A Proof System for Communicating Sequential Processes

KRZYSZTOF R. APT
University of Rotterdam
NISSIM FRANCEZ
Technion—Israel Institute of Technology
and
WILLEM P. DE ROEVER
University of Utrecht

An axiomatic proof system is presented for proving partial correctness and absence of deadlock (and failure) of communicating sequential processes. The key (meta) rule introduces cooperation between proofs, a new concept needed to deal with proofs about synchronization by message passing. CSP's new convention for distributed termination of loops is dealt with. 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.

Key Words and Phrases: Hoare-style proof rules, partial correctness, global invariant, cooperating proofs, CSP, communicating processes, concurrency, absence of deadlock, blocking
CR Categories: 4.32, 5.24

1. INTRODUCTION AND PRELIMINARIES

1.1 Introduction

This paper presents a proof system for CSP, a language for *C*ommunicating *S*equential *P*rocesses due to Hoare [11]. 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 the way synchronization and message passing can be employed in a programming language, 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

This work was supported by the National Science Foundation under Grant MCS 78-673 during the authors' stay at the University of Southern California in Los Angeles.
Authors' addresses: K. R. Apt, Faculty of Economics, University of Rotterdam, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands; N. Francez, Department of Computer Science, Technion—Israel Institute of Technology, Haifa, Israel; W. P. de Roever, Department of Computer Science, University of Utrecht, P.O. Box 80.002, 3508 TA Utrecht, The Netherlands.

to be taken care of:

(1) CSP stresses *simultaneity* rather than mutual exclusion as a synchronization mechanism by using simultaneous communication as the only means of synchronization.
(2) The two communication primitives of CSP, input and output commands, can function as a choice mechanism by acting as guards in (possibly nondeterministic) guarded choices and repetitions.
(3) CSP focuses 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 postassertion referring only to variables of the process in which the statement appears. 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 postassertion of an input command cannot be validated since the assertions of its corresponding output command occur in another sequential component. Such 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 failure; it generalizes the notion of blocking [18] to an environment in which some processes, which are intended to terminate, fail to communicate.

The distinction between cooperation and combat functioned as an almost philosophical guideline in our efforts. Examples are 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 concept needed to deal with synchronization by message passing: *cooperation* between proofs. These remarks are elaborated in the last section.

This proof system derives from various related work:

(1) Owicki's and Lamport's landmark in the proof theory of concurrent processes [13, 17, 18]. We benefited also from relative completeness proofs due to Owicki and to Mazurkiewicz [15, 16].
(2) 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 [7]. 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 [2], which brought the semantics closer to a proof system.

(3) The concept of assumption/commitment pairs (interface predicates) as introduced by Francez and Pnueli [8] to characterize the assumptions which a process has to make about the behavior 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 behavior. Thus, assumption/commitment pairs are assertions which express the cooperation between a process and its environment.

While writing this paper, we learned about related work by Carl Hauser (in preparation) and Chandy and Misra [4]. Some time after submission of the paper we were informed of independent, very much related work by G. M. Levin [14], briefly discussed in the last section.

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—one of a distributed partition algorithm due to W. Feijen and described in Dijkstra [5] (our proof differs from that of Dijkstra), and the other of an algorithm for the distributed computation of the greatest common divisor of n natural numbers taken from Francez and Rodeh [9]. Section 4 generalizes the proof system to freedom from deadlock and failure and contains some applications. The last section contains an assessment and comparison of our method with related Hoare-like proof systems for other concurrent languages.

1.2 Preliminaries: Definition of CSP

Full details of CSP are contained in [11]. For our purpose the following informal description of its syntax and meaning suffices:

(1) The basic command of CSP is $[P_1 \| \cdots \| P_n]$ expressing *concurrent* execution of processes $P_1, \ldots, P_n$, $n \geq 2$.

(2) Every P_i refers to a statement S_i, as indicated by $P_i :: S_i$. No S_i contains variables subject to change in S_j $(i \neq j)$.

(3) *Communication* between P_i and P_j $(i \neq j)$ is expressed by the receive and send primitives $P_j?x$ and $P_i!x$, respectively. *Input* command $P_j?x$ (in S_i) expresses a request to P_j to assign a value to the (local) variable x of P_i. *Output* command $P_i!y$ (in S_j) expresses a request to P_i to receive a value from P_j. Execution of $P_j?x$ in S_i and $P_i!y$ in S_j is synchronized ("P_i waits at $P_j?x$ until P_j is ready at $P_i!y$, and vice versa," as the lingo goes) and results in assigning the value of y to x.

(4) *Guarded commands*: The case of two guarded possibilities is used to illustrate the command structure. Let guards B_i denote Boolean expressions $i = 1, 2$. "□" denotes the guarded command separator; ";" denotes sequential composition; and "skip" is a statement with no effect.

Guarded *selection*: $[B_1 \to S_1 \,\square\, B_2 \to S_2]$ *fails* for $B_1 \vee B_2 = \textbf{false}$, and leads to (possibly nondeterministic) selection of S_i for execution if $B_i = \textbf{true}$.

Guarded *iteration*: $*[B_1 \to S_1 \,\square\, B_2 \to S_2]$ *terminates* for $B_1 \vee B_2 = \textbf{false}$, and otherwise executes $[B_1 \to S_1 \,\square\, B_2 \to S_2]$; $*[B_1 \to S_1 \,\square\, B_2 \to S_2]$.

CSP's main feature is that the I/O commands $P_j?x$ and $P_i!y$ can also be used as guards. As an expression $P_j?x$ (respectively, $P_i!y$) evaluates to **false** in case P_j (respectively, P_i) has terminated. For example,

$$[P_1 :: [P_2?x \to \text{skip} \,\square\, P_2!x \to \text{skip}] \,\|\, P_2 :: \text{skip}]$$

leads to failure of P_i, while

$$[P_1 :: *[P_2?x \to \text{skip} \,\square\, P_2!x \to \text{skip}] \,\|\, P_2 :: \text{skip}]$$

properly terminates.

Also, as an expression, $P_j?x$ (respectively, $P_i!y$) evaluates to **true** if synchronization occurs with a matching output (respectively, input) command or guard. For example,

$$[P_1 :: [P_2?x \to \text{skip} \,\square\, P_2!x \to \text{skip}] \,\|\, P_2 :: [P_1?y \to \text{skip} \,\square\, P_1!y \to \text{skip}]]$$

has the same effect as executing $x := y$ or $y := x$ nondeterministically, and

$$[P_1 :: *[P_2?x \to \text{skip}] \,\|\, P_2 :: P_1!0]$$

has the same effect as executing $x := 0$ just once.

In Hoare's conception of CSP only finite processes are considered; thus

$$[P_1 :: *[P_2?x \to \text{skip}] \,\|\, P_2 :: *[P_1!0 \to \text{skip}]],$$

so-called *infinite chattering*, is considered a semantic error.

Using the CSP guards, the guarded commands generalize as follows:

(1) A guard may be a Boolean expression, an I/O command, or a combination of both (separated by ";"). A Boolean guard is *passable* if it is true; an I/O command is passable when a corresponding I/O command in the process addressed is ready; and a combination is passable if each of its components is passable.
(2) A guarded selection fails in the case in which all guards are false.
(3) A guard is false in one of the following cases:
    (i) It is a Boolean expression 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.
(4) "*" denotes a repetitive construct. Repetition continues as long as there exists a passable guard and terminates when all guards are false.

