply

(3) $x_i = x_{i+1}$, and therefore

(4) $x_1 = x_2 = \ldots = x_n$.

Finally, (4) and $gcd(x_1, \ldots, x_n) = gcd(\sigma_1 \ldots, \sigma_n)$ imply the required conclusion $\bigwedge_{i=1}^{n} x_i = gcd(\sigma_1, \ldots, \sigma_n)$.

We are left with the problem of verifying that GI is indeed a global invariant, and LI_i is a local loop invariant. The second task involves ordinary sequential reasoning using the input and output axioms, and is left to the reader.

On the other hand, a proof of the global invariance of GI uses the concept of cooperation.

(a) Initially, $\bigwedge_{i=1}^{n} \neg rsl_i \wedge \neg rsr_i$ is false, and the two first clauses of GI are trivially true. Also $\bigwedge_{i=1}^{n} x_i=\sigma_i$ trivially implies the third clause.

(b) One pair of matching bracketed sections is the one consisting of the first alternative of some P_i and the fourth alternative of P_{i-1}. Hence, we have to show

$$\{rsl_i \wedge LI_i \wedge LI_{i-1} \wedge GI\}$$

$$P_{i-1}\,!\,x_i;\ \underbrace{rsl_i{:=}false;\ rcvl_i{:=}false}_{A}$$

$$\|$$

$$P_i\,?\,z_{i-1};\ \underbrace{rcvr_{i-1}{:=}true;\ [\ldots\ ]}_{B}$$

$$\{LI_i \wedge LI_{i-1} \wedge GI\}$$

The variables changed are: rsl_i, rsl_{i-1}, rsr_{i-1}, $rcvl_i$, $rcvl_{i-1}$, z_{i-1}, x_{i-1}

By the rule of formation it remains to be proved that

$$\{x_i=z_{i-1} \wedge rs\ell_i \wedge LI_i \wedge (\neg rs\ell_{i=1} \wedge rcv\ell_{i-1} \rightarrow y_{i-1} \geq x_{i-1}) \wedge GI\}$$

$$A;B$$

$$\{LI_i \wedge LI_{i-1} \wedge GI\}$$

holds, where the above pre-condition is the post-condition of

$$P_{i-1}\,!\,x_i \| P_i\,?\,z_{i-1}$$

inferred by the axioms of communication and preservation.

First, $x_i=z_{i-1}$ implies, by the known mathematical facts about the gcd function, that $gcd(x_1,\ldots,x_n)=gcd(\sigma_1,\ldots,\sigma_n)$ remains true afte executing A ; B. All other changes need just routine checks.

(c) The other matching bracketed section is the second alternative of P_i and the third alternative of P_{i+1} and is verified similarly.

4. DEADLOCK FREEDOM

Similarly to Owicki and Gries , our proof system can be used to show that a given program is deadlock free. Furthermore, the method can be used to prove the absence of abortion due to attempts at communication with processes that already terminated. (This question does not arise in [OG1, OG2],], because such a situation cannot be described in their programming languages).

We adopt the concept of *blocking* introduced in [OG2] , used there to characterize those states in which execution cannot be continued. Our adoption takes into account the distributed termination convention of CSP in that communication at the guards of *an i/o guarded repetition will not be blocked* in case all the processes referred to in these guards have terminated. All other communications which address processes that have terminated will be blocked. Intuitively, a program is blocked (in a given state) if the set of processes which did not terminate as yet is not empty, all are waiting for communication, and there exists amongst them no pair of processes which wait for each other (one for input and the other for output); and also, there exists no process in that set which would exit a loop by the distributed termination convention. Thus in a blocked state no process can proceed.

Given a program P and an assertion p, we say that P is *deadlock free* (relative to p) if no execution of P, starting in an initial state satisfying p, can reach a state in which P becomes blocked.

In order to be able to prove the absence of deadlock (and abortion) we first strengthen our proof rule for i/o guarded repetition. The required extension incorporates the distributed termination convention, amounting to the fact that the repetitive command is exited in case we can partition the set of indices of all guards in the command into two disjoint sets: The one (to be denoted by A) contains the indices of those guards which contain a true boolean part b_j, and refer in their i/o part to a process that has terminated; the other set (to be denoted by B) contains indices of guards with false boolean part b_j. This information will be collected in the post-condition of the loop in a way similar to that of the inclusion into the post-condition of the negation of all guards in the sequential case. Notice that no guard in either A or B can be passed.

