RC 10969 (#49271) 2/5/85
Computer Science 56 pages

Proving Correctness of Concurrent Programs - A Quick Introduction[1]

Krzysztof R. Apt[2]

L.I.T.P., Université Paris 7,
2 Place Jussieu,
75251 Paris, France.

Typed by Barbara J. White and Nancy Perry

Abstract: A systematic presentation of Hoare type approach to proving correctness of concurrent programs is provided. Various proof systems for nondeterministic programs, disjoint parallel programs, parallel programs with shared variables and synchronization constructs are studied. Special emphasis is put on the issue of fairness.

[1] This paper will appear in: Current Trends in Theoretical Computer Science (E. Börger, ed.), Computer Science Press.

[2] Current address: IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

1. INTRODUCTION

The aim of this paper is to explain a particular approach to correctness of concurrent programs. Design of concurrent programs is a difficult art and construction of correctness proofs of concurrent programs is an equally nontrivial task. To better understand the issues at stake, let us study a very simple example program.

1.1. An example of a concurrent program

Consider the following simple problem.

Problem. Write a program that finds a zero of a function f from integers into integers.

We wish to explore the fact that a search for positive and non-positive zero's can be done in parallel.

Solution 1

Consider the following program S_1:

$$
\begin{array}{lll}
S_1 \equiv & \text{found} := \text{false}; x := 0; & \\
& \textbf{while} \neg \text{found} \ \textbf{do} & x := x + 1; \\
& & \text{found} := f(x) = 0 \\
& \textbf{od}. &
\end{array}
$$

Then S_1 stops when a positive zero of f is found. Similarly, the following program S_2 stops when a non-positive zero of f is found:

$$
\begin{array}{lll}
S_2 \equiv & \text{found} := \text{false}; x := 0; & \\
& \textbf{while} \neg \text{found} \ \textbf{do} & y := y - 1; \\
& & \text{found} := f(y) = 0 \\
& \textbf{od}. &
\end{array}
$$

Thus the program $[S_1 \| S_2]$, the parallel composition of S_1 and S_2, stops when a zero of f is found and is a solution to the problem.

Unfortunately, this is not always the case. Imagine the following scenario. Let f have only one zero, a positive one. Consider now an execution of $[S_1 \| S_2]$ in which initially only its first component is activated until it terminates once the zero of f is found. At this moment the second

component is activated, found is reset to false and since no other zero's of f exist found will never be reset to true. In other words, this execution of $[S_1 \| S_2]$ will never terminate.

Obviously our mistake consisted of initializing found to false twice - once in each component. A straightforward fix-up consists of initializing found only once, outside the parallel composition. This brings us to the following solution.

Solution 2

Let

$S_1 \equiv$ $x := 0$;
while $\neg$ found **do** $x := x + 1$;
found: $= f(x) = 0$
od

and

$S_2 \equiv$ $y := 0$;
while $\neg$ found **do** $y := y - 1$;
found: $= f(y) = 0$
od.

Then

$$S \equiv \text{found: } = \text{false};[S_1 \| S_2]$$

is a solution to the problem.

But is it actually? Suppose once again that f has exactly one zero, a positive one and consider an execution of S in which only its first component is activated until found is set to true upon finding the zero of f. Suppose that at this moment the second component is activated until found is reset to false. Now, since no other zero's of f exist, found will never be reset to true and this execution of S will never terminate. Thus the above solution is incorrect.

What went wrong here? A close inspection of the scenario just presented reveals that the problem arose due to the fact that found could be reset to false once it was already true. In this way the information that a zero of f was found got lost.

One way of correcting this mistake is by ensuring that found is never reset to false inside of the parallel composition. To this purpose it is sufficient to replace the unconditional assignment

$$\text{found}: = f(x) = 0$$

by the conditional one:

$$\textbf{if } f(x) = 0 \textbf{ then found}: = \textbf{true fi}$$

and similarly with the other assignment to found.

In such a way we obtain the following solution.

Solution 3

Let

$$\begin{aligned} S_1 \equiv\ & x: = 0; \\ & \textbf{while} \neg \text{ found } \textbf{do } x: = x + 1; \\ & \qquad \textbf{if } f(x) = 0 \textbf{ then found}: = \textbf{true fi} \\ & \qquad\qquad \textbf{od} \end{aligned}$$

and

$$\begin{aligned} S_2 \equiv\ & y: = 0; \\ & \textbf{while} \neg \text{ found } \textbf{do } y: = y - 1; \\ & \qquad \textbf{if } f(y) = 0 \textbf{ then found}: = \textbf{true fi} \\ & \qquad\qquad \textbf{od.} \end{aligned}$$

Then

$$S \equiv \text{found}: = \text{false};\ [S_1 \parallel S_2]$$

is a solution to the problem.

But is it really? Suppose that f has only positive zero's and consider an execution of S in which the first component S_1 of the parallel program $[S_1 \parallel S_2]$ is never activated. Then this execution will never terminate even though f has a zero.

The above scenario is a debatable one. One might object that an execution sequence in which one component of a parallel program is never activated is not a legal one. After all the main reason

for writing parallel programs is that the components can and will be executed in parallel. In more technical terms we should decide whether we adopt the hypothesis of *fairness* which here means that every component of a parallel program is eventually activated. If we adopt it then the last solution is a correct one and the just exhibited execution sequence of S is illegal. If we reject it then the last solution is incorrect as the exhibited execution sequence of S is legal.

We shall now present a solution which is appropriate when the fairness hypothesis is not adopted. Surprisingly, as it will turn out in Section 3.5 the proof of correctness of the above solution under the assumption of fairness will amount to the proof of correctness of the final solution we shall present here under no assumption of fairness.

The solution consists of building into the above program S an abstract *scheduler* which will ensure that each component of the parallel program is eventually executed. To this purpose we need a new programming construct - **await** B **then** R **end** allowing to temporarily suspend an execution of a component. Informally, a component of a parallel program executes an await-statement if the Boolean expression B evaluates to **true**. The statement R is then executed as an indivisible action, i.e. an action which cannot be interrupted by an activation of another component.

With each parallel component we associate a *priority* variable which is used by the scheduler. The scheduler activates the component with the lowest priority and updates the priorities of all the components. A component with the higher priority is suspended until its priority becomes the lowest.

Summarizing, this solution has the following form:

Solution 4

Let

$S_1 \equiv x := 0;$
while ¬ found **do**
await $z_1 \leq z_2$ **then**
$z_1 := ?;\ z_2 := z_2 - 1$
end;
$x := x + 1;$
if $f(x) = 0$ **then** found : = **true fi**
od

and

$S_2 \equiv y := 0;$
while $\neg$ found **do**
await $z_2 \leq z_1$ **then**
$z_2 := ?;\ z_1 := z_1 - 1$
end;
$y := y - 1;$
if $f(x) = 0$ **then** found $:=$ **true fi**
od.

Then

$$S \equiv z_1 := ?;\ z_2 := ?;\ \text{found} := \text{false};[S_1 \parallel S_2]$$

is a solution to the problem when the fairness hypothesis is not adopted.

$z := ?$ stands here for a *random assignment* which denotes an assignment of an arbitrary non-negative integer to the variable z. (If you feel uncomfortable about this instruction you can replace it everywhere in the above program by an assignment to a fixed non-negative integer, say 0.) The scheduling parts are framed. Note that they are activated only once per a loop round. An infinite execution sequence in which only one parallel component is activated is now impossible because of the continuous decrease of the priority variable of the other component and the continuous resetting of the component priority variable to a non-negative value.

But is S really a solution to the problem? Consider an execution sequence of S in which a zero is found by the first component at the moment when $z_1 \leq z_2$ holds and the second component is suspended in front of the await-statement. Then the first component will terminate and the second component will remain suspended forever. Thus this execution sequence will terminate but in an improper way. This might be still considered as unharmful because after all the program eventually stops as desired when a zero is found. However, such a solution is unacceptable in the context in which the above program is to be followed by another one and thus activated upon *proper* termination of the first one.

