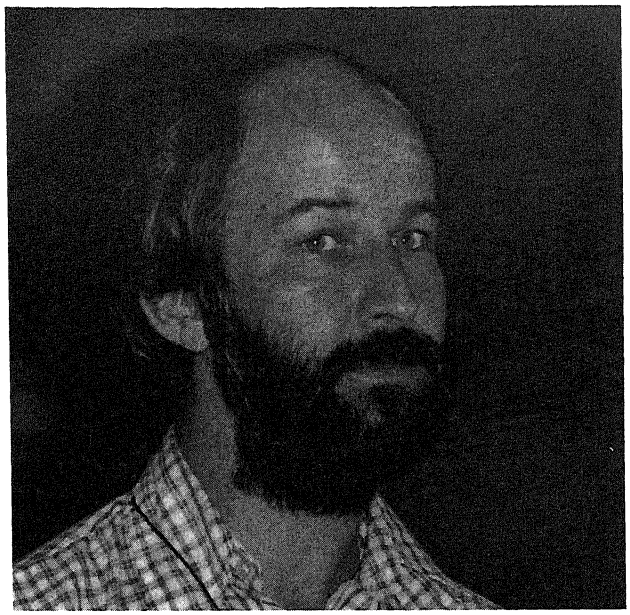

K. Apt

E.W. Dijkstra

PROVING CORRECTNESS OF CSP PROGRAMS - A TUTORIAL

Krzysztof R. Apt
L.I.T.P., Université Paris 7
2, Place Jussieu, 75251 Paris, FRANCE

Abstract

A structured presentation of a proof system for CSP programs is given. The presentation is based on the approach of Apt, Francez and de Roever [AFR]. Its new aspects are the use of static analysis and of proofs from assumptions instead of proof outlines. Also, in contrast to [AFR] total correctness is studied.

1. INTRODUCTION

In 1978 C.A.R. Hoare introduced in [H4] a language for distributed programming called *C*ommunicating *S*equential *P*rocesses - in short CSP. Subsequently this language has received a great deal of attention. In particular various proof methods for its programs have been proposed ([AFR], [CC], [LG], [LS], [MC], [MP], [S]).

The aim of this paper is to provide a structured exposition of a proof system for CSP programs based on the approach of Apt, Francez and De Roever [AFR]. We attempt to show that this system is a logical - admittedly multistep - extension of a proof system for disjoint parallel programs first studied by Hoare in [H3]. The new aspects of our approach with respect to that of [AFR] are the use of static analysis and of proofs from assumptions instead of proof outlines.

Static analysis allows to reduce substantially the number of cases which have to be considered at the level of interaction between the proofs of the component programs. It was suggested in the framework of program proving in [A4] but was originally proposed earlier and independently by R.N. Taylor in [T] as a tool for study of ADA tasks.

In our approach proofs from assumptions replace proof outlines originally introduced by Owicki and Gries in [OG1] and used in [AFR]. Proofs from assumptions were introduced in the context of program proving by Hoare in [H2] to handle the case of recursive procedures. Here their use is even easier to understand - assumptions are simply made about the appropriate subparts of the programs which do not have a meaning when viewed in isolation - the i/o commands. The use of proofs from assumptions instead of proof outlines makes the presentation simpler and more transparent.

NATO ASI Series, Vol. F14
Control Flow and Data Flow: Concepts of Distributed Programming. Edited by M. Broy

Finally, in contrast to [AFR] total correctness of CSP programs is studied. This also covers proofs of deadlock freedom already studied in [AFR] but now considerably simplified thanks to the use of static analysis. Other aspects of total correctness - proofs of absence of failures and of termination require some care in the presence of i/o guards and do not seem to have received sufficient attention in the literature.

2. AN OVERVIEW AND PRELIMINARIES

2.1 An overview

At the risk of not being fully understood during the first reading we provide now a bird's eye view of our approach. Reader may find it useful to return here while reading successive sections of the paper.

The basic vocabulary like partial and total correctness is introduced in the next, very brief subsection. In contrast, various programming concepts like i/o commands, processes, etc... are introduced only gradually. The exposition starts with a presentation of the proofs systems for partial and total correctness of nondeterministic programs of Dijkstra [D1, D2] in section 3. The proof systems of course are not new. A first incursion into the realm of parallelism takes place in section 4. Here only the most trivial type of parallelism is allowed - a parallel composition of nondeterministic programs whose variables do not conflict with each other. The corresponding proof rule for parallel composition is straightforward. A minor novelty of this section is that already at this stage the concept of auxiliary variables is introduced and it is indicated why they are needed.

The next step consists of allowing communication between the components of a parallel program. In section 5 a minimal extension of the language of section 4 is studied - i/o commands are allowed here to be used but only as atomic actions. What changes should now be made with respect to the proof system studied in section 4 ? Consider a component of a parallel program (called a process). If we knew the meaning of its i/o commands then we could prove its correctness. In other words we can prove correctness of each process *relative* to appropriate assumptions about its i/o commands. The next step consist of discharging these assumptions. It is done as follows. First, using the static analysis all pairs of i/o commands which can be synchronized are identified. Next, for each such pair it is checked that the assumptions about them are mutually consistent. If it is the case then we can discharge all the assumptions and prove correctness of the parallel program in question.

This approach unfortunately fails when applied to more complicated programs and has to be modified accordingly. The refinement makes use of *global invariants*, i.e. properties which hold throughout the executions of the parallel program and of *bracketed sections*, i.e. sections which contain i/o commands and within which the invariant actually does not need to hold. In the modified version of the proof system one first proves correctness of each process relative to assumptions about its bracketed sections and next discharges them by checking their mutual consistency similarly as before. Moreover, it is checked that the global invariant is preserved. It is done by a combination of syntactic and semantic means. First, in order to "localize"

the possible changes of the invariant it is required that its free variables cannot be changed outside of the bracketed sections. Secondly, it is proved (during the verification of the assumptions about the bracketed sections) that a joint execution of every pair of synchronized bracketed section preserves the invariant. All these checks allow to conclude that the parallel program under consideration is correct with respect to the originally chosen assertions and moreover preserves the global invariant. Global invariants are helpful because they allow to relate variables from different processes. The presentation of the system is completed by the introduction of auxiliary variables.

The whole approach can be readily used to prove deadlock freedom of the parallel programs considered. First, using the static analysis one identifies all situations in which deadlock could arise. With each such situation one can associate an appropriate assertion which will hold at the moment this situation is reached. This assertion can be constructed once the correctness proof of the parallel program is given. To prove deadlock freedom it now suffices to check that for every deadlock situation the appropriate assertion is inconsistent.

In section 6 this approach is generalized to handle the case of programs allowing i/o commands as guards. The necessary modifications are mostly routine and hardly necessitate additional comments.

2.2 Preliminaries

Throughout the paper we fix an arbitrary first order language with equality containing two boolean constants true and false. Its formulae are called *assertions* and denoted by the letters p,q,r. Simple variables are denoted by the letters u,x,y,z and expressions by the letters s,t. Quantifier-free formulae are called *boolean expressions* and are denoted by the letter b. p[t/u] stands for a *substitution* of t for all free occurrences of u in p. By free(p), free(t) we denote the set of all variables which occur free in the formula p and expression t, respectively.

By a *correctness formula* we mean a construct of the form {p} S {q} where p,q are assertions and S is a program. The classes of programs considered will be defined in the subsequent sections.

Two types of interpretation of correctness formulae {p} S {q} will be considered in this paper. We say that {p} S {q} is true (or *holds*) in the sense of *partial correctness* if all properly terminating computations of S starting in a state satisfying p terminate in a state satisfying q. We say that {p} S {q} is true (or *holds*) in the sense of *total correctness* if it holds in the sense of partial correctness and moreover all computations of S starting in a state satisfying p properly terminate.

We assume that variables are of type integer or boolean. Thus programs are executed over a domain consisting of all integers and {true, false} with the usual operations available.

3. NONDETERMINISTIC PROGRAMS

We start our analysis by concentrating on nondeterministic programs à la Dijkstra [D1, D2]. We allow as atomic actions the skip statement and assignment u:=t. Programs are built up using the composition operation ";" and allowing the *alternative command*

$[\ \square_{i=1}^{m}\ b_i \to S_i\ ]$ and *repetitive command* $*[\ \square_{i=1}^{m}\ b_i \to S_i\ ]$ where

b_i are boolean expressions (called *guards*) and S_i are programs.

3.1 Partial correctness

The following proof system allows to prove partial correctness of the nondeterministic programs introduced above :

AXIOM 1 : SKIP AXIOM

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

AXIOM 2 : ASSIGNMENT AXIOM

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

RULE 3 : COMPOSITION RULE

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

RULE 4 : ALTERNATIVE COMMAND RULE

$$\frac{\{p \wedge b_i\}\ S_i\ \{q\},\ i=1,\ldots,m}{\{p\}\ [\ \square_{i=1}^{m}\ b_i \to S_i\ ]\ \{q\}}$$

RULE 5 : REPETITIVE COMMAND RULE

$$\frac{\{p \wedge b_i\}\ S_i\ \{p\},\ i=1,\ldots,m}{\{p\}\ *[\ \square_{i=1}^{m}\ b_i \to S_i\ ]\ \{p \wedge \bigwedge_{i=1}^{m} \neg b_i\}}$$

RULE 6 : CONSEQUENCE RULE

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

We call this proof system N. It is an appropriate modification of the original proof system for while-programs proposed by Hoare in [H1].

3.2 Total correctness

Nondeterministic programs can fail to terminate properly because of *failures*. According to [D2] a failure arises if at the moment of starting an execution of an alternative command all its guards evaluate to false. Obviously the rule of alternative command does not exclude such a possibility. The following modification of this rule ensures the desired property.

RULE 7 : ALTERNATIVE COMMAND RULE II

$$\frac{p \to \bigvee_{i=1}^{m} b_i,\ \{p \wedge b_i\}\ S_i\ \{q\},\ i=1,\ldots,m}{\{p\}\ [\ \Box_{i=1}^{m}\ b_i \to S_i\ ]\ \{q\}}$$

The first premise guarantees that at the moment an alternative command is to be executed at least one of its guards evaluates to true.

Next, we have to take care of termination of repetitive commands. The current version of the rule of repetitive commands is clearly insufficient for this purpose. We follow here the approach of [APS] and modify rule 5 as follows.