To express this information, we introduce local propositional variables End^i_j, $i \neq j$, $1 \leq i, j \leq n$, with the following interpretation: End^i_j holds iff P_i "assumes" that P_j terminated. All these propositional variables have false as their initial truth value. When they are included in some assertion with true as their truth value, it will be only due to a loop exit in some process. In the proof (but not in the program) this change of value will be described as if assignments take place upon loop exit. End^i_j can *only* be used in proofs concerning P_i.

The new rule for i/o guarded repetition becomes now

R2' guarded repetition

$$\frac{\{p \wedge b_j\}\ \alpha_j\ \{r_j\},\ \{r_j\}\ S_j\ \{p\},\ j=1, \ldots, m}{\{p\} * [\square\ (j=1, \ldots, m)\ b_j, \alpha_j \rightarrow S_j]\ \{p \wedge \bigwedge_{j=1} (\neg b_j \vee End^i_{k_j})\}}$$

Here k_j denote the index of the process referred to by α_j, and i denotes the index of the process containing the loop.

The propositional variables End^i_j are used in general in the global invariant I. Therefore, we must add a clause to the definition of cooperation. This clause will take care that the invariant is preserved upon exit of an i/o guarded repetition by some process, when the corresponding processes (referred in the guards in A) have terminated, as expressed by using their post-conditions.

The clause to be added is the following:

iii) Let C be the set of indices of all processes referred to in α_j for $j \in A$.

Then

$$\bigwedge_{j \in C} post(P_j) \wedge pre(S) \wedge \bigwedge_{j \in A} b_j \wedge \bigwedge_{j \in B} \neg b_j \wedge I \rightarrow I[true/End^i_j]_{j \in C}$$

holds, where S denotes a subprogram of P_i of the form

$$*[\,\Box\,(j=1, \ldots, m)\; b_j, \alpha_i \rightarrow S_j\,].$$

Also, A, B have a meaning as described above, $A \cap B = \emptyset$, $A \cup B = \{1, \ldots, m\}$. Here $I[true/End^i_j]_{j \in C}$ stands for the formula obtained from I by a simultaneous substitution of true for End^i_j for $j \in C$.

$\Box$

Next, we proceed with the formal definitions required in order to formulate the theorem about deadlock freedom. We assume that a specific proof outline for each process is given.

The following definition intends to characterize those situations in which execution can proceed smoothly; such situations will not have to be considered in the proof of deadlock freedom, since they imply that the program is not blocked in that situation.

<u>Definition:</u> An m-tuple of assertions $<p_1, \ldots, p_m>$ <u>matches</u> iff

i) $m=2$ and p_1, p_2 are pre-assertions of a matching pair of bracketed sections,

or ii) for $i=1, \ldots, m-1, p_i$ is post (P_{k_i}), p_m is pre $(S) \wedge \bigwedge_{j \in A} b_j \wedge \bigwedge_{j \in B} \neg b_j$ for S a subprogram in P_m, A and B as above, $|A| = m-1$, $\{k_1, \ldots, k_{m-1}\}$=the set of indices of processes referred to by some α_j, $j \in A$.

□

Note that the second clause corresponds to the already discussed termination convention of i/o guarded repetition.

Next, we proceed with the definition of the blocking concept as it applies here. Remember that a blocked tuple of assertions is intended to indicate states in which the program deadlocks or aborts. We will have to prove that no such blocked tuple of assertions can simultaneously hold in any state which can be reached by an execution starting in a state satisfying the pre-condition of the whole program.

<u>Definition:</u> An n-tuple $<p_1, \ldots, p_n>$ of assertions is <u>blocked</u> iff all of the following conditions are satisfied:

i) each assertion p_i is either a pre-assertion of a bracketed section of P_i, or post (P_i) or pre $(S) \wedge \bigwedge_{j \in A} b_j \wedge \bigwedge_{j \in B} \neg b_j$, with S, A, B as considered above.

ii) at least one p_i is not post (P_i) (i.e. not <u>all</u> processes terminated already).

iii) no subtuple $<p_{k_1}, \ldots, p_{k_m}>$ of $<p_1, \ldots, p_n>$ matches.

□

<u>Theorem</u>: Given a proof $\{p\}$ P $\{q\}$ with global invariant I, then P is deadlock free (relative to p) if for every blocked n-tuple $<p_1, \ldots, p_n>$, $\neg(\bigwedge_{i=1}^{n} p_i \wedge I)$ holds.

□

Hence, in order to prove that P is deadlock free, we have to identify all blocked tuples of assertions, and the global invariant I should be such that a contradiction can be derived from the conjunction of the invariant and the given blocked tuple. The operational meaning of this contradiction is as follows: there is no moment during execution at which control of every P_i reaches a point in which the assertion p_i (taken from the given blocked tuple) holds. If the conditions of the theorem hold then execution can proceed smoothly (possibly forever).