In other words, we are interested here in solutions free from *deadlock* - a situation in which no progress is possible even though not all parallel components have properly terminated.

A deadlock free solution to our problem can be obtained by an appropriate modification of the last solution. To this purpose it is enough to build into the program a *signaling scheme* that will release one component in case of the (proper) termination of the other one. We thus introduce two new variables, called end_1 and end_2 which will indicate whether the first resp. the second component has terminated. A suspended component will always be released when the other component has terminated due to an additional disjunct in the **await**-condition.

Summarizing, this solution has the following form:

Solution 5

Let

$S_1 \equiv x := 0;$
while ¬ found **do**
await $z_1 \leq z_2$ [∨ end_2] **then**
$z_1 := ?;\ z_2 := z_2 - 1$
end;
$x := x + 1;$
if $f(x) = 0$ **then** found := **true fi**
od;
[$end_1 :=$ **true**]

and

$S_2 \equiv y := 1;$
while ¬ found **do**
await $z_2 \leq z_1$ [∨ end_1] **then**
$z_2 := ?;\ z_1 := z_1 - 1$
end;
$y := y - 1;$
if $f(y) = 0$ **then** found := **true fi**
od;
[$end_2 :=$ **true.**]

Then

$$S \equiv z_1 := ?;\ z_2 := ?;\ \mathrm{end}_1 := \mathrm{false};\ \mathrm{end}_2 := \mathrm{false};\ \mathrm{found} := \mathrm{false};\ [S_1 \| S_2]$$

is a deadlock-free solution to the problem when the fairness hypothesis is not adopted.

The signaling scheme built into the previous solution is framed.

We assure the reader that the above solution is correct. It is by no means an efficient solution - for example, the assignments to the priority variables can be performed outside the await-statements. On the other hand, the last two transformations are special cases of the transformations we study in this paper and the above discussion should facilitate their understanding.

1.2. The correctness problem

We hope to have convinced the reader that the design and correctness of concurrent programs is not a simple issue. The problem we discussed in the previous subsection seemed to be completely trivial and yet several, sometime subtle errors crept in. It should be clear that an informal justification of the correctness of concurrent programs is not sufficient. After all we produced in the previous subsection at least two incorrect correctness proofs.

There has been a number of formal approaches to correctness of concurrent programs. Before we briefly discuss them we should perhaps first agree what program properties we actually wish to prove.

In the case of sequential programs, i.e. those in which a control resides at each moment in only one control point, these usually are:

1. Delivering correct results.

   For example a sorting program should indeed sort the input.

2. Termination.

   For example a sorting program should always terminate.

3. Lack of failures.

   For example there should be no division by zero, no overflow etc..

In the case of concurrent programs, i.e. those in which a control can reside at the same time in several control points, as observed before we are additionally interested in establishing:

4. Deadlock freedom.

5. Correctness under the fairness assumption.

It should be stressed that this list is by no means exhaustive. In fact, there is an important class of *continuously operating* concurrent programs, i.e. those which never terminate, that is left out of considerations in this paper. Mutual exclusion algorithms belong to this class of programs. For this type of program different properties than termination are of relevance.

There has been a number of formal approaches to program correctness which were proposed and used in the literature. The most common of them is that based on an *operational reasoning*. It consists of an analysis in terms of the execution sequences of the given program. To this purpose an informal understanding of the program semantics is used. While this analysis is usually successful in the case of sequential programs it is much less so in the case of concurrent programs. The number of possible execution sequences is then most often forbiddingly large and it is all too easy to overlook a possible execution sequence.

A different approach is that based on an *axiomatic reasoning*. According to this approach in order to prove that a program S satisfies a property P we should find a proof system T with a language L(T) such that

i) T is sound for the program S, i.e. any theorem of T is true for S,

ii) the property of P can be expressed in L(T) by a formula φ,

iii) φ can be proved in T.

(Strictly speaking, the operational reasoning is a special, degenerate case of the axiomatic reasoning in which the property P constitutes the only axiom of the proof system. Then the properties ii) and iii) are trivial and the whole burden of the proof lies on the property i).)

This approach started with the seminar paper of Hoare [H1] where an axiomatic proof has been proposed to prove correctness of simple while-programs. His approach, often called Hoare's logic, has received a great deal of attention since then. An interested reader may wish to consult Apt [A1]

for a survey of various Hoare style proof systems proposed for several programming constructs used in the imperative programming languages. In 1976 this approach has been extended to the case of concurrent programs by Owicki and Gries [OG1, OG2] and Lamport [L1].

This aim of this paper is to provide a systematic exposition of this method applied to various types of concurrent programs.

It should be stressed here that there are other approaches to the correctness of concurrent programs. They are not discussed in this paper. Perhaps the most important among them is the one started by Pnueli [P] and further developed in Manna and Pnueli [MP1, MPP2, MP3] which is based on *temporal logic*. This approach allows to study more complicated properties than those listed above and is particularly useful when dealing with continuously operating concurrent programs.

1.3. Preliminaries

Throughout the paper we fix an arbitrary assertion language containing two Boolean constants **true** and **false**. Its formulas are called *assertions* and denoted by the letters p, q, r. Quantifier free formulas are called *Boolean expressions* and are denoted by the letter B.

We assume that the variables are of the type **integer** or **Boolean**. Variables are denoted by the letters x, y, z, u, a, b. Their type is fixed by the context in which they are used. Expressions are denoted by the letters s, t. p[t/u] stands for a *substitution* of t for all free occurrences of u in p. 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.

The programs are executed over a domain consisting of all integers and {**true**, **false**} with the usual operations available. By a (proper) *state* we mean a function assigning to all variables a value from the domain. States are denoted by the letters σ, τ. The notions of a value of an expression t in a state σ (written as $\sigma(t)$) and truth of an assertion p in a state σ (written as $\models p(\sigma)$) and truth of an assertion are defined as usual. By $\sigma[d/x]$, where d is a value from the domain, we mean the state obtained from σ by assigning to x the value d and retaining the values of other variables.

We allow three special states: $\perp$ reporting nontermination of a program, **fail** reporting a failure in an execution of a program and Δ reporting deadlock in an execution of a program. We have by

definition $\models p(\perp)$, $\models p(\text{fail})$ and $\models p(\Delta)$ for all formulas p. We define [p] to be the set of all states σ which satisfy p, i.e. such that $\models p(\sigma)$ holds.

We say that $\{p\}S\{q\}$ is true in the sense of *partial correctness* (and write $\models \{p\}S\{q\}$) 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 in the sense of *total correctness* (and write $\models_{tot}\{p\}S\{q\}$) if it is true in the sense of partial correctness and moreover, all computations of S starting in a state satisfying p properly terminate. We shall consider in this paper various other notions of program correctness.

A program semantics $\mathcal{M}_n[[S]]$ of a program S is a mapping from the set Σ of proper states into the subsets of $\Sigma \cup \{\perp, \text{fail}, \Delta\}$. Each program semantics $\mathcal{M}_n[[\,\cdot\,]]$ fixes a notion of program correctness defined by

$$\models_n \{p\}S\{q\} \text{ iff } \mathcal{M}_n[[S]]([p]) \subseteq [q]$$

where

$$\mathcal{M}_n[[S]]([p]) = \bigcup_{\sigma \in [p]} \mathcal{M}_n[[S]](\sigma).$$

We shall consider here various semantics, among others partial correctness semantics $\mathcal{M}[[\,\cdot\,]]$ and total correctness semantics $\mathcal{M}_{tot}[[\,\cdot\,]]$ related to the notions of partial and total correctness, respectively.

All proof systems discussed in this paper are geared towards proving correctness formulas in various senses. All considered proof systems are *sound* in the sense that every provable correctness formula is true in an appropriate sense. Soundness proofs of the discussed proof systems are omitted but relevant lemmas are often exhibited.