Guarded commands (i.e., selection or repetition) introduce the possibility that more than one matching pair of I/O commands occurs; for instance, in the example below the first communication of P_1 can be either with P_2 or with P_3,

but not simultaneously with both: $P :: [P_1 \parallel P_2 \parallel P_3]$, where

$$P_1 :: [P_2?x \rightarrow S_1 \,\square\, b_1;\ P_3!y \rightarrow S_2];\ *[b_2;\ P_2!u \rightarrow S_3 \,\square\, b_3;\ P_3?u \rightarrow S_4]$$
$$P_2 :: *[P_1?s \rightarrow S_5 \,\square\, P_1!t \rightarrow S_6 \,\square\, P_3?s \rightarrow S_7]$$
$$P_3 :: P_1?z;\ *[b_1 \rightarrow S_8 \,\square\, b_2 \rightarrow S_9].$$

Finally, to avoid some cumbersome notational problems in Section 4, we consider in this paper only guarded commands of which all the guards either contain an I/O command or are all Boolean.

2. THE PROOF SYSTEM

We intend to reason about CSP programs in a manner analogous to the work of Owicki and Gries [18]. *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 [17] and [18], 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 for 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 one to deduce any postassertion q of the input command whatsoever. However, any q thus introduced will later (when proofs are tested for cooperation) be checked against some postassertion 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*

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

The meaning of this rule is that the postassertion of an I/O guarded selection

must be established along each possibly selected path. We discuss later the problem of paths never selected.

R2. *I/O Guarded Repetition*

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

Note that this rule does not take into account the full exit conditions of the loop. We shall return to this problem at the end of the section.

Subsequently we use the following well-known axioms and proof rules:

A3. *Assignment*

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

A4. *Skip*

$$\{p\}\text{skip}\{p\}.$$

R3. *Alternative Command*

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

R4. *Repetitive Command*

$$\frac{\{p \wedge b_i\}S_i\{p\},\ i = 1, \ldots, m}{\{p\}^*[\square(i = 1, \ldots, m)\ b_i \rightarrow S_i]\{p \wedge \neg(b_1 \vee \cdots \vee b_m)\}}.$$

R5. *Composition*

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

R6. *Consequence*

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

R7. *Conjunction*

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

Using these axioms and proof rules, we can establish proofs for formulas of the form $\{p\}P_i\{q\}$, where P_i is a process. Each such proof can be represented, as in [18], by a *proof outline* 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 always refers 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 $[P_1 \| \cdots \| P_n]$ using the proofs concerning programs P_i, $i = 1, \ldots, n$. This rule has the following form:

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

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 $\{p_i\}P_i\{q_i\}$ contain no variables subject to change in P_j for $i \neq j$;

(ii) $\{\text{pre}_1 \wedge \text{pre}_2\}P_j?x \parallel P_i!y\ \{\text{post}_1 \wedge \text{post}_2\}$ holds whenever $\{\text{pre}_1\}P_j?x\{\text{post}_1\}$ and $\{\text{pre}_2\}P_i!y\{\text{post}_2\}$ are taken from the proofs of $\{p_i\}P_i\{q_i\}$ and $\{p_j\}P_j\{q_j\}$, respectively.[1]

We shall need the following axioms to establish cooperation:

A5. *Communication*

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

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

A6. *Preservation*

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

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

Note that A2 and A4 are subsumed by A6. We also need the following proof rule, needed to eliminate auxiliary variables from the preassertions.

R8. *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

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

where $P_1 :: P_2!x$, $P_2 :: P_1?y; P_3!y$, and $P_3 :: P_2?u$.

Here are the proof outlines:

$$\{x = z\}P_2!x\{x = z\},$$
$$\{\textbf{true}\}P_1?y\{y = z\}; P_3!y\{y = z\},$$
$$\{\textbf{true}\}P_2?u\{u = z\}.$$

The proofs clearly cooperate; for example,

$$\{x = z\}P_2!x \parallel P_1?y\{x = z \wedge y = z\}$$

can be derived as follows. By the communication axiom $\{\textbf{true}\}P_2!x \parallel P_1?y\{x = y\}$, so by the consequence rule, $\{x = z\}P_2!x \parallel P_1?y\{x = y\}$. On the other hand, by the preservation axiom, $\{x = z\}P_2!x \parallel P_1?y\{x = z\}$; so by the conjunction rule, $\{x = z\}P_2!x \parallel P_1?y\{x = y \wedge x = z\}$. Finally, $\{x = z\}P_2!x \parallel P_1?y\{x = z \wedge y = z\}$ by the consequence rule. Thus we get $\{x = z\}[P_1 \parallel P_2 \parallel P_3]\{x = z \wedge y = z \wedge u = z\}$.

[1] Such pairs of I/O instructions will be said to be *syntactically matching*.

Now by applying the consequence rule, we get $\{x = z\}[P_1 \parallel P_2 \parallel P_3]\{x = u\}$, from which the claim follows by applying the substitution rule, and substituting x for z in the precondition. □

This approach fails when dealing with programs in which some output commands do 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, $\{\textbf{true}\}[P_1 \parallel P_2 \parallel P_3]\{x = 0\}$ holds. However, this cannot be proved in the above system, for any such proof would require establishing both $\{\textbf{true}\}P_3!y\{x = 0\}$ and $\{\textbf{true}\}P_3?y\{x = 0\}$. 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 $\{p\}P_i?x \parallel P_j!y\{q\}$ for arbitrary p and q?" That this is not the case follows from the cooperation test. Using A5, the axiom of communication, and A6, the axiom of preservation, only formulas of the form $\{r\}P_i?x \parallel P_j!y\{x = y \wedge r\}$ 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 comparison of *all* I/O pairs which syntactically match, even though some syntactically possible communications 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, $\{\textbf{true}\}[P_1 \parallel P_2]\{x = 2\}$ holds. To prove this, we are forced to use $x = 2$ as the postassertion 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). □

In general, syntactic matching of a pair of I/O instructions does not imply that this communication will ever take place, i.e., it does not imply their *semantic* match. In order to take care that semantically unmatched 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, need not be program variables either; in general auxiliary variables are needed).

For example, consider the following program sections:

$$\cdots P_2?x;\ i := i + 1 \cdots \parallel \cdots P_1!y;\ j := j + 1 \cdots$$

where i and j count the number of communications actually occurring in each process, and let the criterion for semantic matching be $i = j$. However, $i = j$ is not a global *invariant* since the two assignments to i and j will not necessarily be executed simultaneously, in contrast to the corresponding I/O commands which *are* executed simultaneously.

To resolve these difficulties, we must reduce the number of places where the global invariant should hold. This is 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 of Hoare [10] concerning resource invariants, where the global invariant does not need to hold within the critical sections. An analogous problem arises when dealing with monitor invariants [12].

Regarding the program sections just considered, the bracketing is

$$\cdots \langle P_2?x;\ i := i + 1\rangle \cdots \parallel \cdots \langle P_1!y;\ j := j + 1\rangle \cdots,$$

so that $i = j$ holds outside the brackets.

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

$$S_1;\ \alpha;\ S_2 \qquad \text{or} \qquad \alpha \rightarrow S_1,$$

and S_1 and S_2 do not contain any I/O statements. □

With each proof of $\{p\}[P_1 \parallel \cdots \parallel P_n]\{q\}$ we now associate a global invariant I and appropriate brackets. Therefore, the proof rule concerning parallel composition becomes the following:

R9. *Parallel Composition*

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

provided no variable free in I is subject to change outside a bracketed section.

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

Definition. Let $\langle S_1\rangle$ and $\langle S_2\rangle$ denote two bracketed sections from P_i and P_j $(i \neq j)$. We say that $\langle S_1\rangle$ and $\langle S_2\rangle$ *match* if S_1 and S_2 contain matching I/O commands. □

Definition. The proofs of the $\{p_i\}P_i\{q_i\}$, $i = 1, \ldots, n$, *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) $\{\text{pre}(S_1) \wedge \text{pre}(S_2) \wedge I\}\ S_1 \parallel S_2\{\text{post}(S_1) \wedge \text{post}(S_2) \wedge I\}$ holds for all matching pairs of bracketed sections $\langle S_1 \rangle$ and $\langle S_2 \rangle$. □

The following additional proof rules are used to establish cooperation:

R10. *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, and S_4 do not contain any I/O commands.