The theorem above is a consequence of the following one, whose proof is part of the proof of the soundness and completeness of the system.

<u>Theorem</u>: Let a proof $\{p\}$ P $\{q\}$ be given. If during execution each P_i is about to execute a statement with a pre-assertion pre_i, then $\bigwedge_{i=1}^{n} pre_i$ is satisfied by the (global) state at that moment.
If none of the processes is within a bracketed section then I holds.

□

Sometimes a stronger invariant will be needed for proving deadlock freedom than for proving partial correctness.

Now we apply these concepts to the partition example considered in section 3. We refer to the proof presented there.

In order to prove the absence of deadlock in this program, we have to strengthen the invariant GI to include

$$GI' \equiv End_1^2 \rightarrow mx \leq x,$$

and add $mx > x$ to the pre-condition of the two bracketed sections in the loop of P_1, as well as adding $mx \leq x$ to the post-condition of P_1. Also, the use of the strong version of i/o guarded repetition rule implies that End_1^2 is added to post (P_2). In showing the invariance of GI', the only case that has to be checked is the loop exit of P_2; since we can assume post (P_1), GI' holds indeed.

Next, we consider all blocked pairs <p, q> of assertions, and show that their conjunction with $GI \wedge GI'$ is contradictory.

In all cases which do not involve the post-assertions of P_1 or P_2 the contradiction is reached by observing that all blocked pairs imply different parities of the ℓ_i's whereas GI implies $\ell_1 = \ell_2$.

For example, with p as the pre-condition of P_1's first bracketed a section and q as the pre-assertion of P_2's first bracketed section inside its loop , we have

$$\ell_1 = 0 \wedge \text{odd}(\ell_2) \wedge \ell_1 = \ell_2$$

which is contradictory.

The only other case with an essentially different proof, which does not use the fact that GI implies $\ell_1 = \ell_2$, is when p denotes the pre-assertion of P_1's first bracketed section inside its loop and P_2 has terminated, i.e. q contains End_1^2 (amongst others). Then we have

$$mx > x \wedge (End_1^2 \supset mx \leq x) \wedge End_1^2$$

which again is contradictory.

Note that only here the additional invariant GI′ was used.

Returning to the gcd program from section 3, we will prove now that there is no other blocking possibility in that program besides the intended one (as stated in the explanation to the program).

$$\text{Let } GI' \equiv \bigwedge_{i=1}^{n} (End_{i+1}^{i} \equiv End_{i}^{i+1}).$$

We shall prove the invariance of GI′. By using the strong repetition rule R_2', we get that each post (P_i) implies

$$End_{i+1}^{i} \wedge End_{i-1}^{i}$$

(by considering the third and fourth alternatives of each loop). Initially GI′ holds, since all End_j^i are initially false.

All we have to consider now is a loop exit of some P_i, and then post$(P_{i+1}) \wedge$ post (P_{i-1}) may be assumed, i.e. we have to verify

$$GI' \wedge End_i^{i+1} \wedge End_i^{i-1} \rightarrow GI' \,[true/End_{i+1}^{i},\ true/End_{i-1}^{i}]$$

which trivially holds.

A simple consequence of GI′ is

$$(**) \quad \bigwedge_{i \neq j} End_j^i \equiv End_i^j \, .$$

The meaning of this condition is that either all processes have terminated, or none did.

Any blocked tuple of assertions (besides the one considered in section 3) implies that some of the assertions in the tuple are post (P_i) for some $1 \leq i \leq n$, i.e. that some (but not all) of the processes terminated, which clearly contradicts (**).

In order to conclude that the situation considered in section 3 does occur (i. e. is inevitably reachable) we have to use:

i) A well-foundedness argument to prove the absence of infinite computations.

ii) The distributed termination pattern theorem [F] to show that the program does not terminate, since its termination dependency graph is cyclic.

iii) The absence of other blocked tuple of assertions than the one considered in section 3, as was shown above.

The proof of (i) is beyond the scope of the paper so is omitted.

5. CONCLUSION AND COMPARISON WITH RELATED WORK

We have presented a proof system for partial correctness and absence of deadlock in CSP programs. Now that we have gone through all stages of its development, it may be useful to compare our proof system with related Hoare-style proof systems dealing with concurrency.