2. NONDETERMINISTIC PROGRAMS

To make this paper self-contained, we review in this section the correctness of nondeterministic programs in the style of Dijkstra [D1, D2]. We concentrate here on the issue of fairness as the results concerning it will be of relevance in the following section. More extensive treatment of the subject can be found in [A4].

We allow as atomic actions the skip statement and assignment statement. Programs are built using the composition operator ";" and allowing

- the *alternative command*

$$[\ \mathop{\Box}_{i=1}^{m} B_i \rightarrow S_i]$$

and

- the *repetitive command*

$$*[\ \mathop{\Box}_{i=1}^{m} B_i \rightarrow S_i]$$

where B_i are Boolean expressions (called *guards*) and S_i are programs.

Nondeterministic programs form a good starting point to study parallel programs. This is due to the fact that every parallel program $[S_1 \| \ldots \| S_n]$ is equivalent in an appropriate sense to the nondeterministic program

$$*[\mathop{\Box}_{i=1}^{n} \text{enabled } (S_i) \rightarrow \text{execute } S_i \text{ one step}]$$

where the definition of executing S_i one step depends on the way the atomic actions are defined in the context of parallel composition.

The nondeterministic programs in which only the alternative commands of the form $[B \rightarrow S_1 \ \Box \ \neg B \rightarrow S_2]$ and repetitive commands of the form $*[B \rightarrow S]$ are used and are called deterministic programs or while-programs. Note that the above commands correspond to the customary commands **if** B **then** S_1 **else** S_2 **fi** and **while** B **do** S **od**, respectively. while-programs

are deterministic in the sense that they produce at most one final state.

We shall also use other constructs which can be straightforwardly defined in terms of the alternative and repetitive commands.

2.1 Semantics

We recall here a simple semantics of nondeterministic programs due to Hennessy and Plotkin [HP]. This semantics is based on the consideration of a transition relation '$\rightarrow$' between pairs $\langle S,\sigma\rangle$ consisting of a program S and a state σ. The intuitive meaning of the relation

$$\langle S_1,\sigma\rangle \rightarrow \langle S_2,\tau\rangle$$

is the following: executing S_1 one step in a state σ can lead (nondeterministically) to a state τ with S_2 being remainder of S_1 still to be executed. It is convenient to assume the empty program E. Then S_2 is E if the considered step of S_1 leads to state τ with S_1 properly or improperly terminated. We assume that, for any S, E; S = S; E = S.

We define the above relation by the following clauses where $\sigma \neq \perp$, fail, Δ :

(i) $\langle \text{skip}, \sigma\rangle \rightarrow \langle E, \sigma\rangle$,

(ii) $\langle x := t,\sigma\rangle \rightarrow \langle E,\sigma[\sigma(t)/x]\rangle$,

(iii) $\langle [\underset{i=1}{\overset{m}{\Box}} B_i \rightarrow S_i],\sigma\rangle \rightarrow \langle S_i,\sigma\rangle$ if $\models B_i(\sigma)$,

(iv) $\langle [\underset{i=1}{\overset{m}{\Box}} B_i \rightarrow S_i],\sigma\rangle \rightarrow \langle E, \text{fail}\rangle$ if $\models \bigwedge_{i=1}^{m} \neg B_i(\sigma)$,

(v) $\langle *[\underset{i=1}{\overset{m}{\Box}} B_i \rightarrow S_i],\sigma\rangle \rightarrow \langle S_i; *[\underset{i=1}{\overset{m}{\Box}} B_i \rightarrow S_i],\sigma\rangle$ if $\models B_i(\sigma)$,

(vi) $\langle *[\underset{i=1}{\overset{m}{\Box}} B_i \rightarrow S_i],\sigma\rangle \rightarrow \langle E,\sigma\rangle$ if $\models \bigwedge_{i=1}^{m} \neg B_i(\sigma)$,

(vii) if $\langle S_1,\sigma\rangle \rightarrow \langle S_2,\tau\rangle$ then $\langle S_1; S,\sigma\rangle \rightarrow \langle S_2; S,\tau\rangle$.

Let $\rightarrow^*$ stand for the transitive, reflexive closure of $\rightarrow$.

We now introduce the following definitions.

Definition 2.1.

(i) We say that S can *diverge* from σ if there exists an infinite sequence $<S_i,\sigma_i>$ $(i = 0,1,\ldots)$ such that

$$<S,\sigma> = <S_0,\sigma_0> \rightarrow <S_1,\sigma_1> \rightarrow \ldots .$$

(ii) We say that S can *fail* from σ if for some S_1

$$<S,\sigma> \rightarrow^* <S_1, \text{fail}>$$

(iii) A finite or infinite sequence $<S_i,\sigma_i>$ $(i = 0,1,\ldots)$ such that

$$<S,\sigma> = <S_0,\sigma_0> \rightarrow <S_1,\sigma_1> \rightarrow \ldots$$

which cannot be extended is called a *computation of* S *starting in* σ.

We now define two types of semantics for the nondeterministic programs by putting

$$\mathcal{M}[[S]](\sigma) = \{\tau : <S,\sigma> \rightarrow^* <E,\tau>\}$$

and

$$\mathcal{M}_{tot}[[S]](\sigma) = \mathcal{M}[[S]](\sigma) \cup \{\perp : \text{S can diverge from } \sigma\} \cup \{\text{fail} : \text{S can fail from } \sigma\}.$$

Next, we provide four different proof systems to prove different types of correctness of nondeterministic programs.

2.2 Partial correctness

The following proof system allows proof of partial correctness of the nondeterministic

programs:

AXIOM 1: SKIP AXIOM

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

AXIOM 2: ASSIGNMENT AXIOM

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

RULE 3: COMPOSITION RULE

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

RULE 4: ALTERNATIVE COMMAND RULE

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ i=1,\ldots,m}{\{p\}\ [\underset{i=1}{\overset{m}{\square}} B_i \rightarrow S_i]\ \{q\}}\ .$$

RULE 5: REPETITIVE COMMAND RULE

$$\frac{\{p \wedge B_i\}\ S_i\ \{q\},\ i = 1,\ldots,m}{\{p\}\ *\ [\underset{i=1}{\overset{m}{\square}} B_i \rightarrow S_i]\ \{p \wedge \overset{m}{\underset{i=1}{\bigwedge}} \neg B_i\}}$$

RULE 6: CONSEQUENCE RULE

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

We call this proof system PC and write $\vdash_{PC} \varphi$ to denote the fact that the correctness formula φ can be proved in PC using for the consequence rule all true assertions as axioms. We use an analogous notation for the other proof systems.

2.3 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

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

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

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

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

We call the resulting proof system TC.

2.4 Weak fairness

We observed in the introduction that fairness is a natural assumption concerning parallel programs. The modeling of parallel programs by the nondeterministic ones given at the beginning of this section is appropriate only if the assumption of fairness is adopted for

nondeterministic programs, as well. The exact form of this assumption depends on the type of parallel programs we wish to simulate.

There are at least two natural fairness assumptions which can be adopted in the case of nondeterministic programs. The first of them is *weak fairness.*

Definition 2.2

Let a computation ξ of a nondeterministic program S be given.

i) We say that a guard B of S is *enabled* if it evaluates to true at the moment the control in the program is just before it.

ii) We say that ξ is *weakly unfair* if it is infinite and there exists a guard B of S which from a certain moment on is continuously enabled and never selected for execution.

iii) We say that ξ is *weakly fair* if it is not weakly unfair.

For example, the only infinite computation of the program

b:= true;

*[b → skip □ ¬b → b:= false]

is weakly unfair, i.e., it is not weakly fair. On the other hand the infinite computation of the program

a := true ;

* [a → b: = ¬b □ b → a: = false]

in which only the first guard is selected is weakly fair since the second guard is not continuously true in it.