R11. *Arrow*

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

R10 and R11 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 R12, a slightly strengthened version of a rule from [18], for deleting assignments to auxiliary variables.

R12. *Auxiliary Variables.* Let AV be a set of variables such that $x \in \text{AV} \Rightarrow$ x appears in S' only in assignments $y := t$, where $y \in \text{AV}$. Then if q does not contain free any variables from AV, and S is obtained from S' by deleting all assignments to variables in AV,

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

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'.

$\{i = 0 \wedge j = 0\}$
$[\{i = 0\}$
$[\langle P_2?x\{x = 2\} \rightarrow i := 1\rangle\{x = 2 \wedge i = 1\}; \text{skip}\{x = 2\}$
$\square$
$\quad \langle P_2!0\{\textbf{true}\} \rightarrow i := 1\rangle\{i = 1\};$
$\quad\quad \langle P_2?x\ \{x = 1\}; i := 2\rangle\{x = 1 \wedge i = 2\}; x := x + 1\ \{x = 2\}$
$]\{x = 2\}$
$\parallel$
$[\{j = 0\}$
$\quad \langle P_1!2\{\textbf{true}\} \rightarrow j := 1\rangle\{j = 1\}\ \text{skip}\{\textbf{true}\}$
$\square$
$\quad \langle P_1?z\{z = 0\} \rightarrow j := 1\rangle\{z = 0 \wedge j = 1\};$
$\quad\quad \langle P_1!1\{\textbf{true}\}; j := 2\rangle\{j = 2\}$
$]\{\textbf{true}\}$
$]$
$\quad \{x = 2\}$

We choose $I \equiv (i = j)$. Cooperation is easily established. Note that $(i = 0 \wedge (z = 0 \wedge j = 1) \wedge I) \equiv \mathbf{false}$, so the bracketed sections containing $P_2?x$ and $P_1!1$ pass the cooperation test trivially. (One has for any S, $\{\mathbf{false}\}S\{\mathbf{false}\}$ by the preservation axiom, so $\{\mathbf{false}\}S\{p\}$ for any p by the consequence rule.) Hence, by the parallel composition rule, consequence rule, and auxiliary variables rule,

$$\{i = 0 \wedge j = 0 \wedge i = j\}[P_1 \parallel P_2]\{x = 2\}$$

holds. Applying the substitution rule we finally get

$$\{\mathbf{true}\}[P_1 \parallel P_2]\{x = 2\}. \quad \square$$

At this stage we return to the problem signaled earlier—namely, that of rule R2. Rule R2 alone 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 the b_i's may be false. Now that we introduce global invariants, we can settle this problem by expressing exit conditions in the global invariant I. As an illustration, let us prove

$$\{b\}[P_1 \parallel P_2]\{b\}$$

with

$$P_1 :: *[b;\ P_2?x \rightarrow b := \mathbf{false}] \qquad \text{and} \qquad P_2 :: \text{skip}.$$

We simply choose I to be b and take all other assertions true. The cooperation of proofs is voidly satisfied.

A slightly less trivial proof establishes $\{\mathbf{true}\}[P_1 \parallel P_2]\{\neg b\}$ with P_1 as above and $P_2 :: P_1!y$. In this case we have to express the fact that after the communication takes place, b turns false. To this purpose we introduce an auxiliary variable i.

We present the proof outlines for the bracketed programs

$$\{\mathbf{true}\}*[b;\ \langle P_2?x \rightarrow b := \mathbf{false}\rangle]\{\mathbf{true}\}$$
$$\{i = 0\}\langle P_1!y;\ i := 1\rangle\{i = 1\}.$$

We choose for I the formula $(i = 1 \rightarrow \neg b)$. Cooperation is easily established using the formation rule. By the parallel composition rule, consequence rule, and the auxiliary variables rule,

$$\{i = 0 \wedge (i = 1 \rightarrow \neg b)\}[P_1 \parallel P_2]\{\neg b\},$$

so finally, by the substitution rule, $\{\mathbf{true}\}[P_1 \parallel P_2]\{\neg b\}$.

These two examples have been given to indicate why rule R2 is sufficient for proofs of partial correctness. In Section 4 we discuss the problem of whether this rule is sufficient for proofs of deadlock freedom.

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' such that $|S| = |S'|$, $|T| = |T'|$, and every element of S' is smaller than any element of T'. The program P and its correctness proof are