RULE 8 : REPETITIVE COMMAND RULE II

$$\frac{\{p(\bar{n}) \wedge b_i\}\ S_i\ \{\exists\ \bar{m} <\bar{n}\ p(\bar{m})\},\ i=1,\ldots,m}{\{\exists\ \bar{n}\ p(\bar{n})\}\ *\ [\ \Box_{i=1}^{m}\ b_i \to S_i\ ]\ \{\exists\ \bar{n}\ p(\bar{n}) \wedge \bigwedge_{i=1}^{m} \neg b_i\}}$$

Here $p(\bar{n})$ is an assertion with a free variable $\bar{n}$ which does not appear in $*\ [\ \Box_{i=1}^{m}\ b_i \to S_i\ ]$. Both $\bar{n}$ and $\bar{m}$ range over natural numbers. We call $\bar{n}$ a *parameter variable*.

We denote the resulting system NT. We refrain here from a further discussion of the issue of correctness of nondeterministic programs. A theoretical analysis of the above two proof systems can be found in [A3].

4. DISJOINT PARALLELISM

As a next step towards the analysis of CSP programs we introduce *disjoint parallelism*, a concept first studied by Hoare in [H2]. Given a program S we denote by free(S) the set of all variables which occurs in it and by change(S) the set of all its variables which are subject to change, i.e. which appear on the left hand side of an assignment. We call the variables from the set free(S) - change(S) *"read only"* variables of S.

We now introduce disjoint parallelism by allowing the construct :

$$[S_1 \| \ldots \| S_n]$$

standing for a parallel composition of the programs $S_1,\ldots,S_n$, under the condition that :

$$free(S_i) \cap change(S_j) = \emptyset \text{ for } i \neq j.$$

Thus some of the programs $S_1,\ldots,S_n$ can have a variable in common but if this is the case then none of S_i can change its value. In other words the programs $S_1,\ldots,S_n$ can share only "read only" variables. So for example $[x:=z \| y:=z]$ is allowed but $[x:=z \| y:=x]$ or $[x:=z \| x:=z]$ not.

Programs of the above type are called *disjoint parallel programs*. $S_1,\ldots,S_n$ are activated in parallel and the execution of $[S_1 \| \ldots \| S_n]$ terminates when all S_i have terminated.

4.1. Proof rules

Hoare suggested in [H2] the following proof rule for disjoint parallel programs.

RULE 9 : RULE OF DISJOINT PARALLEL COMPOSITION

$$\frac{\{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n}{\{\bigwedge_{i=1}^{n} p_i\}\ [S_1 \| \ldots \| S_n]\ \{\bigwedge_{i=1}^{n} q_i\}}$$

provided $free(p_i,q_i) \cap change(S_j) = \emptyset$ for $i \neq j$.

Observe the natural connection between the condition imposed on the programs S_i and the condition of the rule. We can say informally that the proof rule follows the syntax of the programs.

Obviously the condition imposed on the free variables of p_i and q_i is necessary. Consider for example the correctness formulae $\{y = 0\}\ x:=0\ \{y = 0\}$ and $\{\underline{true}\}\ y:=1\ \{\underline{true}\}$. They both can be proved but $\{y = 0\}\ [x:=0 \| y:=1]\ \{y = 0\}$ obviously does not hold.

The above rule does not suffice to prove all properties of disjoint parallel programs. It is quite easy to prove that the correctness formula $\{x=y\}\ [x:=x+1 \| y:=y+1]\ \{x=y\}$ cannot be proved using rule 9 (and possibly the rule of consequence) only.

Therefore we supplement the proof system by the following substitution rule being a special case of a proof rule first introduced by Gorelick in [G] (see also section 3 of [Al].

RULE 10 : SUBSTITUTION RULE

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

This rule allows to initialize variables not appearing in the program or a post-assertion to an arbitrary value.

Example 1

We now illustrate the use of the substitution rule by proving the previously introduced formula :

$\{x = y\}\ [x:=x+1 \parallel y:=y+1]\ \{x = y\}$.

We obviously have :

$\{x = z\}\ x:=x+1\ \{x = z+1\}$

and

$\{y = z\}\ y:=y+1\ \{y = z+1\}$.

We can now apply the rule of disjoint parallel composition. We obtain

$\{x = z \wedge y = z\}\ [x:=x+1 \parallel y:=y+1]\ \{x = z+1 \wedge y = z+1\}$

so by the consequence rule

$\{x = z \wedge y = z\}\ [x:=x+1 \parallel y:=y+1]\ \{x = y\}$

The variable z does not appear in the program or in the post-assertion so we can apply the substitution [y/z] we obtain the desired result. □

4.2.Auxiliary variables

Note that the last step in the above proof could have been alternatively obtained as follows.

We have

$\{x = y\}\ z:=y\ \{x = z \wedge y = z\}$

so by the composition rule

$\{x = y\}\ z:=y\ ;\ [x:=x+1 \parallel y:=y+1]\ \{x = y\}$.

We can now obtain the desired formula by dropping the assignment z:=y. Formally this step requires use of a new rule allowing to delete assignments to the so-called auxiliary variables.

Definition 1 Let A be a set of variables of a program S. A is called the set of *auxiliary variables* of S if

i) all variables of A appear in S only in assignments,

ii) no variable of S from outside of A depends on the variables

from A. In other words there does not exist an assignment x:=t within S such that $x \notin A$ and $free(t) \cap A \neq \emptyset$.

Thus for example {z} is a set of auxiliary variables of the program z:=y ; [x:=x+1 || y:=y+1] but {y} is not as z depends on y.

The following proof rule was first introduced by Owicki and Gries in [OG1, OG2].

RULE 11 : RULE OF AUXILIARY VARIABLES

Let A be a set of auxiliary variables of a program S. Let S' be obtained from S by deleting all assignments to the variables in A. Then

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

provided $free(q) \cap A = \emptyset$.

It is useful to note that the substitution rule can be derived using the above rule. Indeed, suppose that {p} S {q} holds and that $z \notin free(S,q)$. We have {p[t/z]} z:=t {p} so by the composition rule {p[t/z]} z:=t ; S {q}. We now obtain {p[t/z]} S {q} applying the rule of auxiliary variables.

5. I/O COMMANDS AS ATOMIC ACTIONS

We now extend the syntax by introducing commands allowing communication between the components of a disjoint parallel program. First we introduce names for each of the component ; $P_i :: S_i$ denotes that P_i is the name of the component S_i. Components will be called from now on *processes*.

Processes can address each other using i/o commands being either the *input command* $P_j?x$ or an *output command* $P_j!t$. Input command $P_j?x$ within the process P_i expresses a request to P_j to assign a value to the variable x of P_i. Output command $P_i!t$ within the process P_j expresses a request to P_i to receive from P_j the value of t. Execution of $P_j?x$ within the process P_i and of $P_i!t$ within the process P_j is synchronized and results in assigning the value of t to x. Thus the joint effect of the execution of $P_j?t$ and $P_i!t$ is that of the assignment x:=t.

When studying parallel composition of processes we have to modify the definition of the set change(S) by putting into it also all variables appearing in the input commands of S.

We now allow parallel composition of processes

$$[P_1 :: S_1 \| \ldots \| P_n :: S_n]$$

under the same condition as before i.e. that

$$free(S_i) \cap change(S_j) = \emptyset \quad \text{for} \quad i \neq j.$$

Thus similarly as before only "read only" variables can be shared among the processes.

The programs introduced above are special cases of CSP programs.

5.1 Static analysis

For reasons that will become clear in the next subsections we shall now try to identify all pairs of i/o commands which can be synchronized during executions of a parallel program

$$P \equiv [P_1 :: S_1 \| \ldots \| P_n :: S_n].$$

We follow here the approach of [A4]. Given a pair α,β of i/o commands we say that they *match*, when one of them is an input command, the other an output command and they address each other, i.e. for some i,j $i \neq j$, α is from P_i, β is from P_j, α adresses P_i and β addresses P_j. If a pair of i/o commands can be synchronized then it necessarily matches. The converse is obviously not true. We now propose another, more restrictive necessary condition for a pair of i/o commands to be able to synchronize.

Given a parallel program $P \equiv [P_1 :: S_1 \| \cdots \| P_n :: S_n]$ we first label all its i/o commands in order to distinguish different occurrences of the same command.

We proceed in two stages.

1) With each process P_i we associate a regular language $L(P_i)$ defined by structural induction on S_i. We put

$$L(\underline{skip}) = L(x:=t) = \{\epsilon\},$$
$$L(k:P_j?x) = \{k:\langle j,i\rangle\},$$
$$L(k:P_j!t) = \{k:\langle i,j\rangle\},$$
$$L(S_1;S_2) = L(S_1)L(S_2),$$
$$L([\ \Box_{i=1}^{m} b_i \rightarrow S_i]) = \bigcup_{i=1}^{m} L(b_i \rightarrow S_i),$$
$$L(b_i \rightarrow S_i) = L(b_i)L(S_i),$$
$$L(b_i) = \{\epsilon\},$$
$$L(*\ S) = L(S)^*.$$

Intuitively, $L(P_i)$ is the set of all a priori possible communication sequences arising in the computations of P_i during which the boolean expressions are not interpreted. Each communication sequence consists of the elements of the form $k:\langle i,j\rangle$ or $k:\langle j,i\rangle$ where k is a label of an i/o command uniquely identified and $\langle i,j\rangle$ ($\langle j,i\rangle$) records the fact that this i/o command stands for a communication from $P_i(P_j)$ to $P_j(P_i)$.

Example 2

Let $P \equiv$
$[P_1::*[i < 10 \rightarrow i:=i+1\ ;\ k_1:P_2!0\ ;\ k_2:P_2!i]$
$\|P_2::*[j < 10 \rightarrow l_1:P_1?x\ ;\ l_2:P_1?j]].$

Then

$$L(P_1) = \{(k_1:\langle 1,2\rangle)(k_2:\langle 1,2\rangle)\}^*$$

and

$$L(P_2) = \{(l_1:\langle 1,2\rangle)(l_2:\langle 1,2\rangle)\}^*. \ \Box$$

2) We associate with P a regular language L(P). Its letters are of the form k,l:<i,j> standing for an instance of a communication between the output command of P_i labeled by k and the input command of P_j labeled by l.

First we define a projection function $[.]_i$ $(1 \leq i \leq n)$ from the alphabet of L(P) into the alphabet of $L(P_i)$. We put

$$[k,l:\langle i,j\rangle]_i = k:\langle i,j\rangle,$$
$$[k,l:\langle i,j\rangle]_j = l:\langle i,j\rangle,$$
$$[k,l:\langle i,j\rangle]_h = \epsilon \text{ if } h \neq i,j,$$

and naturally extend it to a homomorphism from the set of words of L(P) into the set of words of $L(P_i)$.

We now define

$$L(P) = \{h: [h]_i \in L(P_i),\ i=1,\ldots,n\}.$$

Intuitively, L(P) is the set of all possible communication sequences of P which can arise in *properly terminating* computations of P during which the boolean expressions are not interpreted.