We now define the weakly fair semantics of nondeterministic programs by putting

$$\mathcal{M}_{\text{wfair}}\,[[S]]\,(\sigma) =$$

$$\mathcal{M}[[S]](\sigma) \cup \{\perp : \text{S can diverge from } \sigma \text{ by a weakly fair computation}\}$$
$$\cup \{\text{fail} : \text{S can fail from } \sigma\}.$$

We say that {p} S {q} *holds under the assumption of weak fairness* ($\models_{wfair}$ {p} S {q} in short) if every weakly fair computation of S starting in a state satisfying p properly terminates in a state satisfying q.

We now present a proof system appropriate for proving correctness of nondeterministic programs under the assumption of weak fairness. We follow here the approach of Apt and Olderog [AO] and of Apt, Pnueli and Stavi [APS].

Let S be a given nondeterministic program. We proceed by the following steps.

Step 1 Transform S into a program $T_{wfair}(S)$ that generates exactly all weakly fair computations of S.

By Step 1, $\models_{wfair}$ {p} S {q} iff $\models_{tot}$ {p} $T_{wfair}(S)$ {q} (under certain restrictions).

Step 2. Find a proof system allowing to prove total correctness of the programs of the form $T_{wfair}(S)$.

Step 3. Transform the proof from Step 2 into a proof system allowing to prove $\models_{wfair}$ {p} S {q} directly without reference to $T_{wfair}(S)$.

ad Step 1

Consider the following program

x:= 0 ; b:= true ;

*[b → x:= x + 1 □ b → b:= false].

Under the assumption of weak fairness this program always terminates but any non-negative integer can be the final value of x. It is known (see essentially Dijkstra [D2]) that this effect cannot be achieved by the nondeterministic programs studied here.

To control this unbounded nondeterminism we allow in $T_{wfair}(S)$ the *random assignment*

x:= ?

which sets x to an arbitrary non-negative integer and thus has the following semantics:

$< x:= ? , \sigma > \rightarrow <E, \sigma[d/x]>$ for every $0 \leq d$.

Observe that under the assumption of weak fairness the above program is equivalent to

$$x := ?\ ;\ b := \text{false}.$$

To make the approach easier to follow, consider first a special case.

Case 1. S is of the form $*[\ \overset{m}{\underset{i=1}{\Box}} B_i \to S_i]$ where each S_i is deterministic. Following Apt and Olderog [AO] we define $T'_{wfair}(S)$ as follows:

$$T'_{wfair}(S) \equiv z_1 := ?;\ldots; z_m := ?;$$
$$*[\ \overset{m}{\underset{i=1}{\Box}} B_i \wedge turn = i \to S_i\ ;\ z_i := ?;$$
$$\textbf{for } j \neq i \textbf{ do}$$
$$[B_j \to z_j := z_j - 1 \Box \neg B_j \to z_j := ?]$$
$$\textbf{od}$$
$$]$$

where $turn = i \equiv i = \min\{j \mid z_j = \min\{z_k \mid B_k\}_{k=1,\ldots,m}\}$ and the variables $z_1,\ldots,z_m$ do not occur in S.

The variables $z_1,\ldots,z_m$ can be interpreted as *priorities* assigned to the subprograms $S_1,\ldots,S_m$, respectively. The addition of the conjuncts $turn = i$ to the guards B_i makes the guards deterministic (i.e., mutually exclusive). At each moment when the control is at the main loop entry a guard with the smallest priority is selected for execution. After the execution of the corresponding subprogram S_i the priorities are recomputed.

The exact relation between S and $T'_{wfair}(S)$ is expressed by the following lemma where $Z = \{z_1,\ldots,z_m\}$.

Lemma 2.1. For all states σ $\mathcal{M}_{wfair}[[S]](\sigma) = \mathcal{M}_{tot}[[T'_{wfair}(S)]](\sigma) \mod Z$. □

The notation "mod Z" suggests that the states produced by the programs agree on all variables except those in Z.

Corollary 1. For all assertions p and q which do not contain $z_1,\ldots,z_m$ as free variables

$$\models_{wfair} \{p\}\ S\ \{q\} \text{ iff } \models_{tot} \{p\}\ T'_{wfair}(S)\ \{q\}.$$ □

Consider now the general case.

Case 2. S is an arbitrary nondeterministic program.

We proceed by the following successive steps (see Apt, Pnueli and Stavi [APS]):

1°. Replace each subprogram $*[\prod_{i=1}^{m} B_i \to S_i]$ of S by

$$*[\bigvee_{i=1}^{m} B_i \to [\prod_{i=1}^{m} B_i \to S_i]]$$

2°. Replace each subprogram $[\prod_{i=1}^{m} B_i \to S_i]$ of S by the following subprogram:

$$\text{for } j:=1 \text{ to } m \text{ do } [B_j \to z_j := z_j - 1 \ \square \ \neg B_j \to z_j := ?] \text{ od};$$
$$[\prod_{i=1}^{m} B_i \wedge \bar{z} \geq 0 \to z_i := ?; S_i].$$

where $\bar{z} \geq 0$ stands for $z_1 \geq 0 \wedge \ldots \wedge z_m \geq 0$.

3°. Rename all variables $z_1,\ldots,z_m$ appropriately so that each alternative command has its "own" set of these variables.

Call the resulting program $T_{wfair}(S)$.

This transformation is not a straightforward generalization of the first transformation. First of all the guards in the transformed program are not deterministic. Secondly, this transformation introduces failures which will result once a priority variable is decreased below zero. More formally the following holds.

Lemma 2.2. For all states σ

$$\mathcal{M}_{wfair}[[S]](\sigma) = \mathcal{M}[[T_{wfair}(S)]](\sigma) - \{fail\} \mod Z. \quad \square$$

The presence of failures allows us only to conclude the following.

Corollary 2.2. For all assertions p and q which do not contain $z_1,\ldots,z_m$ as free variables and all programs S

$$\models_{wfair} \{p\}\ S\ \{q\} \text{ iff } \models_{wtot} \{p\}\ T_{wfair}(S)\ \{q\}$$
$$\text{and } \forall\sigma\ [\models p(\sigma) \Rightarrow S \text{ does not fail from } \sigma]. \quad \square$$

Here the subscript "wtot" refers to weak total correctness - a notion obtained by disregarding possible failures in the definition of total correctness.

ad Step 2.

To prove weak total correctness of the programs of the form T_{wfair} (S) we have to take care of the random assignments. To this purpose we introduce the following axiom.

AXIOM 9 : RANDOM ASSIGNMENT AXIOM

$$\{ \forall x \geq 0\ p\}\ x := ?\ \{p\}.$$

However, it is not sufficient to add the above axiom to the proof system defined in Section 2.3. The reason is that in the presence of random assignments some programs always terminate but the actual number of steps does not depend on the initial state and is unbounded. An example of such a program is

$$\begin{aligned} S \equiv\ & * [b \wedge 0 < y \rightarrow \\ & \quad [b \rightarrow y := ?\ ;\ b := \text{false} \\ & \quad \Box \neg b \rightarrow y := y-1 \\ & \quad] \\ &] \end{aligned}$$

To prove total correctness of such programs the repetitive command rule II (rule 8) is not sufficient. An appropriate modification is obtained by allowing the parameter variable to range over ordinals instead of natural numbers. We thus adopt the following rule instead of rule 8:

RULE 10: REPETITIVE COMMAND RULE III

$$\frac{\{p(\alpha) \wedge B_i\}\ S_i\ \{\exists \beta < \alpha\ p(\beta)\},\ i=1,\ldots,m}{\{\exists \alpha p(\alpha)\} * [\ \mathop{\Box}_{i=1}^{m} B_i \rightarrow S_i] \{\exists \alpha p(\alpha) \wedge \mathop{\bigwedge}_{i=1}^{m} \neg B_i\}}$$

where $p(\alpha)$ is an assertion with a free variable α which does not appear in the programs and ranges over ordinals.

ad Step 3

Corollary 2 states that in order to prove total correctness of a nondeterministic program S under the assumption of weak fairness it essentially suffices to prove weak total correctness of $T_{wfair}(S)$. But instead of proving the correctness of $T_{wfair}(S)$ directly we rather transform the proof into a direct proof of S by "absorbing" the transformation into the assertions of existing rules. Finally, we take care of the problem of failures in the same way as in Section 3.3, i.e., by adding a new premise to the hypotheses of the (new) alternative command rule.