Our final rule for parallel composition is related to the corresponding rule of Owicki and Gries [OG2] in which the premise is that proofs for component programs are interference free, in that both are metarules involving comparison between proofs. However, it also relates to the system presented is Owicki and Gries [OG1], which deals with shared resources and critical sections, in that a global invariant I is used which must be preserved by each pair of matching bracketed sections. This suggests that any pair of matching bracketed sections constitutes a (semantically determined) critical section using a resource. The fact that only <u>one</u> global invariant is used implies that exactly one resource is associated with each program. Such a resource can be used only by pairs of processes; in fact, in case these pairs of processes are mutually disjoint, several pairs of processes can use this resource at the same time.

The fact that we deal with simultaneity as a synchronization primitive relates in turn our approach to that of Mazurkiewicz [M] where simultaneous <u>await</u> statements are considered. However, since message passing is absent in his language, the issue of cooperation ~ a direct consequence of the disjointness of CSP processes ~ does not arise. Also, since in [M] shared variables are used, his proofs have to be checked for interference freedom, whereas in our system the property of disjointness of processes preserved in the component proofs implies that the need for testing upon interference freedom does not arise.

One of the features of our system is that the cooperation test requires us to supply new formal proofs which do not constitute a part of the (sequential) proof outlines. This phenomenon is also present in [OG2] where new proofs are needed to show interference freedom. These proofs can be viewed as global reasoning since they involve more than one process. However, in our case, unlike that of [OG2], we can control the size of these proofs by having the liberty of choosing the bracketed sections ourselves. The bigger the bracketed sections the more sizeable proofs have to be carried out. The (to be published) proof of relative completeness of our system implies that we can always choose bracketed sections of the form $\alpha;S$ where S is an assignment (for updating the local history of communications) thus reducing global reasoning.

Our method suffers from the same drawback as the one presented in [OG]; in the worst case the test for cooperation, e.g. for the case of two processes, can involve as many as m_1*m_2 checks, where m_1 and m_2 are proportional to the lengths of the component programs. The same problem can arise in proofs of absence of deadlock. However, in practice the number of cases is significantly smaller, and often several of them can be trivially established, as is the case in testing cooperation between syntactically matching but semantically not matching pairs. For example, in our proof for the partitioning program 8 cases had to be established in the cooperation test and 15 for the proof of absence of deadlock, but only 4 cases have a not immediate proof in the cooperation test and only one such case occurs in the proof of absence of deadlock.

REFERENCES

(1) [ADF] K.R. Apt, W.P. de Roever, N. Francez: Weakest precondition semantics for communicating processes. To appear.

(2) [CM] K. M. Chandy, J. Misra: An axiomatic proof technique for networks of communicating processes. TR-98, Dept. of Computer Science, Univ. of Texas at Austin, 1979.

(3) [D1] E. W. Dijkstra: A personal summary of the Gries-Owicki theory. EWD-554, 1976.

(4) [D2] E. W. Dijkstra: A correctness proof for communicating processes - a small exercise. EWD-607, 1977.

(5) [F] N. Francez: On achieving distributed termination. Proceedings of the international symposium on semantics of concurrent computations, Evian, France. Lecture Notes in Computer Science 70 (G. Kahn-ed.), Springer - Verlag, 1979.

(6) [FLHR] N. Francez, C.A.R. Hoare, D. J. Lehmann, W. P. de Roever: Semantics of nondeterminism, concurrency and communication. To appear in JCSS, 1979.

(7) [FP] N. Francez, A. Pnueli: A proof method for cyclic programs, Acta Informatica 9, 1978.

(8) [FR] N. Francez, M. Rodeh: Achieving distributed termination without freezing. Submitted for publication, 1979.

(9) [H1] C.A.R. Hoare: Towards a theory of parallel programming in: Operating Systems Techniques (C.A.R. Hoare, R. Perrot-eds.), Academic Press, 1972.

(10) [H2] C.A.R. Hoare: Communicating sequential processes. CACM 21, 8, 1978.

(11) [HW] J. H. Howard: Proving monitors. CACM 19, 5, 1976.

(12) [L] L. Lamport: Proving the correctness of multiprocess programs. IEEE Transactions on Software Engineering 3(2), 1977.

(13) [M] A. Mazurkiewicz: A complete set of assertions on distributed systems. Institute of Computer Science, Polish Academy of Science, 1979.

(14) [O] S. S. Owicki: A consistent and complete deductive system for the verification of parallel programs. Proc. 8th ACM Symp. on Theory of Computing, 1976.

(15) [OG1] S. S. Owicki, D. Gries: Verifying properties of parallel programs: an axiomatic approach, CACM 19, 5, 1976.

(16) [OG2] S. S. Owicki, D. Gries: An axiomatic proof technique for parallel programs I. Acta Informatica 6, 1976.