Finally we define

$$STAT = \{(k:\alpha,\ l:\beta) : k:\alpha \text{ is from } P_i,\ l:\beta \text{ is from } P_j,\ \& \exists h \exists a\ [h \in L(P),\ a \text{ is an element of } h,\ L(k:\alpha) = \{[a]_i\} \text{ and } L(l:\beta) = \{[a]_j\}\}.$$

Intuitively, STAT (standing for *static match*) is the set of all pairs of i/o commands which can be synchronized during a properly terminating computation of P which ignores the boolean guards.

The set STAT should be compared with two other sets of pairs of i/o commands :

SYNT = {(k:α, l:β) : k:α and l:β match}
SEM = {(k:α, l:β) : in some "real" properly terminating computation of P k:α and l:β are synchronized}.

Obviously, in contrast to STAT and SYNT the set SEM as a function of P is not computable. We have $SEM \subseteq STAT \subseteq SYNT$.

Example 3

Consider the program P from example 2. We have
$L(P) = \{(k_1,l_1 : \langle 1,2\rangle)(k_2,l_2 : \langle 1,2\rangle)\}^*$

and consequently

$$STAT = \{(k_1:P_2!0,\ l_1:P_1?x),(k_2:P_2!i,\ l_2:P_1?j)\}.$$

On the other hand

$$SYNT = STAT \cup \{(k_1:P_2!1,l_2:P_1?j),(k_2:P_2!i,l_1:P_1?x)\}.\square$$

5.2 Proofs from assumptions

The proof systems presented in section 3 are appropriate for proving correctness of nondeterministic programs. While trying to extend these systems to deal with the CSP programs introduced above we first have to take care of the i/o commands.

Consider a parallel program

$$P \equiv [P_1 :: S_1 \| \ldots \| P_n :: S_n].$$

To be able to provide a proof of a formula $\{p_i\}\ S_i\ \{q_i\}$ for a given process P_i we lack information about the behaviour of its i/o commands. Consider them for a moment as actions with unspecified meaning, black boxes so to say, about which we shall make some assumptions.

Let A_i be a set of correctness formulas, one for each i/o command from S_i. In other words

$$A_i = \{\{p_\alpha\}\ \alpha\ \{q_\alpha\} : \alpha \text{ is an i/o command from } S_i\}.$$

We now write

$$A_i \vdash \{p_i\}\ S_i\ \{q_i\} \quad (\text{or } A_i \vdash \{p_i\}\ P_i\ \{q_i\})$$

to denote the fact that $\{p_i\}\ P_i :: S_i\ \{q_i\}$ can be proved from the set of assumptions A_i. If we want to stress which proof system is used - N or NT we shall subscript the provability sign "$\vdash$" appropriately. Lack of subscript in the example proofs will mean that both options hold. We shall freely use labels to distinguish different occurrences of the i/o commands.

Example 4

Consider the following trivial CSP program

$$P \equiv [P_1 :: P_2?z\ ;\ [x \geq z \rightarrow P_2!x \ \square\ x \leq z \rightarrow P_2!z] \ \| \ P_2 :: P_1!y\ ;\ P_1?u].$$

P_1 and P_2 hold values x and y, respectively. P_1 computes maximum of these two values and sends the result to P_2. A desired property of P to be proved is thus

$$\{\underline{true}\}\ P\ \{u = \max(x,y)\}.$$

Consider now the following lists of assumptions

$$A_1 = \{\{\underline{true}\}\ P_2?z\ \{y = z\},$$
$$\{x = \max(x,y)\}\ P_2!x\ \{x = \max(x,y)\},$$
$$\{z = \max(x,y)\}\ P_2!z\ \{z = \max(x,y)\}\},$$

$$A_2 = \{\{\underline{true}\}\ P_1!y\ \{\underline{true}\},$$
$$\{\underline{true}\}\ P_1?u\ \{u = \max(x,y)\}\}.$$

We now claim that

$$A_1 \vdash \{\underline{true}\}\ P_1\ \{x = \max(x,y) \vee z = \max(x,y)\} \qquad (1)$$

and

$$A_2 \vdash \{\underline{true}\}\ P_2\ \{u = \max(x,y)\}. \qquad (2)$$

The proofs are straightforward. Let q stand for the formula $x = \max(x,y) \vee z = \max(x,y)$.

To prove (1) it suffices to prove

$$A_1 \vdash \{y = z\}\ [x \geq z \rightarrow P_2!x \ \square\ x \leq z \rightarrow P_2!z]\{q\}$$

i.e. thanks to the rule of alternative commands

$$A_1 \vdash \{y = z \wedge x \geq z\}\ P_2!x\ \{q\} \qquad (3)$$

and

$$A_1 \vdash \{y = z \wedge x \leq z\}\ P_2!z\ \{q\}. \qquad (4)$$

But

$$(y = z \wedge x \geq z) \rightarrow x = \max(x,y)$$

and

$$(y = z \wedge x \leq z) \rightarrow z = \max(x,y),$$

so by the rule of consequence (3) and (4) hold.

The proof of (2) is obvious.□

5.3 Tying proofs together – the cooperation test

So far we have only introduced a way of writing that behaviour of a process depends on the behaviour of its i/o commands. The next step consists of *discharging* the introduced assumptions by checking their mutual consistency.

Consider statements (1) and (2) from example 4. We would like now to check that the assumptions made about the i/o commands were justified. How can we do that ? The idea is quite simple.

Take for example the assumption $\{\underline{true}\}\ P_2?z\ \{y=z\}$. From the text of the program we see that the only output command of P_2 which can synchronize with $P_2?z$ is $P_1!y$. And indeed, after the communication between $P_2?z$ and $P_1!y$, $y = z$ holds.

We formalize this idea as follows. First, the effect of a communication is that of an assignment. Thus we adopt the following natural axiom

AXIOM 12 : COMMUNICATION AXIOM

$$\{p[t/x]\}\ P_i?x \parallel P_j!t\ \{p\}$$

where $P_i?x$ and $P_j!t$ are taken from P_j and P_i respectively.

Next, given two assumptions $\{p^i\}\ P_i?x\ \{q^i\}$ and $\{p^j\}\ P_j!t\ \{q^j\}$ used in the proofs of P_j and P_i, respectively, we say that they *cooperate* if

$$\{p^i \wedge p^j\}\ P_i?x \parallel P_j!t\ \{q^i \wedge q^j\} \qquad (5)$$

can be proved.

Intuitively speaking, (5) guarantees that if p^i and p^j hold when the control is before $P_i?x$ and $P_j!t$, respectively then q^i and q^j hold when the control is after $P_i?x$ and $P_j!t$, respectively. The notion of

cooperation is due to [AFR] and to [LG] where it is called satisfaction.

Remark 1

Note that (5) can be proved iff

$\{p^i \wedge p^j\}\ x:=t\ \{q^i \wedge q^j\}$ can be proved which in turn can be proved iff $(p^i \wedge p^j) \rightarrow (q^i \wedge q^j)[t/x]$ holds.□

Generalization to the case of sets of assumptions makes use of the static analysis carried out in section 5.1.

Definition 2 We say that two assumptions and *statically match* if the pair of their i/o commands statically matches, i.e. belongs to STAT.

Definition 3 Let $A_1,\ldots,A_n$ be sets of assumptions for the proofs of processes $P_1,\ldots,P_n$, respectively. We say that $A_1,\ldots,A_n$ *cooperate* if every statically matching pair of assumptions from $A_1 \cup\ldots\cup A_n$ cooperates.

A given assumption in general will have to be checked against several other assumptions for cooperation. The choice of the set STAT in the above definition is motivated by the fact that it is the best known for us computable approximation to the set SEM. Note that the cooperation test takes care of all pairs of i/o commands from the set SEM.

If the sets of assumptions $A_1,\ldots,A_n$ cooperate then we can discharge them. This leads us to the following proof rule being a generalization of the rule of disjoint parallel composition.

RULE 13 : RULE FOR PARALLEL COMPOSITION

$A_i \vdash \{p_i\}\ S_i\ \{q_i\}$, $i=1,\ldots,n$,
$free(p_\alpha,q_\alpha,p_i,q_i) \cap change(S_j) = \emptyset$ for all $\{p_\alpha\}\ \alpha\ \{q_\alpha\} \in A_i$ and $i \neq j$,

$A_1,\ldots,A_n$ cooperate

$$\{\bigwedge_{i=1}^{n} p_i\}\ [P_1 :: S_1 \parallel\ldots\parallel P_n :: S_n]\ \{\bigwedge_{i=1}^{n} q_i\}$$

Note that the list of hypotheses in the second line is an obvious extension of the appropriate condition from the rule of disjoint parallel composition. It is easy to see that this extension is necessary.

Example 5

We illustrate the use of the above rule by considering the program P from example 4. Let us now try to prove the desired formula

$\{\underline{true}\}\ P\ \{u = max(x,y)\}$.

We use the proofs from assumptions listed under (1) and (2). Note that the appropriate disjointness conditions are satisfied (y is not changed by P_2). We now prove that A_1 and A_2 cooperate. To this purpose we first

identify the statically matching pairs of i/o commands. These are :

$P_2?z,\ P_1!y$;
$P_2!x,\ P_2?u$;
$P_2!z,\ P_2?u$.

For each corresponding pair of the assumptions we have to establish the cooperation test. Take for example the second pair. We have to prove

$$\{x = max(x,y) \wedge \underline{true}\}\ P_2!x \parallel P_2?u\ \{x = max(x,y) \wedge u=max(x,y)\},$$

According to remark 1 it suffices to check that

$$x = max(x,y) \rightarrow (x = max(x,y) \wedge u = max(x,y))[x/u]$$

which is obvious. Other tests are equally straightforward.

Applying now the rule of parallel composition we get

$$\{\underline{true}\}\ P\ \{u = max(x,y) \wedge (x = max(x,y) \vee z = max(x,y))\}$$

from which the desired formula follows.□

5.4 A discussion

In contrast to the input command the output command seems to have an obvious meaning. Its execution changes no variable of the process to which it belongs. Thus one could naturally adopt the following output command axiom

$$\{p\}\ P_i!t\ \{p\}$$

After all this is the form of the assumptions about the output commands used in the proofs in the examples 4 and 5.

The cooperation test would then reduce to a verification that the post-assertions of the input commands hold, i.e. that $\{p\}\ P_i?x \parallel P_j!t\ \{q\}$ holds for every statically matching pair $(P_i?x,\ P_j!t)$ and assumption $\{p\}\ P_i?x\ \{q\}$.