inspired by Dijkstra [5]; however the proof presented here differs from Dijkstra's. $P :: [P_1 \parallel P_2]$, as given below, and $S \neq \emptyset$.

```
P1 :: mx := max(S);
      P2!mx; S := S − {mx};
      P2?x; S := S ∪ {x};
      mx := max(S);
      *[mx > x → P2!mx; S := S − {mx};
                 P2?x; S := S ∪ {x};
                 mx := max(S)
      ]

P2 :: P1?y; T := T ∪ {y};
      mn := min(T);
      P1!mn; T := T − {mn};
      *[P1?y → T := T ∪ {y};
                mn := min(T);
                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 proof of correctness 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 (by requiring $l_1 = l_2$);
(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 the 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) Assignments to the auxiliary variables l_1 and l_2.
(2) The pre- and postconditions required for a proof, taking into account deletions 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 GI may be violated within these units, but *not* outside these units.

Annotated text of P_1:

$\{|S| = n_1 > 0 \wedge S = S_0 \wedge \max(S) \in S \wedge l_1 = 0\} mx := \max(S);$
$\{mx \in S \wedge |S| = n_1 \wedge l_1 = 0\}$
$\langle P_2!mx;\ l_1 := l_1 + 1;\ \{mx \in S\} S := S - \{mx\}\rangle;$
$\{|S| = n_1 - 1 \wedge l_1 = 1\}$
$\langle P_2?x;\ l_1 := l_1 + 1;\ \{x \notin S\} S := S \cup \{x\}\rangle;$
$\{|S| = n_1 \wedge x \in S \wedge l_1 = 2\} mx := \max(S);$
LI_1: $\{|S| = n_1 \wedge mx = \max(S) \wedge x \leq \max(S) \wedge \mathrm{even}(l_1) \wedge l_1 \geq 2\}$
$*[mx > x \rightarrow \{mx \in S \wedge \mathrm{LI}_1\}\ \langle P_2!mx;\ l_1 := l_1 + 1;\ \{mx \in S\} S := S - \{mx\}\rangle;$
$\{|S| = n_1 - 1 \wedge \mathrm{odd}(l_1) \wedge l_1 \geq 2\}$
$\langle P_2?x;\ l_1 := l_1 + 1 \{x \notin S\};\ S := S \cup \{x\}\rangle;$
$\{|S| = n_1 \wedge x \in S \wedge \mathrm{even}(l_1)\} mx := \max(S)$
LI_1: $\{|S| = n_1 \wedge x \in S \wedge mx = \max(S) \wedge \mathrm{even}(l_1) \wedge l_1 \geq 2\}$
$]$
$\{\max(S) = x \wedge |S| = n_1 \wedge \mathrm{even}(l_1)\}$

Annotated text of P_2:

$\{|T| = n_2 \geq 0 \wedge T = T_0 \wedge l_2 = 0\}$
$\langle P_1?y;\ l_2 := l_2 + 1;\ \{y \notin T\} T := T \cup \{y\}\rangle;$
$\{|T| = n_2 + 1 \wedge l_2 = 1\} mn := \min(T);$
$\{|T| = n_2 + 1 \wedge mn = \min(T) \wedge l_2 = 1\}$
$\langle P_1!mn;\ l_2 := l_2 + 1;\ \{mn \in T\} T := T - \{mn\}\rangle;$
LI_2: $\{|T| = n_2 \wedge mn < \min(T) \wedge \mathrm{even}(l_2) \wedge l_2 \geq 2\}$
$*[\langle P_1?y \rightarrow l_2 := l_2 + 1;\ T := T \cup \{y\}\rangle;$
$\{|T| = n_2 + 1 \wedge \mathrm{odd}(l_2)\} mn := \min(T);$
$\{|T| = n_2 + 1 \wedge mn = \min(T) \wedge \mathrm{odd}(l_2) \wedge l_2 \geq 2\}$
$\langle P_1!mn;\ l_2 := l_2 + 1;\ T := T - \{mn\}\rangle$
LI_2: $\{|T| = n_2 \wedge mn < \min(T) \wedge \mathrm{even}(l_2) \wedge l_2 \geq 2\}$
$]$
$\{|T| = n_2 \wedge mn < \min(T)\}$

The global invariant GI:

$\mathrm{GI} \equiv S \cap T = \varnothing \wedge S \cup T = S_0 \cup T_0 \wedge l_1 = l_2 \wedge (\mathrm{even}(l_1) \wedge l_1 \geq 2 \rightarrow x < \min(T))$.

For the sake of the proof we assume that $\min(\varnothing) = +\infty$.

We restrict ourselves to proving cooperation between proofs for the first bracketed section of P_1 and P_2, and for the second bracketed section of P_1 and P_2; the customary kind of sequential reasoning is omitted. Proofs for the cooperation between the third bracketed section and the fourth are actually identical and are omitted. Proofs for syntactically matching but semantically nonmatching sections are trivial; for instance, the first section of P_1 and the third of P_3 are trivially cooperating since $\neg$GI holds (in this case $\neg(l_1 = 0 \wedge l_2 \geq 2 \wedge l_1 = l_2)$). 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. We have $\mathrm{pre}_1 \equiv mx \in S \wedge |S| = n_1 \wedge l_1 = 0$, and $\mathrm{pre}_2 \equiv |T| = n_2 \wedge T = T_0 \wedge l_2 = 0$. Also, $\mathrm{post}_1 \equiv |S| = n_1 - 1 \wedge l_1 = 1$ and $\mathrm{post}_2 \equiv |T| = n_2 + 1 \wedge l_2 = 1$.

We must prove

$\{\mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}$
$P_2!mx;\ l_1 := l_1 + 1;\ S := S - \{mx\} \parallel P_1?y;\ l_2 := l_2 + 1;\ T := T \cup \{y\}$
$\{\mathrm{post}_1 \wedge \mathrm{post}_2 \wedge \mathrm{GI}\}$.

By the communication and preservation axioms,

$$\{\mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}\, P_2!mx \parallel P_1?y\{mx = y \wedge \mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}.$$

Precondition of section $l_1 := l_1 + 1; S := S - \{mx\}; l_2 := l_2 + 1; T := T \cup \{y\}$ w.r.t. postcondition $\mathrm{post}_1 \wedge \mathrm{post}_2 \wedge \mathrm{GI}$ is

$$\begin{aligned} l_1 = l_2 = 0 \wedge y \notin T \wedge |T| = n_2 \wedge mx \in S \wedge |S| \\ = n_1 \wedge S \cap T = \varnothing \wedge S \cup T = S_0 \cup T_0, \end{aligned}$$

which is implied by $\{mx = y \wedge \mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}$. Therefore the formation rule yields the result, since

$$\{\mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}\, P_2!mx \parallel P_1?y\{mx = y \wedge \mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}$$

and

$$\begin{aligned} \{mx = y \wedge \mathrm{pre}_1 \wedge \mathrm{pre}_2 \wedge \mathrm{GI}\}\, l_1 := l_1 + 1; S := S - \{mx\}; \\ l_2 := l_2 + 1; T := T \cup \{y\}\{\mathrm{post}_1 \wedge \mathrm{post}_2 \wedge \mathrm{GI}\} \end{aligned}$$

hold.

Proof of cooperation between second bracketed sections. We have $\mathrm{pre}_1' \equiv |S| = n_1 - 1 \wedge l_1 = 1$ and $\mathrm{pre}_2' \equiv |T| = n_2 + 1 \wedge mn = \min(T) \wedge l_2 = 1$. Also $\mathrm{post}_1' \equiv |S| = n_1 \wedge x \in S \wedge l_1 = 2$ and $\mathrm{post}_2' \equiv |T| = n_2 \wedge mn < \min(T) \wedge \mathrm{even}(l_2) \wedge l_2 \geq 2$.

We must prove

$$\begin{aligned} &\{\mathrm{pre}_1' \wedge \mathrm{pre}_2' \wedge \mathrm{GI}\} \\ &P_2?x; l_1 := l_1 + 1; S := S \cup \{x\} \parallel P_1!mn; l_2 := l_2 + 1; T := T - \{mn\} \\ &\{\mathrm{post}_1' \wedge \mathrm{post}_2' \wedge \mathrm{GI}\}. \end{aligned}$$

By the communication axiom and preservation axiom,

$$\{\mathrm{pre}_1' \wedge \mathrm{pre}_2' \wedge \mathrm{GI}\}\, P_2?x \parallel P_1!mn\{mn = x \wedge \mathrm{pre}_1' \wedge \mathrm{pre}_2' \wedge \mathrm{GI}\},$$

since $\mathrm{odd}(l_1)$. Now observe that

$$\begin{aligned} &\{mn = x \wedge \mathrm{pre}_1' \wedge \mathrm{pre}_2' \wedge \mathrm{GI}\} \\ &l_1 := l_1 + 1; S := S \cup \{x\}; l_2 := l_2 + 1; T := T - \{mn\} \\ &\{\mathrm{post}_1' \wedge \mathrm{post}_2' \wedge \mathrm{GI}\} \end{aligned}$$

holds. Note that $x < \min(T)$ in the postassertion 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

$$\begin{aligned} &\{|S| = n_1 > 0 \wedge S = S_0 \wedge |T| = n_2 \geq 0 \wedge T \\ &\quad = T_0 \wedge l_1 = 0 \wedge l_2 = 0 \wedge \mathrm{GI}\} \\ &[P_1' \parallel P_2'] \\ &\{\mathrm{LI}_1 \wedge \mathrm{LI}_2 \wedge \mathrm{GI}\} \end{aligned}$$

where P_1' and P_2' are the modified versions of P_1 and P_2. From this we obtain

$$\begin{aligned}&\{|S| = n_1 > 0 \wedge S = S_0 \wedge |T| = n_2 \geq 0 \wedge T\\&= T_0 \wedge S \cap T = \varnothing \wedge l_1 = 0 \wedge l_2 = 0\}\\&[P_1' \| P_2']\\&\{|S| = n_1 \wedge |T| = n_2 \wedge S \cap T = \varnothing \wedge S \cup T\\&= S_0 \cup T_0 \wedge \max(S) < \min(T)\}.\end{aligned}$$

Now by dropping the assignments to l_1 and l_2 and subsequently substituting 0 for l_1 and l_2 in the precondition, we get the desired formula.

3.2. Distributed Computation of the Greatest Common Divisor of n Numbers

As another example, we 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 [9]. This program has the property that when all processes reach 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 than the corresponding program that will terminate when all processes reach final states.
(2) There exists an automatic transformation transforming every such blocked program into an equivalent terminating program. See [6, 9] for details of 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 [13]) than just partial correctness.