This procedure leads to the following new proof rules for alternative and repetitive commands.

RULE 11: WFAIR ALTERNATIVE COMMAND RULE

$$p \rightarrow \bigvee_{i=1}^{m} B_i,$$

$$\{\exists \tilde{z}_m,\ldots,\tilde{z}_m \exists z_i p\ [\text{if } B_j \text{ then } z_j + 1 \text{ else } \tilde{z}_j \text{ fi}/z_j]_{j\neq i} \wedge B_i \wedge \bar{z} \geq 0\}$$

$$S_i$$

$$\{q\}\ ,\ i = 1,\ldots,m$$

$$\{p\}\ [\ \square_{i=1}^{m} B_i \rightarrow S_i]\ \{q\}$$

RULE 12: WFAIR REPETITIVE COMMAND RULE

$$\{\exists \tilde{z}_1,\ldots,\tilde{z}_n \exists z_i\ p(\alpha)\ [\text{if } B_j \text{ then } z_j + 1 \text{ else } \tilde{z}_j \text{ fi}/z_j]_{j\neq i} \wedge B_i \wedge \bar{z} \geq 0\}$$

$$S_i$$

$$\{\exists \beta < \alpha\ p(\beta)\}\ ,\ i = 1,\ldots,m$$

$$\{\exists \alpha\ p(\alpha)\}\ ^*\ [\ \underset{i=1}{\overset{m}{\Box}} B_1 \rightarrow S_i]\ \{\exists \alpha\ p(\alpha) \wedge \bigwedge_{i=1}^{m} \neg B_i\}$$

where $p(\alpha)$ is as in rule 10.

Summarizing, the proof system appropriate for proving total correctness of nondeterministic programs under the weak fairness hypothesis consists of axioms 1.2 and rules 3, 11, 12 and 6. The random assignment axiom is not needed - it was only used to derive the final form of the above two rules.

2.5. Strong fairness

Another natural fairness assumption is that of strong fairness.

Definition 2.3

Let ξ be a computation of a nondeterministic program S.

i) We say that ξ is *strongly unfair* if it is infinite and there exists a guard B of S which from a certain moment on is infinitely often enabled and never selected for execution.

ii) We say that ξ is *strongly fair* if it is not strongly unfair.

Now, the hypothesis of strong fairness can be treated in an analogous way as that of weak fairness. To obtain a transformation realizing strong fairness we simply replace in the corresponding transformations from Step 1 in the previous section the updating of priority

variables $[B_j \rightarrow z_j := z_j - 1 \,\square\, \neg B_j \rightarrow z_j := ?]$ by

$$[B_j \rightarrow z_j := z_j - 1 \,\square\, \neg B_j \rightarrow \text{skip}]$$

The remaining steps in the development of the proof system for strong fairness are the same as before and omitted.

3. PARALLEL PROGRAMS

We now consider parallel programs. These are programs of the form

$$S \equiv S_0 ; [S_1 \| \ldots \| S_n]$$

where S_0 - called an *initial part* - consists of a (possibly empty) sequence of assignments and $S_1,\ldots,S_n$ - called the *components of the parallel program* - are deterministic programs. For a moment we disallow synchronization constructs in the component programs.

3.1 Semantics

Their semantics are obtained by augmenting the list of clauses given in Section 2.1 by the following one handling the parallel composition.

viii) if $\langle R_i,\sigma \rangle \rightarrow \langle R'_i,\tau\rangle$ then

$$\langle [R_1 \| \ldots \| R_n], \sigma\rangle \rightarrow \langle [R_1 \| \ldots \| R_{i-1} \| R'_i \| R_{i+1} \| \ldots \| R_n], \tau\rangle$$

This leads to two types of semantics of parallel programs - $\mathcal{M}$ and $\mathcal{M}_{tot}$ defined similarly as before but this time with $\langle [\underbrace{E \| \ldots \| E}_{n \text{ times}}] , \tau\rangle$ as the final configuration. We shall also consider a third semantics taking into account the fairness assumption. To this purpose we introduce the following notion.

Definition 3.1

Let ξ be a computation of a parallel program $S \equiv S_0 ; [S_1 \| \ldots \| S_n]$.

i) We say that the component S_i $(1 \leq i \leq n)$ *has terminated in* ξ if for some $R_1,\ldots,R_n$ and τ where $R_i \equiv E$, $\langle [R_1 \| \ldots \| R_n], \tau\rangle$ is an element of ξ.

ii) We say that the component S_i is *active* in the step

$$\langle [R_1 \| \ldots \| R_n], \sigma\rangle \rightarrow \langle [R'_1 \| \ldots \| R'_n], \tau\rangle$$

of ξ if $R_i \not\equiv E$ and $\langle R_i, \sigma\rangle \rightarrow \langle R'_i,\tau\rangle$.

iii) We say that ξ is *unjust* if some component S_i did not terminate in ξ and is only

finitely many times active in ξ.

iv) We say that ξ is *just* if it is not unjust.

We now define the just semantics of parallel programs by putting

$$\mathcal{M}\text{just}\ [[S]]\ (\sigma) = \mathcal{M}\ [[S]]\ (\sigma)$$
$$\cup\ \{\perp\text{: S can diverge from } \sigma \text{ by a just computation}\}$$
$$\cup\ \{\text{fail: S can fail from } \sigma\}$$

This semantics takes care of the fairness assumption mentioned in the introduction. Thus Solution 3 from the introduction is a correct solution to the zero finding problem in the case when the above semantics is adopted.

It should be observed that all three semantics introduced here handle parallelism by reducing it to an arbitrary interleaving of the executions of the components programs - at each step only one component of the program is active. Whether this is a realistic approximation of a truly parallel execution of a parallel program depends on how *atomic actions* of parallel programs are defined. By an atomic action we mean a statement within a component whose execution cannot be interrupted by the activation of another component.

The usual requirement concerning the parallel execution is that *concurrent reading and writing* is disallowed, i.e., an execution of an assignment to a variable x cannot be interrupted by any other action referring to x. This requirement implies that assignments and evaluations of the guards are considered as atomic actions. One can prove that for this choice of atomic actions the above defined semantics are indeed equivalent to appropriately defined "parallel" semantics. Consequently the above semantics are sufficient to analyze the behavior of the parallel programs.

The granularity of interleaving can be increased here by decreasing the size of atomic actions. This can be achieved by allowing only very simple types of assignments and Boolean expressions whose execution blocks other components for a more negligible amount of time.

3.2 Disjoint parallelism

The main difficulty in studying parallel programs lies in the use of *shared variables* A variable is called *shared* if it can be modified within one component of a parallel program and moreover is referred to within another. Then the behavior of the latter component can depend on the behavior of the first one. This can result in a nondeterministic behavior of a parallel program with deterministic components. To see this, consider, for example, the program $x:= 0; [x:= 1 \parallel y:= x]$ in which the final value of y is either 0 or 1 .

Due to these difficulties it is natural to study first parallel programs without shared variables. Such programs are called *disjoint parallel programs* and were originally studied in Hoare [H2].

More formally, let free (S) stand for the set of all variables which occur in S and let change (S) stand for the set of all variables of S which can be modified by it, i.e., which appear on the left hand side of an assignment.

Then $S \equiv S_0 ; [S_1 \parallel ... \parallel S_n]$ is called a *disjoint parallel program* if

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

Thus the program $[x:= z \parallel y:= z]$ is a disjoint parallel program but $[x:= z \parallel y:= x]$ or $[x:= z \parallel x:= y]$ is not.