Unfortunately such a simplification does not work. First, to establish the post-assertion of the input command one needs in general some information about the value sent. Consider for example the case when P_1 is of the form $\ldots P_2?x\ldots$ and P_2 is of the form $\ldots y:=2\ ;\ P_1!y\ldots$ where the exhibited i/o commands statically match. The above simplification does not allow to prove that after the communication has taken place, $x=2$ holds, an information which can be needed. In contrast, the adopted cooperation test allows to infer such an information - it suffices to use the assumption $\{y = 2\}\ P_1!y\ \{y = 2\}$, or even $\{y = 2\}\ P_1!y\ \{\underline{true}\}$.

Secondly, in some cases one has to make specific assumptions about the i/o commands which will never be executed.

Example 6

Consider the following program

$P \equiv [P_1 :: P_2!1$

$\| P_2 :: [\underline{true} \rightarrow P_1?x \square \underline{true} \rightarrow P_3!x]$

$\| P_3 :: skip].$

Note that $\{\underline{true}\}\ P\ \{x = 1\}$ holds in the sense of partial correctness - whenever P terminates properly, $x = 1$ holds. (Of course P does not need to terminate properly as a deadlock can arise - an issue we shall deal with in the next section).

To prove the desired formula we have to use the post-assertion $x = 1$ for the process P_2. This requires to use assumptions $\{\underline{true}\}\ P_1?x\ \{x = 1\}$ and $\{\underline{true}\}\ P_3!x\ \{x = 1\}$ in the proof of P_2. The second assumption seems to be completely arbitrary but it passes the cooperation test voidly.

It is now clear how to complete the proof and we leave it to the reader. □

In general, assumptions about the i/o commands will be of one of two types depending whether a given i/o command statically matches another i/o command. If it does then the assumption of an input command $P_i?x$ will in general be of the form $\{p[t/x]\}\ P_i?x\ \{p\}$ where t is an anticipated value of x after the communication and for an output command $P_i\ t$ it will be simply $\{p\}\ P_i!t\ \{p\}$ for an appropriately chosen p. If on the other hand the i/o command does not statically match any other i/o command then the corresponding assumption can be arbitrary.

5.5 Deadlock freedom

Once the syntax has been extended by allowing i/o commands another possibility of abnormal termination of programs has been introduced - that of deadlock. Formally, deadlock arises when not all processes have terminated but none of them can proceed. Such a situation can arise only when the following three conditions are satisfied :

1) each processes is in front of an i/o command or has terminated,

2) no pair of i/o commands mentioned in 1) matches,

3) not all processes have terminated.

As an example of a deadlock situation consider the program from the example 6. If process P_2 happens to choose the second branch of its alternative command then deadlock will arise once P_3 terminates.

In order to prove deadlock freedom of a given parallel program P we shall first have to identify all situations in which deadlock could arise. Similarly as it was the case with matching of i/o commands the set of such situations as a function of P is not computable. We once again resort to

static analysis.

Given a process P_i let $C(P_i)$ be the set of all i/o commands occuring in P_i augmented by the element end P_i.

We say that a tuple $\langle a_1,\ldots,a_n\rangle$ from $C(P_1)x\ldots x\ C(P_n)$ is *blocked* if

i) $\exists$ i $a_i \neq$ end P_i
(not all processes have terminated),

ii) $\neg\exists$ i,j $(a_i,a_j) \in$ SYNT
(no communication can take place).

We now identify all blocked tuples which can be reached during computations which ignore the boolean guards.

Let Init(L) denote for a formal language L its *left factor*, i.e. the set $\{u : \exists w\ (uw \in L)\}$.

We say that a tuple $\langle a_1,\ldots,a_n\rangle$ from $C(P_1)x\ldots x\ C(P_n)$ is *statically blocked* if

i) it is blocked,

ii) $\exists$ h $\searrow$i
$[a_i \neq$ end $P_i \rightarrow [h]_i c \in$ Init(L(P_i)) where $L(a_i) = \{c\}$
$\wedge\ a_i =$ end $P_i \rightarrow [h]_i \in L(P_i)]$

The second condition states that there exists a communication sequence which reaches the vector of the control points associated with $\langle a_1,\ldots,a_n\rangle$. Reachability is checked by considering the projections $[h]_i$ of the sequence h. If $a_i \neq$ end P_i then $[h]_i c$ where $L(a_i) = \{c\}$ should be an initial part of a sequence from $L(P_i)$. If $a_i =$ end P_i then $[h]_i$ should be a sequence from $L(P_i)$.

We now return to the issue of proving deadlock freedom. Consider a situation when n proofs from assumptions

$$A_i \vdash \{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n$$

satisfying the hypotheses of the rule of parallel composition are given.

With each statically blocked tuple $\langle a_1,\ldots,a_n\rangle$ we associate a tuple $\langle r_1,\ldots,r_n\rangle$ of assertions defined as follows :

if $a_i = k:\alpha$ then $r_i \equiv p_\alpha$ where $\{p_\alpha\}\ \alpha\ \{q_\alpha\} \in A_i$,
if $a_i =$ end P_i then $r_i \equiv q_i$.

If the parallel program P is executed in an initial (global) state satisfying the pre-assertions $p_1,\ldots,p_n$ then at the moment when the deadlock situation corresponding with a statically blocked tuple $\langle a_1,\ldots,a_n\rangle$ is reached, the corresponding assertions $r_1,\ldots,r_n$ hold. This fact can be formally proved once a formal semantics of CSP programs is given (see [A2]).

This observation leads us to the following theorem allowing to prove deadlock freedom.

We say that the program P is *deadlock free relative to the assertion* p if in the computations of P starting in a global state satisfying p deadlock cannot arise. If p ≡ true then we simply say that P is *deadlock free*.

Theorem 1 Let a proof of {p} P {q} be given. Then P is deadlock free relative to p if for all statically blocked tuples $\langle a_1,\ldots,a_n\rangle$ $\neg \bigwedge_{i=1}^{n} r_i$ holds for the corresponding tuple of assertions $\langle r_i,\ldots,r_n\rangle$.□

The theorem holds because as observed above every deadlock situation corresponds with a statically blocked tuple.

To illustrate use of the theorem consider the following two examples.

Example 7

Take the program P from example 4. We prove that P is deadlock free. To this purpose we first have to identify all statically blocked pairs. But it is easy to see that their set is empty. Thus P is automatically deadlock free. Here no proof theoretic analysis is necessary.□

Example 8

Consider the following program

$$P \equiv [P_1 :: P_2!1$$

$$\| P_2 :: [\underline{true} \rightarrow P_1?x \square x < 0 \rightarrow P_3!x]$$

$$\| P_3 :: skip].$$

We now prove that P is deadlock free with respect to the assertion x≥0. We first identify all statically blocked triples. It is easy to see that there is only one such tuple, namely $\langle P_2!1, P_3!x, \text{ end } P_3\rangle$. We now have to find appropriate proofs from assumptions. It is easy to see that

$$A_1 \vdash \{\underline{true}\}\ S_1\ \{\underline{true}\},$$

$$A_2 \vdash \{x\geq 0\}\ S_2\ \{x=1\},$$

$$A_3 \vdash \{\underline{true}\}\ S_3\ \{\underline{true}\}$$

where $A_1 = \{\{\underline{true}\}\ P_2!1\ \{\underline{true}\}\}$,

$$A_2 = \{\{x\geq 0\}\ P_1?x\ \{x=1\},\ \{\underline{false}\}\ P_3!x\ \{x=1\}\},$$

A_3 is empty and S_1, S_2, S_3 are bodies of the processes P_1, P_2 and P_3, respectively.

Note that the above proofs from assumptions satisfy the disjointness proviso from the rule of parallel composition. Also the cooperation test is easily established. Now consider the above identified statically blocked triple $\langle P_2!1,\ P_3!x,$ end $P_3\rangle$. The corresponding triple of assertions is $\langle$true, false, true$\rangle$ whose conjunction obviously cannot be realized. Thus P is deadlock free. Moreover, applying the rule of parallel composition we get $\{x\geq 0\}\ P\ \{x=1\}$ but now, thanks to the deadlock freedom of P, in the sense of total correctness.□

5.6 Global invariants and bracketed sections

The proposed approach unfortunately fails when dealing with more complicated programs. Consider the following example.

Example 9

Let $P \equiv [P_1 :: *[i<10 \rightarrow i:=i+1\ ;\ P_2!i]$
$\|P_2 :: *[j<10 \rightarrow P_1?j]]$.

We clearly have

$$\{i = 0 \wedge j = 0\}\ P\ \{i = 10 \wedge j = 10\} \qquad (6)$$

and we can prove this formula in the sense of partial correctness. Indeed, it is easy to see that

$$\{i \leq 10\}\ P_2!i\ \{i \leq 10\}\ \vdash_N\ \{i = 0\}\ P_1\ \{i = 10\}$$

and

$$\{j \leq 10\}\ P_1?j\ \{j \leq 10\}\ \vdash_N\ \{j = 0\}\ P_2\ \{j = 10\}.$$

Moreover, the disjointness proviso is satisfied and the cooperation test obviously holds. By the rule of parallel composition we get (6) in the sense of partial correctness.

A problem arises if we wish to prove (6) in the sense of total correctness. We have to use appropriately modified proofs from assumptions. It is easy to see that

$$\{i\leq 10 \wedge 10-(i-1) = n\}\ P_2!i\ \{i\leq 10 \wedge 10-(i-1) = n\}\ \vdash_{NT}\ \{i=0\}\ P_1\ \{i = 10\}$$

holds where for the loop invariant $p_1(n)$ we choose $p_1(n) \equiv i\leq 10 \wedge 10-i = n$.

Unfortunately things are not that simple in the case of P_2.

We clearly have

$$\{j<10 \wedge 10-j=\bar{n}\}\ P_1?j\ \{j\leq 10 \wedge 10-(j-1)=\bar{n}\}\ \vdash_{NT}\{j=0\}\ P_2\{j=10\}$$

where we take the loop invariant $p_2(\bar{n}) \equiv j\leq 10 \wedge 10-j = \bar{n}$. However, this choice of assumptions does not pass the cooperation test. The pre-assertions of the assumptions do not allow to relate the old value of j to the new one (i.e. to the value of i) and any other choice of assumptions seems to suffer from the same drawback. Moreover, this problem does not seem to have a solution even if the rule of auxiliary variables were allowed to be used.□

More complicated examples can be constructed to show that the proposed approach can fail for proofs of partial correctness, as well. To resolve these difficulties we introduce the notion of a *global invariant*. We intend to use in the proof the fact that $i=j$ holds whenever the control within P_1 is not immediately after the assignment $i:=i+1$. To express this fact we introduce brackets - here putting $\ldots\langle i:=i+1\ ;\ P_1!i\rangle\ldots$ - in order to delimit program sections within which the invariant - here $i=j$ - does not need to hold.