The program P consists of n parallel processes arranged in a ring configuration, where each process P_i communicates with its own immediate neighbors 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 neighbor, and uses flags rsl (*r*eady to *s*end *l*eft) and rsr (*r*eady to *s*end *r*ight) to avoid sending x_i again before it is 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. When such a number is received from a neighbor process, the number is compared to x_i. If x_i is larger, it is then updated according to Euclid's rule, and the rsl and rsr flags are set on. Otherwise nothing happens. Two auxiliary variables, rcvl (*rec*eiv*e*d from *l*eft) and rcvr (*rec*eiv*e*d from *r*ight), are included for the sake of the proof.

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

In the following annotated text for P_i, LI_i is the loop invariant of P_i which serves also as the precondition and postcondition for the body of the main loop:

Annotated text of P_i:

$\{x_i = \sigma_i > 0 \wedge \mathrm{rsl}_i \wedge \mathrm{rsr}_i\}$
$*[\{\mathrm{LI}_i\}$
$\quad \langle \mathrm{rsl}_i;\ P_{i-1}!x_i \rightarrow \mathrm{rsl}_i := \mathbf{false};\ \mathrm{rcvl}_i := \mathbf{false}\ \{\mathrm{LI}_i\}$
$\square$
$\quad \langle \mathrm{rsr}_i;\ P_{i+1}!x_i \rightarrow \mathrm{rsr}_i = \mathbf{false};\ \mathrm{rcvr}_i := \mathbf{false}\rangle\ \{\mathrm{LI}_i\}$
$\square$
$\langle P_{i-1}?y_i \rightarrow \mathrm{rcvl}_i := \mathbf{true};$
$\quad [y_i \geq x_i \rightarrow \mathrm{skip}$
$\quad \square$
$\quad y_i < x_i \rightarrow [y_i \mid x_i \rightarrow x_i := y_i$
$\quad\quad \square$
$\quad\quad y_i \not| x_i \rightarrow x_i := x_i \bmod y_i$
$\quad\quad];\ \{\mathrm{LI}_i\}\ \mathrm{rsr}_i := \mathbf{true};\ \mathrm{rsl}_i := \mathbf{true}$
$\quad] \rangle \{\mathrm{LI}_i\}$
$\square$
$\langle P_{i+1}?z_i \rightarrow \mathrm{rcvr}_i := \mathbf{true};$
$\quad [z_i \geq x_i \rightarrow \mathrm{skip}$
$\quad \square$
$\quad z_i < x_i \rightarrow [z_i \mid x_i \rightarrow x_i := z_i$
$\quad\quad \square$
$\quad\quad z_i \not| x_i \rightarrow x_i := x_i \bmod z_i$
$\quad\quad];\ \{\mathrm{LI}_i\}\ \mathrm{rsr}_i := \mathbf{true};\ \mathrm{rsl} := \mathbf{true}$
$\quad] \rangle \{\mathrm{LI}_i\}$
$]$

The global invariant GI:

$$\mathrm{GI} \equiv \bigwedge_{i=1}^{n} [\neg \mathrm{rsl}_i \rightarrow (z_{i-1} = x_i \wedge \mathrm{rcvr}_{i-1})$$
$$\wedge \neg \mathrm{rsr}_i \rightarrow (y_{i+1} = x_i \wedge \mathrm{rcvl}_{i+1})$$
$$\wedge \gcd(x_1, \ldots, x_n) = \gcd(\sigma_1, \ldots, \sigma_n)].$$

GI establishes the correct sending and receiving relationship between any triple P_{i-1}, P_i, P_{i+1}, and also establishes 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 behavior of the loop body:

$$\mathrm{LI}_i \equiv (\neg \mathrm{rsl}_i \wedge \mathrm{rcvl}_i \rightarrow y_i \geq x_i)$$
$$\wedge (\neg \mathrm{rsr}_i \wedge \mathrm{rcvr}_i \rightarrow z_i \geq x_i).$$

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

$$\mathrm{BL}_i \equiv (\mathrm{LI}_i \wedge \neg \mathrm{rsl}_i \wedge \neg \mathrm{rsr}_i).$$

Therefore, we have to prove the following property:

$$(*) \qquad (\mathrm{GI} \wedge \bigwedge_{i=1}^{n} \mathrm{BL}_i) \rightarrow (\bigwedge_{i=1}^{n} x_i = \gcd(\sigma_1, \cdots, \sigma_n)).$$

(*) 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 that

(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 that

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

Using (1) and (2), we get

$$x_i \leq z_i = x_{i+1} \qquad \text{and} \qquad x_{i+1} \leq y_{i+1} = x_i$$

which together imply that

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

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

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 first two 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

$$\begin{array}{l} \{rsl_i \wedge LI_i \wedge LI_{i-1} \wedge GI\} \\ P_{i-1}!x_i;\ \underbrace{rsl_i := \textbf{false};\ rcvl_i := \textbf{false}}_{A} \\ \qquad \| \\ P_i!z_{i-1};\ \underbrace{rcvr_{i-1} := \textbf{true};\ [\cdots]}_{B} \\ \{LI_i \wedge LI_{i-1} \wedge GI\}. \end{array}$$

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

By the rule of formation it remains to be proved that

$$\begin{array}{l} \{x_i = z_{i-1} \wedge rsl_i \wedge LI_i \wedge (\neg rsl_{i-1} \wedge rcvl_{i-1} \rightarrow y_{i-1} \geq x_{i-1}) \wedge GI\}, \\ A; B, \\ \{LI_i \wedge LI_{i-1} \wedge GI\} \end{array}$$

holds, where the above precondition is the postcondition 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 after executing $A; B$. All other changes need only routine checks.

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

4. DEADLOCK FREEDOM

Much as in [17, 18], we wish to use our proof system to show that a given program is deadlock free. For this purpose, however, our system as presented so far is incomplete, in contrast to [17, 18], and has to be strengthened. The resulting system can also be used to prove the absence of failure due to attempts at communication with processes that already terminated. (These questions do not arise in the work of Owicki and Gries because the distributed termination convention cannot be described in the programming languages which they consider.)

We adapt the concept of *blocking*, as introduced in [18]. This concept is used to characterize those states in which execution cannot be continued. Our version takes the distributed termination convention of CSP additionally into account, in that communication at the guards of *an I/O guarded repetition will not be blocked* in case all the processes referred to in the guards with a true Boolean component 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 processes are waiting for communication; there exists among them no pair of processes which wait for each other, one for input and the other for output; and 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 initial 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.

We proceed with the formal definitions required in order to formulate the theorem about deadlock freedom. We assume that a specific proof outline is given for each process P_i, $i = 1, \ldots, n$. Let I be the global invariant associated with the proof.

First we describe a *blocked situation.* A blocked situation is characterized by an n-tuple of sets of *communication capabilities* associated with the corresponding processes.

Assume that each process waits for a communication or has terminated. Then its communication capabilities are introduced as follows:

(i) If a process waits in front of an I/O command which is not a guard, then the bracketed section surrounding this I/O command constitutes its only communication capability.
(ii) If a process waits in front of an alternative or repetitive command, then a (possibly empty) subset of the set of all bracketed sections containing the I/O guards of that command form its set of communication capabilities. This subset corresponds to those guards whose Boolean parts evaluate to true.
(iii) If a process has terminated, then its communication capability consists only of acknowledging its termination.

Now, a situation is *blocked* if all of the following clauses hold:

(a) In the n-tuple of sets of communication capabilities there does not exist a matching pair of bracketed sections.
(b) If a process waits in front of a repetitive command, then its set of communication capabilities is nonempty, and not all processes (which are addressed in the bracketed sections) from its sets of communication capabilities acknowledge their termination.
(c) Not all processes acknowledge their termination.

To illustrate the concepts, just introduced, consider the following examples. In all of them we consider the situation in which each process waits to begin, so clause (c) applies trivially.

(1) Let $P :: [P_1 :: P_2!x \| P_2 :: P_1!y]$. Then clause (a) clearly holds, and (b) is obviously satisfied, so P is blocked.
(2) Let $P :: [P_1 :: P_2!x \| P_2 :: P_1?y]$. Then clause (a) does not apply, so the situation is not blocked.
(3) Let $P :: [P_1 :: *[P_2?x \rightarrow S] \| P_2 :: P_1?y]$. Then both (a) and (b) hold, so the situation is blocked.
(4) Let $P :: [P_1 :: *[P_2?x \rightarrow S] \| P_2 :: P_1!y]$. Then (b) holds but (a) does not, so the situation is not blocked.