The following proof rule dealing with disjoint parallel programs was proposed by Hoare in [H2]:

RULE 13: RULE OF DISJOINT PARALLEL COMPOSITION

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

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

Observe that the proviso of the rule is indeed needed. For example, the true premises $\{y = 1\}\ x:= 0\ \{y = 1\}$ and $\{\text{true}\}\ y:= 0\ \{\text{true}\}$ should not lead to the conclusion $\{y = 1\}\ [x:= 0 \parallel y:= 0]\ \{y = 1\}$.

Note the natural connection between the condition imposed on the programs S_i and the condition of the rule.

The above rule is a useful one but it does not suffice to prove all properties of disjoint parallel programs. It is not difficult to prove that the correctness formula $\{x = y\}\ [x:= x + 1 \parallel y:= y + 1]\ \{x = y\}$ cannot be proved in the proof system PC from Section 2.2 augmented by the above rule. Let us see where a possible proof actually breaks down.

We clearly have

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

and

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

so by the rule of disjoint parallel composition

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

Now by the rule of consequence

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

However, we cannot replace the pre-assertion $x = z \wedge y = z$ by $x = y$ since clearly the latter does not imply the former. On the other hand we have

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

so by the composition rule

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

We can now obtain the desired formula by dropping the assignment z:= x. Formally this requires use of a new rule allowing to delete assignments to the so-called auxiliary variables. This brings us to the following definition.

Definition 3.2

Let A be a set of variables of a program S. We call A 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 variable 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 $\neq \emptyset$.

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

Informally, i) states that the auxiliary variables do not affect the control flow of the program and ii) states that they do not affect the data flow of the program.

The following rule was first introduced by Owicki and Gries in [OG1] for the case of arbitrary parallel programs.

RULE 14: RULE OF AUXILIARY VARIABLES

Let A be a set of auxiliary variables of a program S and 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 = $\emptyset$.

Now, using the above rule we can complete the proof of the correctness formula {x = y} [x:= x + 1 ‖ y:= y + 1] {x = y} by dropping the assignment to z .

Adding the last two rules to the proof systems for partial, respectively total correctness we obtain proof systems appropriate for proving partial, respectively total correctness of disjoint parallel programs.

Consider now the problem of justice. It may come out as a surprise that this is not a real issue for disjoint parallel programs. This follows from the following simple lemma.

Lemma 3.1. For all disjoint parallel programs S

$$\mathcal{M}_{tot}\,[[S]] = \mathcal{M}_{just}\,[[S]]$$

holds.

Proof. For any given state σ and a parallel program S we have $\mathcal{M}_{just}\,[[S]]\,(\sigma) \subseteq \mathcal{M}_{tot}\,[[S]](\sigma)$. Consider now the converse implication. The only possible difference can lie in the existence of an infinite computation. So suppose that $\perp \in \mathcal{M}_{tot}\,[[S]](\sigma)$ and let ξ be an infinite computation of S starting in σ.

Due to disjointness of the components, ξ is infinite because of a looping within a single component. Now, if ξ is not just then it can easily be transformed into a just computation by simply activating sufficiently often the components of S which did not terminate in ξ. Insertion of these steps does not affect the behavior of other components due to their disjointness. The resulting computation is an infinite just computation of S starting in ξ. So $\perp \in \mathcal{M}_{just}\,[[S]](\sigma)$. □

This lemma implies that there is no difference in proving total correctness of disjoint parallel programs with or without the assumption of justice.

3.3 Parallel programs with shared variables

Consider now the general case. Then, rule 13 is still sound. However, this time it is completely inadequate. Take, for example, the program $S \equiv [x := x + 1 \,\|\, x := x + 1]$. Then $\{x = 0\}\ S\ \{x = 2\}$ is true. However, due to the restriction of the rule in the case of S x

cannot be referred to in the assertions. A possible use of the rule of auxiliary variables cannot remedy this problem and consequently the correctness formula

$\{x = 0\}\ S\ \{x = 2\}$ cannot be proved in the proof system studied in the previous section.

3.3.1 Partial correctness

To overcome this difficulty we have to find a proof rule for parallel composition which allows references in assertions to the shared variables. We follow here the approach of Owicki and Gries [OG1].

First we introduce the notion of a proof outline. Consider the following proof of (partial) correctness of the gcd program:

$\{x = a \wedge y = b\}$

$\{p\}$

$*\ [x > y \rightarrow \{p \wedge x > y\}$

$x := x - y$

$\{p\}$

$\square\ x < y \rightarrow \{p \wedge x < y\}$

$y := y - x$

$\{p\}$

$]$

$\{x = y \wedge p\}$

$\{x = y = \gcd(a,b)\}$

where

$$p \equiv \gcd(x,y) = \gcd(a,b)\ .$$

The proof is presented here in a special form, called a *proof outline*. A proof outline consists of a program interspersed with assertions. Each subprogram R is preceded and succeeded by an assertion called pre(R) and post(R), respectively. These assertions satisfy certain natural conditions (see below) which make the application of the appropriate rules justified. Observe that we refer here to the proofs of partial correctness.

The following lemma due to Owicki [O] clarifies the notion of a proof outline.

Lemma 3.2

Let S be a deterministic program and let $S_1,...,S_k$ be the list of all subprograms of S. Then $\vdash_{PC} \{p\}\ S\ \{q\}$ iff there exist assertions $pre(S_i)$ and $post(S_i)$ for $i = 1,...,k$ such that

(i) $p \rightarrow pre(S)$, $post(S) \rightarrow q$,

(ii) $pre(S_i) \rightarrow post(S_i)[t/x]$ if S_i is $x := t$,

(iii) $pre(S_i) \rightarrow pre(S_j)$, $post(S_j) \rightarrow pre(S_1)$, $post(S_1) \rightarrow post(S_i)$

if S_i is $S_j ; S_1$,

(iv) $pre(S_i) \wedge B \rightarrow pre(S_j)$, $pre(S_i) \wedge \neg B \rightarrow pre(S_1)$, $post(S_j) \rightarrow post(S_i)$,

$post(S_1) \rightarrow post(S_i)$ if S_i is **if** B **then** S_j **else** S_1 **fi**,

(v) $pre(S_i) \rightarrow post(S_i)$, $post(S_j) \wedge B \rightarrow pre(S_j)$, $post(S_j) \wedge \neg B \rightarrow post(S_i)$

if S_i is **while** B **do** S_j **od** .

Proof. See (essentially) Owicki [O]. □

The proof outlines satisfy the following easy-to-prove lemma.

Lemma 3.3 (Strong soundness for deterministic programs)

Suppose that a proof outline of $\{p\}\ S\ \{q\}$ is given. Then for every computation of S which starts in a state σ satisfying p if

$$<S,\sigma> \rightarrow^* <R ; R_1 , \sigma_0> \rightarrow^* <R_1,\sigma_1>$$

for a subprogram R of S and some R_1, σ and σ_0, then

$$\models pre(R)\ (\sigma_0) \quad \text{and} \quad \models post(R)(\sigma_1) . \qquad \square$$

Informally, the conclusion of this lemma states that for every computation of S starting in a state satisfying p and every substatement R of S if the control is in front of R then pre(R) holds and if the control is just after R then post(R) holds. In other words the pre- and post-assertions hold at the appropriate moments.

This lemma is a generalization of the usual soundness theorem concerning the proof system PC.

Now observe that the above lemma does not hold any more when we consider proof outlines of the components of a parallel program simultaneously. Indeed, consider the proof outlines

$$\{x = 0\}\ x := x + 1\ \{x = 1\}$$

and

$$\{x = 0\}\ x := 0\ \{x = 0\}$$

and a computation of the program $[x := x + 1 \parallel x := 0]$ starting in a state in which $x = 0$ holds. Then it is not true that whenever the control is after $x := x + 1$, $x = 1$ holds. And similarly with the other control points.