Definition 4 A process P_i *bracketed* if the brackets "<" and ">" are interspersed in its text, so that each program section $\langle S\rangle$ is of the form $\langle S_1\ ;\ \alpha\ ;\ S_2\rangle$ where S_1 and S_2 do not contain any i/o command and are possibly empty and α is an i/o command. Program sections $\langle S\rangle$ are called *bracketed sections*.

Once all processes are bracketed we can envisage a more restricted way of executing of CSP programs where a simultaneous execution of a matching pair of i/o commands is replaced by an uninterupted execution of the corresponding bracketed sections. This view of computations of CSP programs leads to the following modification of the approach so far adopted.

First, proofs of the individual processes use as assumptions hypotheses about bracketed sections rather that about i/o commands. Secondly, the cooperation test is now applied to the assumptions about the bracketed sections. Finally a global invariant I is introduced. This invariant is to be preserved by the given parallel program. To this purpose it is enough to ensure that

i) no free variable of I is subject to change outside a bracketed section. Thanks to this restriction I cannot be invalited by executing a command being outside a bracketed section.

ii) a parallel execution of any pair of bracketed sections associated with a statically matching pair of i/o commands preserves I.

Summarizing, we introduce the following definitions.

Definition 5

i) We say that two bracketed sections $\langle S_1\rangle$ and $\langle S_2\rangle$ *statically match* if they contain statically matching i/o commands.

ii) Given two assumptions $\{p^i\}\ \langle S_1\rangle\ \{q^i\}$ and $\{p^j\}\ \langle S_2\rangle\ \{q^j\}$ used in the proofs of P_i and P_j, respectively, we say that they *statically match* if the pair of their bracketed sections statically matches.

Definition 6 Let an invariant I be given.

i) Given two assumptions $\{p^i\}\ \langle S_1\rangle\ \{q^i\}$ and $\{p^j\}\ \langle S_2\rangle\ \{q^j\}$ used in the proofs of P_i and P_j, respectively, we say that they *cooperate with respect to* I if

$$\{p^i \wedge p^j \wedge I\}\ S_1 \parallel S_2\ \{q^i \wedge q^j \wedge I\}$$

can be proved.

ii) Given sets of assumptions $A_1,\ldots,A_n$ used in the proofs of processes $P_1,\ldots,P_n$, respectively, we say that $A_1,\ldots,A_n$ *cooperate with respect to* I if every statically matching pair of assumptions from $A_1\cup\ldots\cup A_n$ cooperates with respect to I.

To prove the correctness formula mentioned in the last definition we need a new proof rule.

RULE 14 : FORMATION RULE

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

provided α and $\bar{\alpha}$ match, S_1,S_2,S_3 and S_4 do not contain any i/o commands, are possibly empty, and no variable in $S_1\ ;\ S_2$ is subject to change in $S_3\ ;\ S_4$ and vice versa.

This rule is sound due to the fact that under the above stated restrictions any execution of the program $(S_1\ ;\ \alpha\ ;\ S_2)\ \|\ (S_3\ ;\ \bar{\alpha}\ ;\ S_4)$ is equivalent to an execution of the program $S_1;S_3;(\alpha\|\bar{\alpha});S_2;S_4$.

Finally the following rule is introduced.

RULE 15 : RULE OF PARALLEL COMPOSITION II

Suppose that all processes P_i, $i=1,\ldots,n$, are bracketed.

$$\frac{\begin{array}{l}A_i\ \vdash\ \{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n,\\ \text{free}(p_{\langle S\rangle},\ q_{\langle S\rangle},\ p_i,q_i)\ \cap\ \text{change}(S_j)\ =\ \emptyset\\ \quad\text{for all } \{p_{\langle S\rangle}\}\langle S\rangle\{q_{\langle S\rangle}\}\ \in\ A_i \text{ and } i\neq j,\\ \text{no variable free in } I \text{ is subject to change outside}\\ \quad\text{a bracketed section,}\\ \\ A_1,\ldots,A_n \text{ cooperate w.r.t. } I\end{array}}{\{\ \bigwedge_{i=1}^{n} p_i\ \wedge\ I\}[P_1::S_1\ \|\ldots\|\ P_n::S_n]\ \{\ \bigwedge_{i=1}^{n} q_i\ \wedge\ I\}}$$

The conclusion of the rule refers to the original form of the processes.

Intuitively, this rule is sound because every execution of a parallel program P corresponds to an execution of its bracketed version in which all pairs of matching bracketed sections are executed in an uninterupted manner.

To clarify the introduced notions we now reconsider the program studied in example 9.

Example 10

Consider the following bracketed version of the program P :

$$P \equiv [\ P_1:: *\ [i<10 \rightarrow \langle i:=i+1\ ;\ P_2!i\rangle]$$

$\| P_2 :: * [j<10 \rightarrow \langle P_1?j\rangle]]$

We choose $I \equiv i=j$. Note that neither i nor j can be changed outside a bracketed section. Similarly as in example 9 we choose the following proofs from assumptions which clearly hold :

$\{i<10 \wedge 10-i = n\}\ \langle i:=i+1;P_2!i\rangle\ \{i\leq 10 \wedge 10-(i-1)=n\} \vdash_{NT}\{i=0\}\ P_1\ \{i=10\}$,

$\{j<10 \wedge 10-j=\bar{n}\}\langle P_1?j\rangle\{j\leq 10 \wedge 10-(j-1)=\bar{n}\} \vdash_{NT}\{j=0\}P_2\{j=10\}$.

We now have to establish the cooperation test, i.e. to prove

$$\{i<10 \wedge 10-i = n \wedge j<10 \wedge 10-j = \bar{n} \wedge i=j\}$$
$$\langle i:=i+1\ ;\ P_2!i\rangle \parallel \langle P_1?j\rangle$$
$$\{i\leq 10 \wedge 10-(i-1) = n \wedge j\leq 10 \wedge 10-(j-1) = \bar{n} \wedge i=j\}$$

which is straightforward due to the formation rule. All the premises of rule 15 are now satisfied and we obtain

$\{i=0 \wedge j=0 \wedge i=j\}\ P\ \{i=10 \wedge j=10 \wedge i=j\}$,

and hence

$\{i=0 \wedge j=0\}\ P\ \{i=10 \wedge j=10\}$.

This formula is now proved in the sense of total correctness *provided* deadlock freedom of P relative to $i=0 \wedge j=0$ can be established. The issue of deadlock freedom will be treated in the next section.□

Remark 2

Note that in the proofs from assumptions we used here different parameter variables n and $\bar{n}$. Even though here no problems would result if the same parameter variables were chosen, in general caution has to be exercised because of possible clashes of variables in the cooperation test.

5.7 Deadlock freedom revisited

Once we modified our approach using global invariants we have to modify our way of dealing with deadlock freedom accordingly. First, we revise the definition of a blocked tuple. Suppose that all processes $P_1,\ldots,P_n$ are bracketed. Let now $B(P_i)$ be the set of all bracketed sections occuring in P_i augmented as before by the element end P_i.

To each tuple $\langle B_1,\ldots,B_n\rangle$ from $B(P_1)x\ldots x\ B(P_n)$ naturally corresponds a tuple $\langle a_1,\ldots,a_n\rangle$ from $C(P_1)x\ldots x\ C(P_n)$ where each bracketed section is replaced by its i/o command. Now, a tuple $\langle b_1,\ldots b_n\rangle$ from $B(P_1)x\ldots x\ B(P_n)$ is *statically blocked* if the corresponding tuple $\langle a_1,\ldots,a_n\rangle$ is statically blocked.

Suppose now that n proofs from assumptions

$$A_i \vdash \{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n$$

and a global invariant I satisfying the hypotheses of the rule of parallel composition II are given. With each statically blocked tuple $\langle B_1,\ldots B_n\rangle$ we associate a tuple $\langle r_1,\ldots r_n\rangle$ of assertions defined analogously as in section 5.5 :

if $B_i = \langle S\rangle$ then $r_i \equiv p_{\langle S\rangle}$ where $\{p_{\langle S\rangle}\}\ \langle S\rangle\ \{q_{\langle S\rangle}\} \in A_i$,
if $B_i =$ end p_i then $r_i \equiv q_i$.

The corresponding theorem about deadlock freedom takes now the following form :

Theorem 2 Consider a parallel program $P \equiv [P_1::S_1 \parallel\ldots\parallel P_n::S_n]$. Let a proof of $\{p\}\ P\ \{q\}$ with a global invariant I be given. Then P is deadlock free relative to p if for all statically blocked tuples $\langle B_1,\ldots,B_n\rangle$

$\neg(\bigwedge_{i=1}^{n} r_i \wedge I)$ holds for the corresponding tuple of assertions $\langle r_1,\ldots,r_n\rangle$

Intuitively speaking, this theorem holds because of the following. Assume a bracketing of P used in the proof of $\{p\}\ P\ \{q\}$. Consider a computation of P which starts in a global state satisfying the assertion p and which leads to a deadlock. Delete from this computations all steps performed by the processes $P_1,\ldots P_n$ within the last bracketed sections entered but not left. In other words each process which in the considered deadlock situation finds itself before an i/o command is "moved back" to the entrance of the associated bracketed section. With the resulting vector of n control points there corresponds a statically blocked tuple $\langle b_1,\ldots b_n\rangle$. It now suffices to show that for the corresponding tuple $\langle r_1,\ldots r_n\rangle$ of assertions the assertion $\bigwedge_{i=1}^{n} r_i \wedge I$ is satisfiable. But this is a direct consequence of the following theorem (see [A2]).

Theorem 3 Let a proof of partial (total) correctness of $\{p\}\ P\ \{q\}$ with a global invariant I be given. Consider a computation of P which starts in a global state satisfying p. If in this computation a process P_i is about to enter a bracketed section or has terminated then the corresponding assertion r_i ($\exists\ n_1,\ldots n_k r_i$ where $n_1,\ldots,n_k$ are parameter variables occuring in r_i) holds. If none of the processes is within a bracketed section then I holds.□

This concludes our discussion of theorem 2. We now return to the program from examples 9 and 10.

Example 11

Consider the proofs from assumptions and the global invariant I provided in example 10. We now prove that P is deadlock free with respect to the assertion $i=0 \wedge j=0$. Let a_1 and a_2 denote the bracketed sections of P_1 and P_2, respectively. The only statically blocked pairs are $\langle a_1,$end $P_2\rangle$ and $\langle$end $P_1,a_2\rangle$. The corresponding pairs of assertions are $\langle i<10 \wedge 10-i=n,\ j=10\rangle$ and $\langle i=10,\ j<10 \wedge 10-j=n\rangle$, respectively.