(5) Let $P :: [P_1 :: *[\textbf{false}; P_2?x \rightarrow S] \| P_2 :: P_1!y]$. Then the set of communication capabilities of P_1 is empty because the Boolean guard of the loop is identically false. Thus (b) does not apply and the situation is not blocked. Indeed, P_1 can exit the loop, and then a blocked situation does indeed arise.
(6) Let $P :: [P_1 :: [\textbf{false}; P_2?x \rightarrow S] \| P_2 :: P_1!y]$. Then both (a) and (b) (notice that P_1 is a guarded selection!) are satisfied and the situation is blocked.

Next, we associate with each blocked situation an n-tuple of assertions. We intend to prove that program P is deadlock free (relative to assertion p) by checking that all blocked situations give rise to unsatisfiability of the global invariant I and all assertions associated with that situation.

In the subsequent discussion the following notation will be useful.

Let S be an alternative statement $[\Box\ (j = 1, \ldots, m)\ b_j; \alpha_j \rightarrow S_j]$ or a repetitive statement $*[\Box\ (j = 1, \ldots, m)\ b_j; \alpha_j \rightarrow S_j]$, and let $A \subseteq \{1, \ldots, m\}$. By $\text{pre}(S, A)$ we mean the assertion $\text{pre}(S) \wedge \bigwedge_{j \in A} b_j \wedge \bigwedge_{j \notin A} \neg b_j$.

Consider now a blocked situation. Let P_i be one of the blocked processes. We associate with P_i an assertion p_i:

(a) If P_i is in the situation as described in (i) above, then p_i is the preassertion of the corresponding bracketed section.
(b) If P_i is in the situation as described in (ii) above, then p_i is $\text{pre}(S, A)$, where S is the guarded command in front of which P_i waits and A is the set of indices corresponding with the set of communication capabilities of P_i.
(c) If P_i is in the situation as described in (iii) above, then p_i is $\text{post}(P_i)$.

We call an n-tuple $\langle p_1, \ldots, p_n \rangle$ of assertions associated with a blocked situation a *blocked* n-tuple.

Then the following theorem holds:

THEOREM 1. *Given a proof of* $\{p\}P\{q\}$ *with global invariant* I, P *is deadlock free (relative to* p*) if for every blocked n-tuple* $\langle p_1, \ldots, p_n\rangle$, $\neg(\bigwedge_{i=1}^{n} p_i \wedge I)$ *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, the proof of which is part of the proof of the soundness and completeness of the system, to be published by the first author.

THEOREM 2. *Let a proof of* $\{p\}P\{q\}$ *be given. If during execution of* P *starting in a state satisfying* p, *each* P_i *is about to execute a statement with a preassertion* pre_i, *then* $\bigwedge_{i=1}^{n} pre_i$ *is satisfied by the (global) state at that moment. If* P_i *has terminated, then* $post(P_i)$ *holds. If none of the processes is within a bracketed section, then* I *holds.*

To illustrate the use of Theorem 1, we now prove deadlock freedom of the programs considered in Examples 1, 3, and 4 of Section 2.

To deal with the program from Example 1, $[P_1 :: P_2!x \,\|\, P_2 :: P_1?y;\ P_3!y \,\|\, P_3 :: P_2?u]$, we need the following new proof outlines:

$$\{i = 0\}\ \langle P_2!x;\ i := 1\rangle\ \{i = 1\},$$
$$\{j = 0 \wedge k = 0\}\ \langle P_1?y;\ j := 1\rangle;\ \{j = 1 \wedge k = 0\}$$
$$\langle P_3!y;\ k := 1\rangle\ \{j = 1 \wedge k = 1\},$$
$$\{l = 0\}\ \langle P_2?u;\ l := 1\rangle\ \{l = 1\}.$$

Let $I \equiv i = j \wedge k = l$.

The proofs clearly cooperate and can be used to establish the rather unimpressive fact that $\{\mathbf{true}\}[P_1 \,\|\, P_2 \,\|\, P_3]\ \{\mathbf{true}\}$ holds. On the other hand the above proof outlines are sufficient for the proof of deadlock freedom. It is easy to see that the conjunction of any blocked triple of assertions implies $i \neq j \vee k \neq l$, which is incompatible with I. By Theorem 1, $[P_1 \,\|\, P_2 \,\|\, P_3]$ is deadlock free relative to **true**.

Having dealt with I/O commands only, let us now consider a program containing an I/O guarded alternative statement, namely, the program from Examples 3 and 4, $[P_1 :: [P_2?x \to \text{skip} \,\square\, P_2!0 \to P_2?x;\ x := x + 1] \,\|\, P_2 :: [P_1!2 \to \text{skip} \,\square\, P_1?z \to P_1!1]]$. In this case the proof outlines given in Example 4 are sufficient to show deadlock freedom relative to **true**. The analysis is simplified by the fact that the Boolean guards of the alternative statements are identical to **true**; this implies that any process waiting to start has exactly two communication capabilities.

In particular, the situation when one process waits to start and the other did not terminate is not blocked. The only situation which is blocked is when one process waits to start and the other has terminated. The corresponding pair of blocked assertions then implies $i \neq j$, which is incompatible with the global invariant $I \equiv i = j$.

Let us now turn our attention to programs containing an I/O guarded repetitive command. One of the simplest examples is a program of the form [skip ‖ *[$\alpha \rightarrow$ skip]]. This program is clearly deadlock free relative to **true**, and the proof of this fact is trivial—according to the definitions there is simply no blocked situation, so no blocked pair of assertions needs to be considered.

We are less fortunate when trying to prove deadlock freedom of the program [$\bar{\alpha}$ ‖ *[$\alpha \rightarrow$ skip]]. In spite of our elaborated definitions it is impossible to prove with our method that the trivial program above is deadlock free relative to **true**! The easiest way to see this is as follows:

(1) The only formally blocked situation is the one when the first process waits to start and the second has terminated. Of course such a situation cannot occur operationally, but our definitions above do not rule this situation out.

(2) Consider now a new, fictitious interpretation of I/O guarded repetitive commands according to which the loop can also be exited *immediately*. Our rule for I/O guarded repetition is still sound under this interpretation, and the description of blocked situations still applies to the new interpretation. As a result, both Theorem 1 and 2 remain valid. If we were now able to prove the required premise of Theorem 1 in the case of the above program, then this program would be deadlock free relative to **true** under the new interpretation. But the latter is clearly not the case, since the new interpretation now makes the only formerly blocked situation reachable.

Note that the reasoning above does not contradict the relative completeness of the introduced proof system for partial correctness. Namely, if $\{p\}P\{q\}$ is true under the usual interpretation, then it is true under the new interpretation, so the argument above does not apply any more.

One is tempted to consider the situation above where the first process waits to start and the other has terminated as not being blocked. However, such a solution does not work with more complicated programs, for instance, when P_2 is of the form *[**false** $\rightarrow$ *[$\alpha \rightarrow$ skip]].

We conclude that the present system is inadequate for reasoning about deadlock freedom, since its underlying interpretation can be changed so as to rule out the example of formal blocking considered above, while keeping axioms and proof rules satisfied.

To remedy the situation, we introduce local propositional variables End_j^i, $i \neq j$, $1 \leq i, j \leq n$, with the following interpretation: End_j^i holds if P_i "assumes" that P_j has terminated. 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 due only to a loop exit in some process. In the proof (but not in the program) this change of value is described as if assignments take place upon loop exit. End_j^i can *only* be used in proofs concerning P_i.

The new rule for I/O guarded repetition now becomes

R2′. *Guarded Repetition*

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

Here k_j denotes 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_j^i are used in general in the global invariant I, so setting them to **true** can affect the invariant. Therefore we must add the following clause to the definition of cooperation:

(iii) Let S denote a subprogram of P_i of the form

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