The reason is that the above proof outlines do not take into account a possible interaction of the other components. This brings us to the following definitions.

Definition 3.3. Given a proof outline of $\{p\}\ S\ \{q\}$ and a statement R with a pre-assertion pre(R), we say that R *does not interfere with* the proof outline of $\{p\}\ S\ \{q\}$ if the following two conditions hold

i) for all subprograms T of S :

$$\{pre(T) \wedge pre(R)\}\ R\ \{pre(T)\}$$

(R preserves all pre-assertions)

ii) $$\{Q \wedge pre(R)\}\ R\ \{Q\}$$

(R preserves the final post-assertion)

Definition 3.4. The proof outlines of $\{p_1\}\ S_1\ \{q_1\},\ldots,\{p_n\}\ S_n\ \{q_n\}$ are *interference-free* if no assignment from one component interferes with the proof outline of another component.

We now introduce the following proof rule

RULE 15: RULE OF PARALLEL COMPOSITION

$$\frac{\text{proof outlines of } \{p_i\}\ S_i\ \{q_i\}\ ,\ i = 1,\ldots,n \quad \text{are interference-free}}{\{\bigwedge_{i=1}^{n} p_i\}\ [S_1 \parallel \ldots \parallel S_n]\ \{\bigwedge_{i=1}^{n} q_i\}}\ .$$

The following lemma constitutes a counterpart of Lemma 3.3. and justifies the above rule.

Lemma 3.4. (Strong soundness for parallel programs)

Suppose that interference-free proof outlines of $\{p_1\}\ S_1\ \{q_1\},\ldots,\{p_n\}\ S_n\ \{q_n\}$ are given. Then for every computation of $[S_1 \parallel \ldots \parallel S_n]$ which starts in a state σ satisfying $\bigwedge_{i=1}^{n} p_i$ if

$$<[S_1 \parallel \ldots \parallel S_n],\sigma] \rightarrow^* <[R_1 \parallel \ldots \parallel R_n],\sigma_0> \rightarrow^* <[T_1 \parallel \ldots \parallel T_n],\sigma_1>$$

for a subprogram R of S_i such that $R_i \equiv R; T_i$, then

$$\models \text{pre}(R)\ (\sigma_0) \quad \text{and} \quad \models \text{post}(R)\ (\sigma_1)\ .$$

In particular, if

$$<[S_1 \parallel \ldots \parallel S_n],\sigma> \rightarrow^* <[E \parallel \ldots \parallel E],\sigma_1>$$

then $\models \bigwedge_{i=1}^{n} q_i(\sigma_1)$.

Proof. Straightforward by induction on the length of the computation. □

To see the use of this rule consider the correctness formula

$\{x = 0\}\ [x := 0 \parallel x := x + 1]\ \{x = 0 \vee x = 1\}$ with the following proof presented in the form

of proof outlines:

$$\{x = 0\}$$

$$[\{x = 0 \vee x = 1\} \quad x := 0 \; \{x = 0 \vee x = 1\}$$

$$\| \{x = 0\} \quad x := x + 1 \; \{x = 0 \vee x = 1\}$$

$$]$$

$$\{x = 0 \vee x = 1\}$$

Note that the proof outlines are indeed interference-free.

Unfortunately, as in the case of disjoint parallelism the above proof rule is not sufficient for proving all properties of parallel programs.

It is easy to see that this rule does not suffice to prove the correctness formula $\{x = 0\}\ [x := x + 1 \parallel x := x + 1]\ \{x = 2\}$. Indeed, suppose by contradiction that for interference-free proof outlines

$$\{x = 0\}$$

$$[\ \{p_1\}\ x := x + 1\ \{q_1\}$$

$$\|\ \{p_2\}\ x := x + 1\ \{q_2\}$$

$$]$$

$$\{x = 2\}$$

holds.

Then also

$$\{x = 0\}$$

$$[\ \{p\}\ x := x + 1\ \{q\}$$

$$\|\ \{p\}\ x := x + 1\ \{q\}$$

$$]$$

$$\{x = 2\}$$

is a valid proof where $p \equiv p_1 \wedge p_2$ and $q \equiv q_1 \wedge q_2$. We thus have

i) $x = 0 \rightarrow p$

ii) $\{p\}\ x := x + 1\ \{p\}$ by interference freedom, i.e., for all $x \quad p(x) \rightarrow p(x + 1)$.

Then by induction $\forall x \geq 0\ p(x)$ and since $\{p\}\ x := x + 1\ \{q\}$ holds, $\forall x \geq 1\ q(x)$ is true. But then q cannot imply $x = 2$. Contradiction.

As in the case of disjoint parallelism we strengthen the proof system by supplementing it with the rule of auxiliary variables. This time, however, we also need an additional construct allowing to turn a deterministic program R into an indivisible action: $\langle R \rangle$. To understand better its function in conjunction with the rule of auxiliary variables consider the following correctness proof.

The proof outlines

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

and

$$\{z = 0\}\ \langle x := x + 1\ ;\ z := 1 \rangle\ \{z = 1\}$$

are interference-free. Using the parallel composition rule and the composition rule we obtain $\{x = 0\}\ z := 0\ ;\ [x := x + 1 \parallel \langle x := x+1\ ;\ z := 1 \rangle]\ \{x = z + 1 \wedge z = 1\}$, so by the consequence rule

$$\{x = 0\}\ z := 0\ ;\ [x := x + 1 \parallel \langle x := x + 1\ ;\ z := 1\rangle]\ \{x = 2\}.$$

Now by the rule of auxiliary variables

$$\{x = 0\}\ [x := x + 1 \parallel \langle x := x + 1 \rangle]\ \{x = 2\}\ .$$

But the assignments are by assumption indivisible actions so we can drop the brackets around $x := x + 1$ and obtain the correctness formula discussed before.

To justify the above proof we need to modify the notions of a proof outline and of interference freedom.

In the definition of a proof outline we simply do not put any assertion within a subprogram of the form $\langle R \rangle$. On the other hand we require that $\{pre(R)\}\ R\ \{post(R)\}$ be provable in the proof system PC. In the definition of interference freedom we now require that no assignment *and* no subprogram of the form $\langle R \rangle$ of one component interferes with the proof outline of another component.

Finally we need the following natural rule.

RULE 16: REDUCTION RULE

Let S' be obtained from S by replacing each subprogram of S of the form <R>, where R is an assignment, by R. Then

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

This completes the presentation of the proof system. Observe that the above proof system is appropriate for proofs of partial correctness only as the proof outlines referred to the proof system PC.

3.3.2 Total correctness

In order to prove total correctness of parallel programs we first introduce a notion of a proof outline appropriate for total correctness. As before a proof outline consists of a program interspersed with assertions. They now satisfy the same conditions as before with the exception of the following ones for the case of while-subprograms:

(v') $r(n) \wedge B \rightarrow pre(S_j)$, $post(S_j) \rightarrow \exists m < n\ r(m)$, $pre(S_i) \rightarrow \exists n\ r(n)$,

$\exists n\ r(n) \wedge \neg B \rightarrow post(S_i)$ if S_i is while B do S_j od, where $r(n)$ is an assertion with a free variable n which does not appear in S_i and ranges over natural numbers.

The following lemma justifies this new definition of a proof outline.

Lemma 3.5

Let S be a deterministic program and let $S_1,...,S_n$ be the list of all subprograms of S. Then $\vdash_{TC} \{p\}\ S\ \{q\}$ iff there exist assertions $pre(S_i)$ and $post(S_i)$ for $i = 1,...,k$ such that the conditions (i) - (iv) of Lemma 3.2 and (v') listed above are satisfied.

Proof. The proof proceeds by induction on the structure of the program S. We consider here only the case of while-programs. The proofs of other cases are the same as in Lemma 3.2.