Clearly $\neg((i<10 \wedge 10-i=n) \wedge (j=10) \wedge (i=j)$ holds and analogously for the second pair of assertions. According to theorem 2 P is deadlock free relative to $i=0 \wedge j=0$.

Summarizing, we have proved the formula $\{i=0 \wedge j=0\}\ P\ \{i=10 \wedge j=10\}$ in the sense of *total correctness*.□

5.8 Auxiliary variables

The final point we have to consider is the use of auxiliary variables. They have been introduced in section 4 in the context of disjoint parallel programs. Since the class of programs we consider here contains those studied in section 4, we naturally need them here, as well. In fact it is not sufficient to use here the substitution rule (rule 10) only. Consider the following example.

Example 12

Let $P \equiv [\ P_1 :: *[i<10 \rightarrow i:=i+1\ ;\ P_2?x\ ;\ P_2!i]$
$\|\ P_2 :: *[j<10 \rightarrow P_1!1\ ;\ P_1?j]]$

We would like now to prove

$$\{i=0 \wedge j=0\}\ P\ \{i=10 \wedge j=10\} \qquad (7)$$

in the sense of total correctness.

Unfortunately, contrary to example 10 we cannot now "couple" the execution of the assignment $i:=i+1$ with that of the i/o command $P_2!i$ within P_1. A solution which naturally suggests itself is to use an auxiliary variable, say k, which would play the "role" of i. We thus modify the program P obtaining

$$P' \equiv [P_1 :: *[i<10 \rightarrow i:=i+1\ ;\ \langle P_2?x \rangle\ ;\ \langle k:=k+1\ ;\ P_2!i\rangle]$$
$$\|P_2 :: *[j<10 \rightarrow \langle P_1!1\rangle\ ;\ \langle P_1?j\rangle]]$$

where P' is already bracketed.

We use the following proofs from assumptions which are easily justified

$\{i\leq 10 \wedge 10-(i-1) = n \wedge k=i-1\}\ \langle P_2?x\rangle\ \{i\leq 10 \wedge 10-(i-1)=n \wedge k=i-1\}$,
$\{i\leq 10 \wedge 10-(i-1) = n \wedge k=i-1\}\ \langle k:=k+1;P_2!i\rangle\ \{i\leq 10 \wedge 10-(i-1)=n \wedge k=i\}$
$\vdash_{NT} \{i=0 \wedge k=0\}\ P_1\ \{i=10 \wedge k=10\}$

where for the loop invariant $p(n)$ we choose $p(n) \equiv i\leq 10 \wedge 10-i=n \wedge k=i$

and

$\{j<10 \wedge 10-j=\bar{n}\}\ \langle P_1!1\rangle\ \{j<10 \wedge 10-j=\bar{n}\}$,
$\{j<10 \wedge 10-j=\bar{n}\}\ \langle P_1?j\rangle\ \{j\leq 10 \wedge\ 10-(j-1)=\bar{n}\}$
$\vdash_{NT} \{j=0\}\ P_2\ \{j=10\}$

where for the loop invariant $p(\bar{n})$ we choose $p(\bar{n}) \equiv j\leq 10 \wedge 10-j=\bar{n}$.

Let $I \equiv k=j$. Note that neither k nor j can be modified outside a bracketed section. Also the proofs from assumptions satisfy the disjointness proviso of rule 15.

We now handle the cooperation test. There are only two pairs of statically matching bracketed sections. The case of the pair $\langle P_2?x\rangle$ and $\langle P_1!1\rangle$ is trivial. For the second case we have to show

$\{i \leq 10 \wedge 10-(i-1) = n \wedge k=i-1 \wedge j<10 \wedge 10-j=\bar{n} \wedge k=j\}$
$\langle k:=k+1 \; ; \; P_2!i \rangle \parallel \langle P_1?j \rangle$
$\{i \leq 10 \wedge 10-(i-1)=n \wedge k=i \wedge j \leq 10 \wedge 10-(j-1)=\bar{n} \wedge k=j\}$

which is obvious as thanks to the communication axiom and the formation rule we can simply replace the program considered by $k:=k+1 \; ; \; j:=i$.

By the rule of parallel composition II we now get

$\{i=0 \wedge k=0 \wedge j=0 \wedge k=j\}$ P' $\{i=10 \wedge k=10 \wedge j=10 \wedge k=j\}$.

To complete the proof we only need to get rid of the references to the variable k both in the assertions and in the program. First note that by the consequence rule

$\{i=0 \wedge k=0 \wedge j=0 \wedge k=j\}$ P' $\{i=10 \wedge j=10\}$.

Applying now the rule of auxiliary variables from section 4 we obtain

$\{i=0 \wedge k=0 \wedge j=0 \wedge k=j\}$ P $\{i=10 \wedge j=10\}$.

It is now sufficient to apply the substitution rule with the substitution $[0/k]$ to get the desired formula (7).

To be more precise we still have to prove deadlock freedom of P relative to $i=0 \wedge j=0$. To this purpose we use the proofs from assumptions given above. The only statically blocked pairs are

$\langle\langle P_2?x \rangle$, end $P_2\rangle$ and $\langle$end P_1, $\langle P_1!1\rangle\rangle$.

Consider the first pair. According to theorem 2 we have to check that

$\neg((i \leq 10 \wedge 10-(i-1)=n \wedge k=i-1) \wedge (j=10) \wedge (k=j))$

which is obvious. For the second pair we have to check

$\neg(j<10 \wedge 10-j=\bar{n}) \wedge i=10 \wedge k=10 \wedge k=j)$

which is equally obvious.

Thus by theorem 2 the program P' is deadlock free relative to $i=0 \wedge j=0$. To conclude the proof it now suffices to use the following theorem.

Theorem 4 Let A be a set of auxiliary variables of a parallel program P' and let P be obtained from P' by deleting all assignments to the variables in A. Then for any assertion p P' is deadlock free relative to p iff P is deadlock free relative to p.□

This concludes the proof of (7).□

We thus add to our proof system the substitution rule and the rule of auxiliary variables. Also, when proving deadlock freedom of parallel programs

we use theorem 2 in conjunction with theorem 4.

The presentation of our proof system for the language introduced at the beginning of this section is now complete.

6. I/O COMMANDS AS GUARDS

The approach presented in the previous section can be easily extended to handle other constructs of CSP. We deal in this section with an aspect of the language so far left out of the considerations - i/o commands used as guards.

6.1 New proof rules for alternative and repetitive commands

We extend the syntax of the processes by generalizing the notion of a guard. So far guards were simply boolean expressions. We now allow the i/o commands to occur as guards. In general, a guard will be a boolean expression, an i/o command (called a *pure i/o guard*) or a boolean expression followed by an i/o command. Thus a general form of an alternative command is now

$$[\ \square_{i=1}^{m_1} b_i \rightarrow R_i \ \square\ \square_{j=1}^{m_2} k_j!\alpha_j \rightarrow S_j \ \square\ \square_{l=1}^{m_3} c_l;k_{m_2+l}!\alpha_{m_2+l} \rightarrow T_l\ ] \qquad (8)$$

where $m_1,m_2,m_3 \geq 0$, b_i,c_l are boolean expressions and R_i,S_j and T_l programs. We labelled here for further reference the i/o guards. Repetitive commands are as before of the form $*S$ for an alternative command S. A communication can now take place between two i/o commands either of which can be in a guard position. A guard α or $b;\alpha$ can be passed if its boolean part (if it exists) evaluates to true and the i/o command α gets executed by performing a communication with a matching i/o command of another process. The following example clarifies the type of nondeterminism arising when the i/o guards are used.

Example 13

Consider the following program

$P \equiv [\ P_1 :: P_2!1$

$\|\ P_2 :: [P_1?x \rightarrow skip \ \square\ P_3!x \rightarrow skip]$

$\|\ P_3 :: skip]$

Here in P_2 the second guard cannot be passed since no communication with P_3 can take place. Eventually the first guard will be passed and the program will terminate with x equal 1. We in fact have

$$\{true\}\ P\ \{x=1\}$$

in the sense of total correctness as we shall prove later. This should be contrasted with the case of the parallel program studied in example 6 in which P_2 was of the form $[true \rightarrow P_1?x \ \square\ true \rightarrow P_3!x]$. There a deadlock *could* arise because in P_2 the second guard could have been chosen.□

Repetitive commands are exited once all guards *fail*. We assume here that a guard fails if it has a boolean part which evaluates to false. Thus repetitive commands containing a pure i/o guard cannot be exited.

To prove correctness of the programs allowing the i/o guards we provide new proof rules for alternative commands and repetitive commands.

RULE 16 : ALTERNATIVE COMMAND RULE III

$\{p \wedge b_i\}\ R_i\ \{q\},\ i=1,\ldots,m_1,$

$\{p\}\ \alpha_j \rightarrow S_j\ \{q\},\ j=1,\ldots,m_2,$

$\{p \wedge c_l\}\alpha_{m_2+l} \rightarrow T_l\ \{q\},\ l=1,\ldots,m_3$

$$\{p\}\ [\ \Box_{i=1}^{m_1} b_i \rightarrow R_i \ \Box\ \Box_{j=1}^{m_2} \alpha_j \rightarrow S_j \ \Box\ \Box_{l=1}^{m_3} c_l;\alpha_{m_2+l} \rightarrow T_l\ ]\ \{q\}$$

RULE 17 : REPETITIVE COMMAND RULE III

$\{p \wedge b_i\}\ R_i\ \{p\},\ i=1,\ldots,m_1,$

$\{p\}\ \alpha_j \rightarrow S_j\ \{p\},\ j=1,\ldots,m_2,$

$\{p \wedge c_l\}\ \alpha_{m_2+l} \rightarrow T_l\ \{p\},\ l=1,\ldots,m_3$

$$\{p\}*[\ \Box_{i=1}^{m_1} b_i \rightarrow R_i \ \Box\ \Box_{j=1}^{m_2} \alpha_j \rightarrow S_j \ \Box\ \Box_{l=1}^{m_3} c_l;\alpha_{m_2+l} \rightarrow T_l\ ]\{p \wedge \wedge_{i=1}^{m_1} \neg b_i \wedge \wedge_{l=1}^{m_3} \neg c_l\}$$

Of course these rules are appropriate for proofs of partial correctness only as absence of failure and termination are not guaranteed.

Appropriate rules for proofs of total correctness are the following modifications of the above two rules.