Let $A \subseteq \{1, \ldots, m\}$, and let C be the set of indices of all processes referred to in α_j for $j \in A$. Then $\bigwedge_{j \in C} \mathrm{post}(P_j) \wedge \mathrm{pre}(S, A) \wedge I \rightarrow (\mathrm{post}(S) \wedge I)\ [\mathbf{true}/\mathrm{End}_j^i]_{j \in C}$ holds.

Here $q\ [\mathbf{true}/\mathrm{End}_j^i]_{j \in C}$ stands for the formula obtained from q by simultaneous substitution of **true** for End_j^i, $j \in C$.

Clause (iii) states that if process P_i is about to exit an I/O guarded repetition (which is expressed by the left-hand side of the formula), then the exit itself (modeled by setting the corresponding End_j^i variables to **true**) both preserves the invariant and establishes the postcondition of the loop. The other assertions do not use End_j^i variables and so cannot be affected by the exit.

The adopted changes retain the validity of Theorem 1.

A simple example serves to illustrate the concepts introduced. Consider the program $P :: [\bar{\alpha} \| *[\alpha \rightarrow \mathrm{skip}]]$ (which caused our troubles originally) with the following proof outlines:

$$\{i = 0\}\ \langle \bar{\alpha};\ i := 1 \rangle\ \{i = 1\},$$
$$\{\neg \mathrm{End}_1^2\}\ *[\alpha \rightarrow \mathrm{skip}]\{\mathrm{End}_1^2\},$$

and let $I \equiv \mathrm{End}_1^2 \rightarrow i = 1$.

All omitted assertions are equal to **true**. The second proof outline makes use of rule R2′. The proofs cooperate—the new clause of cooperation,

$$i = 1 \wedge \neg \mathrm{End}_1^2 \wedge I \rightarrow (\mathrm{End}_1^2 \wedge I)[\mathbf{true}/\mathrm{End}_1^2],$$

clearly holds.

The only blocked situation leads to a blocked pair $\langle i = 0, \mathrm{End}_1^2 \rangle$ of assertions which are clearly incompatible with I. The proof outlines are sufficient to establish the proof of $\{\mathbf{true}\}P\{\mathbf{true}\}$. By Theorem 1, P is deadlock free relative to **true**.

Now we apply these new 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

$$\mathrm{GI}' \equiv \mathrm{End}_1^2 \rightarrow mx \leq x,$$

and add $mx > x$ to the precondition of the two bracketed sections in the loop of P_1, as well as adding $mx \leq x$ to the postcondition of P_1. Also, the use of the strong version of the I/O guarded repetition rule implies that End_1^2 is added to $\mathrm{post}(P_2)$. In showing the cooperation of proofs, the only new case that has to be checked is the loop exit of P_2, since we can assume that $\mathrm{post}(P_1)$, GI′ holds.

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

In all cases which do not involve the postassertions 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 $l_1 = l_2$. For example, with p as the preassertion of the first bracketed section of P_1 and q as the preassertion of the first bracketed section of P_2 inside its loop, we have

$$l_1 = 0 \wedge \text{odd}(l_2) \wedge l_1 = l_2,$$

which is contradictory.

The only other case with an essentially different proof, which does not use the fact that GI implies $l_1 = l_2$, is when p denotes the preassertion of P_1's first bracketed section inside its loop and P_2 has terminated, i.e., q contains End_1^2 (among others). Then we have

$$mx > x \wedge (\text{End}_1^2 \rightarrow mx \leq x) \wedge \text{End}_1^2,$$

which again is contradictory.

Note that it is only here that the additional invariant GI′ is used.

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

Let $\text{GI}' \equiv \bigwedge_{i=1}^{n} (\text{End}_{i+1}^{i} \equiv \text{End}_{i}^{i+1})$. We shall prove the invariance of GI′. By using the strong repetition rule R_2', we get that each post(P_i) implies

$$\text{End}_{i+1}^{i} \wedge \text{End}_{i-1}^{i}$$

(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

$$\text{GI}' \wedge \text{End}_i^{i+1} \wedge \text{End}_i^{i-1} \rightarrow (\text{GI}' \wedge \text{End}_{i+1}^{i} \wedge \text{End}_{i-1}^{i})[\textbf{true}/\text{End}_{i+1}^{i}, \textbf{true}/\text{End}_{i-1}^{i}],$$

which trivially holds.

A simple consequence of GI′ is

$$(**) \qquad \bigwedge_{i \neq j} \text{End}_j^i \equiv \text{End}_i^j.$$

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 [6] to show that the program does not terminate, since its termination dependency graph is cyclic,
(iii) the absence of blocked tuples of assertions other than the one considered in Section 3, as was shown above.

The proof of (i) is beyond the scope of the present paper and therefore 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.

As we see no way of improving in this respect upon Leslie Lamport's lucid comments upon our paper we feel justified in citing him *in extenso:*

> This paper provides a method for proving safety properties (the generalization of partial correctness properties) of programs written in CSP. Proving such properties requires proving that if the program is started in a valid initial state, then a certain assertion will always remain true. This in turn is proved by showing that some assertion I is invariant—i.e., if the program is started in any state in which I is true, then I remains true.
>
> The simplest approach to proving the invariance of I is to show that each atomic action of the program leaves I true. This approach was first described by Ashcroft [3]. The next approach, taken by Owicki and Lamport, takes into account the structure of ordinary multiprocess programs, in which each atomic action occurs as the result of executing one "program step" in some process. The invariant assertion I is written as the conjunction of assertions of the form "control at $x \rightarrow I(x)$," where $I(x)$ is the assertion "attached to" control point x. To prove invariance of I, one proves the following for each control point x.
>
> If $I(x)$ is true, control is at x, and executing the program step at x leaves control at x', then
>
> (1) $I(x')$ is true after execution;
> (2) for each control point y in every other process, if $I(y)$ is true before the execution and control is at y, then $I(y)$ is true after the execution.
>
> The second part of the conclusion was called "interference freedom" by Owicki. This method can be viewed as a special case of Ashcroft's method, in which the assertion I has a special form. Conversely, Ashcroft's method can be viewed as the special case of Owicki's and Lamport's in which the single assertion I is attached to all control points. (This illustrates the futility of trying to decide whether one method is more general than another.)
>
> Because the same assertion is attached to each location, part 2 (interference freedom) of the conclusion is implied by part 1, so no explicit proofs of interference freedom are needed by Ashcroft's method. However, this provides no real advantage since the same amount of verification is required in both methods: the interference freedom proofs appear in Ashcroft's method as the extra complexity of proving that the larger monolithic assertion I is left true by each atomic operation. The difference in the two methods is largely a matter of syntactic convenience. The interference freedom method is more convenient when the global invariant assertion I is conveniently written as the conjunction of assertions $I(x)$ attached to program control points. Ashcroft's method is more convenient when the invariant I is simple and does not need to be decomposed.
>
> In Owicki's treatment, the assertions $I(x)$ could not explicitly mention program control points. This meant that she had to introduce auxiliary variables, instead.

Now suppose we consider a more general multiprocess programming language, in which program steps in one or more different processes may be executed simultaneously as one single step. Let us call $\{x1, \ldots, xi\}$ a multicontrol point if the program steps at control points $x1, \ldots, xi$ are steps which may be executed simultaneously in this way—where each of the x's is in a different process. (The singleton $\{x\}$ is a multicontrol point if the program step at x is a local one, which can be executed by itself.) If $x = \{x1, \ldots, xi\}$, define $I(x)$ to be the conjunction of the assertions $I(x1), \ldots, I(xi)$. The above Owicki/Lamport proof rule can then be generalized by replacing the single control points x and x' by multicontrol points, where "control at x" is defined in the obvious way for a multicontrol point x. (In the new definition, y remains an ordinary [single] control point.) [This methodology was independently developed by Mazurkiewicz [15] where simultaneous *await*-statements are considered.]