RULE 18 : ALTERNATIVE COMMAND RULE IV

$$p \rightarrow (\ \vee_{i=1}^{m_1} b_i \ \vee\ \vee_{j=1}^{m_2} c_j),$$

$\{p \wedge b_i\}\ R_i\ \{q\},\ i=1,\ldots,m_1,$

$\{p \wedge c_j\}\ \alpha_j \rightarrow S_j\ \{q\},\ j=1,\ldots,m_2$

$$\{p\}\ [\ \Box_{i=1}^{m_1} b_i \rightarrow R_i \ \Box_{j=1}^{m_2} c_j\ ;\ \alpha_j \rightarrow S_j\ ]\ \{q\}$$

RULE 19 : REPETITIVE COMMAND RULE IV

$$\frac{\{p(\bar{n}) \wedge b_i\}\ R_i\ \{\exists\, \bar{m} <_{\bar{n}} p(\bar{m})\},\ i=1,\ldots,m_1, \qquad \{p(\bar{n}) \wedge c_j\}\ \alpha_j \to S_j\ \{\exists\, \bar{m} < \bar{n}\ p(\bar{m})\},\ j=1,\ldots,m_2}{\{\exists\, \bar{n}\ p(\bar{n})\}*[\ \underset{i=1}{\overset{m_1}{\Box}} b_i \to R_i \ \Box\ \underset{j=1}{\overset{m_2}{\Box}} c_j;\alpha_j \to S_j]\{\exists\, \bar{n}\ p(\bar{n}) \wedge \underset{i=1}{\overset{m_1}{\wedge}} \neg b_i \wedge \underset{j=1}{\overset{m_2}{\wedge}} \neg c_j\}}$$

where $p(\bar{n})$ is an assertion with a free variable $\bar{n}$ which does not appear in the considered repetitive command. Both $\bar{n}$ and $\bar{m}$ range over natural numbers.

The above four rules have to be used in conjunction with the following natural rule allowing to prove their premises.

RULE 20 : ARROW RULE

$$\frac{\{p\}\ \alpha;S\ \{q\}}{\{p\}\ \alpha \to S\{q\}}$$

Note that in rules 18 and 19 no alternative commands or repetitive commands with pure i/o guards are taken care of. This is natural. A failure arises if all guards of an alternative command fail. But according to the definition of a failure of a guard this can arise only if the alternative command has no pure i/o guards. Thus the appropriate rule needs to be provided *only* for such a type of alternative commands. For alternative commands with pure i/o guards rule 16 should be used.

In contrast, repetitive commands with pure i/o guards can never be exited. Thus their termination can be proved *only* if no pure i/o guards are present. Consequently for repetitive commands with pure i/o guards no rule for proofs of total correctness is provided.

Summarizing, for proofs (from assumptions) of partial correctness rules 16 and 17 are used. For proofs (from assumptions) of total correctness rules 16, 18 and 19 are used where for alternative commands with no pure i/o guards *exclusively* rule 18 is used.

6.2 Proofs of partial correctness

The above introduced proof rules allow us to provide proofs of processes from the assumptions about their i/o commands - now also allowed as guards. What is left to do is to complete the approach by appropriately extending its other building blocks - static analysis and bracketed sections.

Consider first the issue of static analysis. Things are simple - it suffices to extend the approach of section 5.1 to programs admitting i/o guards. We complete the definition of the language $L(P_i)$ given in section 5.1 by putting for the guards of the form $b;k:\alpha$ $L(b;k:\alpha) = L(b)\ L(k:\alpha)$ and retaining all other clauses but now understood in a wider context where guards can contain i/o commands.

We can now adopt the approach of section 5.3 without any changes provided the proofs of partial, respectively total correctness are used as explained above. The next step consists of refining this approach along the lines of section 5.6. To this purpose we generalize the notion of a *bracketed section*. They can be now of one of the following forms :

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

where S_1 and S_2 do not contain any i/o commands. No other changes are needed for proofs of partial correctness - the arrow rule together with the formation rule allows to handle the case of cooperation when one or both bracketed sections are of the form $\alpha \rightarrow S$.

6.3 Deadlock freedom I

To prove deadlock freedom of the programs here considered we have to in the first place revise the definitions of blocked and statically blocked tuples originally given in sections 5.5 and 5.7 as in the presence of i/o guards other deadlock situations can arise.

Given a process P_i we define

$$D(k{:}\alpha) = \{\{k{:}\alpha\}\}$$

where $k{:}\alpha$ occurs in the process as an atomic command,

$$D([\ \underset{i=1}{\overset{m_1}{\square}}\ b_i \rightarrow R_i \ \square\ \underset{j=1}{\overset{m_2}{\square}}\ k_j{:}\alpha_j \rightarrow S_j \ \square\ \underset{l=1}{\overset{m_3}{\square}}\ c_l;k_{m_2+l}{:}\alpha_{m_2+l} \rightarrow T_l\ ]) =$$

$\{A{:}A = \{k_i{:}\alpha_i : i=1,\ldots m_2\} \cup B$ where A is non-empty and $B \subseteq \{k_{m_2+l}{:}\alpha_{m_2+l} : l=1,\ldots,m_3\}\}$, where $m_1 \geq 0$ and $m_2 + m_3 \geq 1$,

$D(* S) = D(S)$ where S contains at least one i/o guard.

For other types of commands S $D(S)$ is not defined. Note that a typical set belonging to $D(S)$ for an alternative command S consists of *all* i/o guards which occur without the boolean guards together with a subset of those i/o guards which occur with a boolean guard.

Given a process P_i we define now $G(P_i)$ to be the union of all sets $D(S)$ for S being a subprogram of P_i together with the element $\{\text{end } P_i\}$. Each element of $G(P_i)$ corresponds to a unique control point within P_i - the control resides either in front of an i/o command, or an alternative or repetitive command with i/o guards or at the end of the process. With each control point of the second type we associated a set of i/o commands which can be at this point executed. This set should not be empty - if it were this would correspond to a failure situation.

We say that a tuple $\langle D_1,\ldots,D_n\rangle$ from $G(P_1)x\ldots x\ G(P_n)$ is *blocked* if

i) $\exists\ i\ D_i \neq \{\text{end } P_i\}$
(not all processes have terminated),

ii) $(\ \underset{i\neq j}{\cup}\ D_i \times D_j) \cap SYNT = \emptyset$
(no communication can take place).

Alternatively ii) can be stated as : no pair of elements from D_i and D_j $(i \neq j)$ matches.

As in section 5.5 we now identify all blocked tuples which can be reached during computations which ignore the boolean guards.

We say that a tuple $\langle D_1,\ldots,D_n\rangle$ from $G(P_1)x\ldots x\ G(P_n)$ is *statically blocked* if

i) it is blocked

ii) $\exists h \bigvee i$

$[D_i \neq \{\text{end } P_i\} \rightarrow [h]_i\ c \in \text{Init}(L(P_i))$ for all $c \in \{L(a) : a \in D_i\}$

$\Lambda\ D_i = \{\text{end } P_i\} \rightarrow [h]_i \in L(P_i)]$.

The second condition is a straightforward modification of the corresponding condition from section 5.5. It simply states that there exists a sequence of communications which reaches the vector of control points associated with the tuple $\langle D_1,\ldots,D_n\rangle$.

Suppose now that n proofs from the assumptions about the i/o commands

$$A_i \vdash \{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n$$

which satisfy the hypotheses of the rule of parallel composition (rule 13) are given. With each statically blocked tuple $\langle D_1,\ldots,D_n\rangle$ from $G(P_1)x\ldots x\ G(P_n)$ we associate a tuple $\langle r_1,\ldots,r_n\rangle$ of assertions defined as follows :

if $D_i = \{k:\alpha\}$ where $k:\alpha$ is an i/o command from P_i then $r_i \equiv p_\alpha$ where $\{p_\alpha\}\ \alpha\ \{q_\alpha\} \in A_i$,

if $D_i = \{k_i:\alpha_i : i=1,\ldots,m_2\} \cup B$ where $B \subseteq \{k_{m_2+l}:\alpha_{m_2+l} : l=1,\ldots,m_3\}$ and D_i is associated with an alternative command (8) or a repetitive command then

$$r_i \equiv \bigwedge_{k:\beta\in D_i} p_\beta \wedge \bigwedge_{l\in\bar{B}} c_l \wedge \bigwedge_{l\notin\bar{B}} \neg c_l \wedge \bigwedge_{i=1}^{m_1} \neg b_i$$

where $\bar{B} = \{l : k_{m_2+l}:\alpha_{m_2+l} \in B\}$ and for each i/o command β $\{p_\beta\}\ \beta\ \{q_\beta\} \in A_i$,

if $D_i = \{\text{end } P_i\}$ then $r_i \equiv q_i$.

The second clause of the above definition requires some explanation. All i/o guards belonging to D_i can be, informally speaking, executed. Thus the corresponding preconditions p_β should hold for all of them. Moreover, the boolean guards associated with the i/o guards from B should hold (because these guards can be executed), the boolean guards associated with the i/o guards not in B should evaluate to <u>false</u> (because these guards are not listed in D_i and consequently cannot be executed) and the boolean guards b_i for $i=1,\ldots,m_1$ should all evaluate to <u>false</u> (because otherwise a progress in the process P_i could be made - remember that we are trying to identify deadlock situations).

To prove deadlock freedom of parallel programs admitting i/o guards theorem 1 from section 5.5 can now be used without any changes.

We now illustrate the approach of this section by considering the following two examples.

Example 14

Consider the program P from example 13. We clearly have :

$\{\underline{true}\}\ P_2!1\ \{\underline{true}\} \vdash_N \{\underline{true}\}\ P_1\ \{\underline{true}\}$,
$\{\underline{true}\}\ P_1?x\ \{x=1\}$, $\{\underline{true}\}\ P_3!x\ \{x=1\} \vdash_N \{\underline{true}\}\ P_2\ \{x=1\}$,
$\vdash_N \{\underline{true}\}\ P_3\ \{\underline{true}\}$.

The cooperation test is easily established so by the parallel composition rule $\{\underline{true}\}\ P\ \{x=1\}$ holds.

To prove this formula in the sense of total correctness we need only to show that P is deadlock free. To this purpose we should identify first all statically blocked triples. But their set is empty. Thus the condition of theorem 1 is voidly satisfied and P is indeed deadlock free.

Note that in the above proofs from assumptions we used an assumption about the i/o guard $P_3!x$ which passed the cooperation test voidly (see also the discussion in section 5.4).□

Example 15

A more interesting program to consider is

$P \equiv [\ P_1 :: P_2!1$

$\|\ P_2 :: [b;\ P_1?x \rightarrow skip \square P_3!x \rightarrow skip]$

$\|\ P_3 :: skip]$.

We now prove $\{b\}\ P\ \{x=1\}$ in the sense of total correctness. It is easy to see that

$\{b\}\ P_1?x\ \{x=1\}$, $\{b\}\ P_3!x\ \{x=1\} \vdash_N \{b\}\ P_2\ \{x=1\}$.