The approach obviously provides a proof methodology for CSP, where the nonlocal multicontrol points involve I/O statements. [The actual transition from proof methodology to proof system is achieved by providing suitable axioms and proof rules, such as the communication axiom, which enable incorporation of the above generalization of condition 1 (i.e., cooperation) into the proof system.] The proof method presented in the present paper can be derived as follows, as a special case of this general method, on the basis of the fact that syntactic restrictions on the type of assertions that can be used make certain verifications unnecessary. First of all, the CSP language is generalized by introducing the "bracketed sections." The bracketing defines the nonlocal atomic operations. The rules for what may appear inside brackets are codifications of the well-known fact that operations that affect only local variables may be subsumed within an adjacent atomic operation. (In particular, it does not make any difference how the local atomic operations are defined.)

The nonlocal multicontrol points are the control points at the beginning of the bracketed statements. The assertion $I(y)$ attached to each control point y is of the form "$I'(y)$ and I," where $I'(y)$ is the assertion explicitly attached to y, and I is the "global invariant." The separation of the proof into a local proof and a proof of "cooperation" involves the separation into local control points (singleton multicontrol points) and nonlocal control points. Rules A1 and A2 simply enforce that the statements involving I/O concern nonlocal control points, and are not considered by the local proof.

The fact that no interference freedom proofs are necessary is an immediate consequence of the restriction that the assertion attached to each control point y is of the form "$I'(y)$ and I," where $I'(y)$ contains variables only modified by that process. [The same remark applies to the proof system considered in [17].] No interference proofs are needed for precisely the same reason that they are not needed in Ashcroft's method: because the only nonlocal assertion is attached to all control points. The global assertion I does not have to appear in the local part of the proof because of the assumption that it contains no variables that can be set by other local operations.

In the present paper program control is modeled by the use of auxiliary variables and the global invariant. A different approach (suggested by L. Lamport) can be envisaged here, in which program control variables are explicitly allowed to appear in assertions making the use of the global (monolithic) invariant unneeded.

A full discussion of the relative merits of these two alternative approaches, i.e., auxiliary variables versus program control variables, is beyond the scope of the paper. We mention only that program control variables lead in general to nonrecursive intermediate assertions (see [1]).

It is also possible to have a proof system for CSP without global invariants, in which only shared auxiliary variables are used. An example is the proof system

presented in [14], where the component proofs have to be checked both for interference freedom and cooperation, since auxiliary variables can be shared.

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 [18], where new proofs are needed to show interference freedom. These proofs can be viewed as global reasoning since they involve more than one process. In our case the bigger the bracketed sections, the more sizable the proofs that have to be carried out. The forthcoming proof of relative completeness of our system implies that we can always choose bracketed sections of the form α; 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 [18]; 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 unmatched pairs. For example, in our proof for the partitioning program, eight cases had to be established in the cooperation test and fifteen for the proof of absence of deadlock, but only four cases have a nontrivial proof of the cooperation test, and only one such case occurs in the proof of absence of deadlock.

Finally, the results of this paper are summarized.

We have presented a system both for understanding and for proving correctness of CSP programs. The main feature of this system is the notion of cooperating proof outlines. The arguments leading to the system as a whole have been motivated within the context of CSP. However Lamport's remarks seem to indicate that the notion of cooperating proof outlines is also essential for proving correctness of concurrent programs written in an extension of the usual shared variable framework with mutual synchronization (by means of "multicontrol points").

CSP expresses distributed termination of processes. We illustrate this aspect in our system by proofs of two examples of distributed computation, one for partitioning a finite set, the other for computing the gcd of n numbers concurrently.

In order to prove absence of deadlock and failure (i.e., abortion), the proof system has to be strengthened. This is a consequence of CSP's distributed termination convention. The final system is obtained by adding the proof theoretical counterpart of this termination convention.

ACKNOWLEDGMENTS

We express our gratitude to Leslie Lamport for his lucid comments, on which a substantial part of Section 5 is based. We also thank David Luckham, Susan Owicki, and Gordon Plotkin for their remarks. Finally, we feel both personally and scientifically indebted to Edsger Dijkstra and Tony Hoare.

REFERENCES

1. APT, K.R. Recursive assertions and parallel programs. Submitted for publication.
2. APT, K.R., DE ROEVER, W.P., AND FRANCEZ, N. Weakest precondition semantics for communicating processes. To appear.
3. ASHCROFT, E. Proving assertions about parallel programs. *J. Comput. Syst. Sci. 10* (1975), 110–135.
4. CHANDY, K.M., AND MISRA, J. An axiomatic proof technique for networks of communicating processes. Tech. Rep. TR-98, Dep. of Computer Science, Univ. of Texas at Austin, 1979.
5. DIJKSTRA, E.W. A correctness proof for communicating processes—A small exercise. EWD-607, Burroughs, Nuenen, The Netherlands, 1977.
6. FRANCEZ, N. On achieving distributed termination. *ACM Trans. Program. Lang-Syst. 2*, 1 (January 1980), 42–55.
7. FRANCEZ, N., HOARE, C.A.R., LEHMANN, D.J., AND DE ROEVER, W.P. Semantics of nondeterminism, concurrency and communication. *J. Comput. Syst. Sci. 19* (1979), 290–308.
8. FRANCEZ, N., AND PNUELI, A. A proof method for cyclic programs. *Acta Inf. 9* (1978).
9. FRANCEZ, N., AND RODEH, M. Achieving distributed termination without freezing. Rep. TR 72, IBM Israel Scientific Center, 1980.
10. HOARE, C.A.R. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R. Perrot, Eds., Academic Press, New York, 1972.
11. HOARE, C.A.R. Communicating sequential processes. *Commun, ACM 21*, 8 (August 1978), 666–677.
12. HOWARD, J.H. Proving monitors. *Commun. ACM 19*, 5 (May 1976), 273–279.
13. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. 3*, 2 (1977), 125–143.
14. LEVIN, G.M. A proof technique for communicating sequential processes (with an example). Tech. Rep., Computer Science Dep., Cornell Univ., 1979. To be submitted to *Acta Inf.*
15. MAZURKIEWICZ, A. A complete set of assertions on distributed systems. Inst. of Computer Science, Polish Academy of Science, 1979.
16. OWICKI, S.S. A consistent and complete deductive system for the verification of parallel programs. Proc. 8th ACM Symp. on Theory of Computing, 1976, 73–86.
17. OWICKI, S.S., AND GRIES, D. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM 19*, 5 (May 1976), 279–285.
18. OWICKI, S.S., AND GRIES, D. An axiomatic proof technique for parallel programs. *I. Acta Inf. 6*, 1976, 319–340.

Received August 1979; revised May 1980; accepted May 1980.