Taking for P_1 and P_3 the proofs from assumptions given in the previous example we get by the parallel composition rule $\{b\}\ P\ \{x=1\}$ in the sense of total correctness "modulo" deadlock freedom.

To prove deadlock freedom we first list all statically blocked triples. There is only one such triple – $\langle\{P_2!1\}, \{P_3!x\}, \{\text{end } P_3\}\rangle$.

The corresponding triple of assertions is $\langle\underline{true}, b \wedge \neg b, \underline{true}\rangle$. Their conjunction is inconsistent so according to theorem 1 P is deadlock free relative to b.□

6.4 Deadlock freedom II

We now consider the case when bracketed sections and global invariants are used. The approach of the previous section has to be refined appropriately. The modifications are analogous as those which have been made in section 5.7.

Suppose that all processes $P_1,\ldots,P_n$ are bracketed. Let now $F(P_i)$ be the set of all non-empty sets of bracketed sections of P_i augmented by the set {end P_i}.

With each tuple $\langle F_1,\ldots,F_n\rangle$ from $F(P_1)x\ldots x\ F(P_n)$ we can associate a tuple $\langle D_1,\ldots,D_n\rangle$ where each D_i is either a non-empty set of i/o commands of P_i or the set {end P_i} by simply replacing every bracketed section by its i/o command. We now say that a tuple $\langle F_1,\ldots,F_n\rangle$ from $F(P_1)x\ldots x\ F(P_n)$ is *statically blocked* if the corresponding tuple $\langle D_1,\ldots,D_n\rangle$ is statically blocked.

Suppose now that n proofs from assumptions

$$A_i \vdash \{p_i\}\ S_i\ \{q_i\},\ i=1,\ldots,n$$

and a global invariant I satisfying the hypotheses of the rule of parallel composition II are given. With each statically blocked tuple $\langle F_1,\ldots,F_n\rangle$ we associate a tuple $\langle r_1,\ldots,r_n\rangle$ of assertions defined as follows :

if F_i is a singleton $\{\langle S\rangle\}$ associated with an i/o command of P_i then $r_i \equiv p_{\langle S\rangle}$ where $\{p_{\langle S\rangle}\}\ \langle S\rangle\ \{q_{\langle S\rangle}\} \in A_i$,

if F_i is associated with an alternative command (8) or a repetitive command then

$$r_i \equiv \bigwedge_{\langle S\rangle\in F_i} p_{\langle S\rangle} \wedge \bigwedge_{l\in\bar{B}} c_l \wedge \bigwedge_{l\notin\bar{B}} \neg c_l \wedge \bigwedge_{i=1}^{m_1} \neg b_i$$

where $\bar{B} = \{l: k_{m_2+1}:\alpha_{m_2+1} \in B\}$ and for each bracketed section $\langle S\rangle$

$\{p_{\langle S\rangle}\}\ \langle S\rangle\ \{q_{\langle S\rangle}\} \in A_i$,

if $F_i = \{\text{end } p_i\}$ then $r_i \equiv q_i$.

The second clause of this definition is motivated by the same reasons as the corresponding clause provided in the previous section. To prove deadlock freedom one can now use without any changes theorem 2 in conjunction with theorem 4. We conclude this section by considering the following simple example which necessitates the use of bracketed sections.

Example 16

Take the following program P being a modification of the program considered in examples 9 and 10 :

$$P \equiv [P_1 :: * [i\leqslant 10\ ;\ P_2\ i \to i:=i+1]$$
$$\|\ P_2 :: * [j<10\ ;\ P_1?j \to \text{skip}]$$

We now prove

$$\{i=1 \wedge j=0\}\ P\ \{i=11 \wedge j=10\} \qquad (9)$$

in the sense of total correctness.

We choose $I \equiv i=j+1$ and a natural bracketing of P_1 and P_2 used in the following proofs from assumptions :

$$\{i<11 \wedge 11-i=n\}\ \langle P_2!i \rightarrow i:=i+1\rangle\ \{i\leq 11 \wedge 11-(i-1)=n\}\ \vdash_{NT} \{i=1\}\ P_1\ \{i=11\},$$

$$\{j<10 \wedge 10-j=\bar{n}\}\ \langle P_1?j \rightarrow skip\rangle\{j\leq 10 \wedge 10-(j-1)=\bar{n}\}\ \vdash_{NT} \{j=0\}\ P_2\ \{j=10\}.$$

which clearly hold. The appropriate loop invariants are respectively

$$P_1(n) \equiv i\leq 11 \wedge 11-i = n$$

and

$$P_2(\bar{n}) \equiv j\leq 10 \wedge 10-i = \bar{n}.$$

To prove the cooperation test we have to show

$$\{i\leq 11 \wedge 11-i = n \wedge j<10 \wedge 10-j = \bar{n} \wedge i=j+1\}$$
$$\langle P_2!i \rightarrow i:=i+1\rangle \parallel \langle P_1?j \rightarrow skip\rangle$$
$$\{i\leq 11 \wedge 11-(i-1) = n \wedge j\leq 10 \wedge 10-(j-1) = \bar{n} \wedge i=j+1\}$$

which is straightforward as the parallel subprogram considered is equivalent to $j:=i\ ;\ i:=i+1$.

By the rule of parallel composition II we get (9) provided we can establish deadlock freedom of P relative to the assertion $i=1 \wedge j=0$.

We first identify all statically blocked pairs. There are two such pairs :

$$\langle\{\langle P_2!i \rightarrow i:=i+1\rangle\},\ \{\text{end } P_2\}\rangle$$

and

$$\langle\{\text{end } P_1\},\ \{\langle P_1?j \rightarrow skip\rangle\}\rangle.$$

The corresponding pairs of assertions are

$$\langle i<11 \wedge 11-i = n,\ j = 10\rangle$$

and

$$\langle i=11,\ j<10 \wedge 10-j = \bar{n}\rangle.$$

To prove deadlock freedom it now suffices to use theorem 2.

This concludes the proof of (9).□

6.5 The distributed termination convention

According to the original convention of [H4] a guard fails if either its boolean part evaluates to false or its communication part addresses a process which has terminated. For the simplicity sake we have not taken here into account the second possibility of a failure. The above convention when applied to repetitive commands is usually called a *distributed termination convention* (DTC). This convention is taken care of in the proof system of [AFR].

We also mention here a result of Apt and Francez [AF] which states that there exists a transformation T that transforms a parallel program P which uses the above convention into an equivalent parallel program T(P) which uses the convention of section 6.1. An equivalence is understood here in the sense of satisfying the same correctness formulae. Thus to prove correctness of a program P in the presence of the DTC it suffices to prove correctness of T(P) in the sense of this paper.

REFERENCES

[A1] APT, K.R., Ten years of Hoare's logic, a survey, part I, ACM TOPLAS 3, No. 4, pp. 431–483, 1981.

[A2] APT, K.R., Formal justification of a proof system for Communicating Sequential Processes, Journal ACM 30, No. 1, pp. 197–216, 1983.

[A3] APT, K.R., Ten years of Hoare's logic, a survey – nondeterminism, part II, Theoretical Computer Science 28, pp. 83–109, 1984.

[A4] APT, K.R., A static analysis of CSP programs, in : Logics of Programs, Proceedings 1983, Lecture Notes in Computer Science 164, Springer Verlag, New York, pp. 1–17, 1984.

[AF] APT, K.R., and FRANCEZ, N. Modeling the distributed termination of CSP, ACM TOPLAS, 6, No.3, pp. 370–379, 1984.

[AFR] APT, K.R., FRANCEZ, N. and DE ROEVER, W.P., A proof system for communicating sequential processes, ACM TOPLAS 2, No. 3, pp. 359–385, 1980.

[APS] APT, K.R., PNUELI, A. and STAVI, J., Fair termination revisited with delay, in : Proc. 2nd Conference FCT and TCS, pp. 146–170, Bangalore, 1982. (Also to appear in Theoretical Computer Science).

[CC] COUSOT, P., and COUSOT, R., Semantic analysis of communicating sequential processes, in : Proceedings ICALP 80, Lecture Notes in Computer Science 85, Springer Verlag, New York, pp. 119 – 133, 1980.

[D1] DIJKSTRA, E.W., Guarded commands, nondeterminacy and formal derivation of programs, Communications ACM 18, No. 8, pp. 453–457, 1975.

[D2] DIJKSTRA, E.W., A Discipline of Programming, Prentice Hall, Englewood Cliffs, 1976.

[G] GORELICK, G.A., A complete axiomatization system for proving assertions about recursive and nonrecursive programs, Tech. Rep. 76, Dep. Computer Science, University of Toronto, 1975.

[H1] HOARE, C.A.R., An axiomatic basis for computer programming, Communications ACM 12, No. 10, pp. 576-580, 583, 1969.

[H2] HOARE, C.A.R., Procedures and parameters : an axiomatic approach, in : Lecture Notes in Mathematics, vol. 188 : Semantics of Algorithmic Languages, Springer Verlag, New York, pp. 102-116, 1971.

[H3] HOARE, C.A.R., Towards a theory of parallel programming, in : Operating Systems Techniques (C.A.R. Hoare, R.H. Perrot, eds.), New York, Academic Press, 1972.

[H4] HOARE, C.A.R., Communicating Sequential Processes, Communications ACM 21, No. 8, pp. 666-677, 1978.

[LS] LAMPORT, L. and SCHNEIDER, F.B., The "Hoare Logic" of CSP, and all that, ACM TOPLAS 6, No. 2, pp. 281-296, 1984.

[LG] LEVIN, G. and GRIES, D., A proof technique for communicating sequential processes, Acta Informatica 15, No. 3, pp. 281-302, 1981.

[MP] MANNA, Z. and PNUELI, A., How to cook a temporal proof system for your pet language, in : Proceedings 10th Annual ACM Symp. on Principles of Progr. Lang., pp. 141-154, 1983.

[MC] MISRA, J. and CHANDY, K.M., Proofs of Networks of Processes, IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, pp. 417-426, 1981.

[OG1] OWICKI, S. and GRIES, D., Verifying properties of parallel programs : an axiomatic approach, Communications ACM 19, No. 5, pp. 279-285, 1976.

[OG2] OWICKI, S. and GRIES, D., An axiomatic technique for parallel programs, Acta Informatica, pp. 319-340, 1976.

[S] SOUNDARARAJAN, N., Correctness proofs of CSP Programs, in : Proc. 1st Conference FCT and TCS, pp. 135-142, Bangalore, 1981 (also in : Theoretical Computer Science 24, No. 2, pp. 131-142, 1983).

[T] TAYLOR, R.N., A general purpose algorithm for analyzing concurrent programs, Communications ACM 26, No. 5, pp. 362-376, 1983.