Let S be of the form while B do S_0 od.

if part

Suppose that $\vdash_{TC} \{p\}\ S\ \{q\}$ holds for some assertions p and q. The last steps of the proof must have consisted of an application of rule 8 (the rule of repetitive command II) followed by a possibly empty number of applications of the consequence rule which can be combined into exactly one application. We thus have $p \rightarrow \exists n\ r(n)$, $\exists n\ r(n) \wedge \neg B \rightarrow q$ and

$$\vdash_{TC} \{r(n) \wedge B\}\ S_0\ \{\exists m < n\ r(m)\} \qquad (1)$$

for some assertion $r(n)$ with a free variable n which does not occur in S and ranges over natural numbers.

We now define the pre- and post-assertions for S and S_0 putting

$pre(S) \equiv p$, $post(S) \equiv q$, $pre(S_0) \equiv r(n) \wedge B$, $post(S_0) \equiv \exists m < n\ r(m)$.

Note that the relevant conditions listed in (i) and (v') are obviously satisfied.

Since (1) holds, by the induction hypothesis there exist appropriate pre- and post-assertions for all subprograms of S_0 which satisfy the conditions listed in the lemma. Disregard the new definitions of $pre(S_0)$ and $post(S_0)$ which satisfy the condition (i) w.r.t. (1). All other pre- and post-assertions together with those defined above satisfy the conditions of the lemma w.r.t. $\{p\}\ S\ \{q\}$.

Only if part

Suppose that the appropriate assertions satisfying the conditions of the lemma for $\{p\}\ S\ \{q\}$ exist. Delete now from this list the assertions $pre(S) \rightarrow \exists n\ r(n)$ and $\exists n\ r(n) \wedge \neg B \rightarrow post(S)$ concerning the program S and listed in (v'). In such a way we obtain appropriate assertions which satisfy the conditions of the lemma in the case of the correctness formula $\{r(n) \wedge B\}\ S_0\ \{\exists m < n\ r(m)\}$.

By the induction hypothesis $\vdash_{TC} \{r(n) \wedge B\}\ S_0\ \{\exists m < n\ r(m)\}$. By the assumption the assertions $p \rightarrow pre(S)$, $pre(S) \rightarrow \exists n\ r(n)$, $\exists n\ r(n) \wedge \neg B \rightarrow post(S)$ and $post(S) \rightarrow q$ hold, so by the consequence rule $\vdash_{TC} \{p\}\ S\ \{q\}$ holds as desired. □

To prove total correctness of parallel programs it is sufficient to use the same proof rules as before but now using proof outlines for total correctness.

As an example consider the program $S \equiv$ [**while** $x > 0$ **do** $x := x-1$ **od** $\parallel$ $x := 0$]. We now prove {**true**} S $\{x = 0 \vee x = -1\}$ in the sense of total correctness. We present the proof in the form of proof outlines together with an appropriate commentary.

{**true**}

$z := 0$;

$\{z = 0\}$

[$\{\exists\, n\ p(n)\}$

while $x > 0$ **do** $\{n \geq x + 1 \wedge$ **if** $z = 1$ **then** $x = 0$ **else** $x > 0$ **fi**$\}$

$x := x - 1$

$\{\exists m < n(m \geq x + 1) \wedge$ **if** $z = 1$ **then** $x = 0 \vee x = -1$ **else** $x \geq 0$ **fi**$\}$

od

$\{\exists\, n\ p(n) \wedge x \leq 0\}$

$\parallel$ $\{z = 0\}$

$\langle x := 0 ; z := 1 \rangle$

$\{z = 1\}$

]

$\{\exists\, n\ p(n) \wedge x \leq 0 \wedge z = 1\}$

$\{x = 0 \vee x = -1\}$

where $p(n) \equiv n \geq x + 1 \wedge$ **if** $z = 1$ **then** $x = 0 \vee x = -1$ **else** $x \geq 0$ **fi**.

Consecutive occurrences of assertions indicate an application of the consequence rule.

First, observe that we indeed deal with proof outlines. The assertions to verify are

$$p(n) \wedge x > 0 \rightarrow n \geq x + 1 \wedge \textbf{if } z = 1 \textbf{ then } x = 0 \textbf{ else } x > 0 \textbf{ fi}$$

and

$$\exists m < n(m \geq x + 1) \wedge \textbf{if } z = 1 \textbf{ then } x = 0 \vee x = -1 \textbf{ else } x \geq 0 \textbf{ fi} \rightarrow \exists m < n\ p(m).$$

They are clearly satisfied. Next we prove the interference freedom. The variable x is not mentioned in the second proof outline. Thus the assignment x:= x − 1 does not interfere with the second proof outline. Consider now the statement < x:= 0 ; z:= 1 > and the first proof outline.

We clearly have

$$\{\exists n\, p(n) \wedge z = 0\} \; \langle x := 0 \; ; \; z := 1 \rangle \; \{\exists n \; p(n)\} \; .$$

The other two cases are equally straightforward to verify. The desired result now follows by the parallel composition rule, rule of auxiliary variables and the reduction rule.

There still remains an issue of total correctness of parallel programs with shared variables under the assumption of justice. We shall handle this problem after discussing the synchronization constructs.

3.4 General parallel programs

In a realistic situation parallel programs are executed in the presence of some synchronization constraints. These constraints cannot be expressed using the syntax discussed so far. Following Owicki and Gries [OG1] we now additionally allow within the context of parallel composition of programs the construct of the form **await B then R** and where R is a deterministic program. Informally, a component program executes an **await**-statement iff with its turn to execute the Boolean expression B evaluates to **true**. R is then executed as an indivisible action.

The **await**-construct is of course too powerful to be implemented efficiently but it allows to model various other more realistic synchronization constructs. For the purpose of correctness proofs of the parallel programs using the latter constructs such as modeling, however inefficient, suffices.

The above programs are subsequently called *general parallel programs* (GP programs, in short).

We now define the semantics of GP programs by adding the following clause to the list considered up until now:

ix) $\langle$**await** B **then** R **end**, $\sigma\rangle \rightarrow \langle E, \tau\rangle$

if $\models B(\sigma)$ and $\langle R, \sigma\rangle \rightarrow^* \langle E, \tau\rangle$.

Introduction of the **await** -statements leads to a new possibility of abnormal termination of a program - that of a deadlock.

Definition 3.5

i) A configuration $\langle[S_1 \| \ldots \| S_n], \sigma\rangle$ is called *deadlocked* if for some i $S_i \not\equiv E$ and moreover $\langle[S_1 \| \ldots \| S_n],\sigma\rangle$ has no successor w.r.t. the relation "$\rightarrow$".

ii) We say that a GP program S *can deadlock from* σ if for some deadlocked configuration $\langle S', \tau\rangle$, $\langle S,\sigma\rangle \rightarrow^* \langle S', \tau\rangle$.

We now define a new $\mathcal{M}_{tot}$ semantics of GP programs taking into account the possibility of deadlocks:

$$\mathcal{M}_{tot}\,[[S]]\,(\sigma) = \mathcal{M}\,[[S]]\,(\sigma) \cup \{\perp\colon \text{S can diverge from } \sigma\} \cup \{\Delta\colon \text{S can deadlock from } \sigma\}$$

where fail is not mentioned because by the syntax restrictions no failure can now arise.

Two different semantics of GP programs taking into account the fairness or justice assumption will be defined and discussed later.

To prove correctness of GP programs we have to in the first place provide a proof rule concerning the **await** -statement. The following rule was proposed in [OG1].

RULE 17: AWAIT RULE

$$\frac{\{p \wedge B\}\ R\ \{q\}}{\{p\}\ \textbf{await}\ B\ \textbf{then}\ R\ \textbf{end}\ \{q\}}$$

Using this rule in conjunction with the proof system PC or TC we can prove partial or total correctness of the components of GP programs. However, in order to deal adequately

with the semantics of the await -statements we adopt the following refined definitions of proof outlines. By a *normal* subprogram of a GP program we mean a subprogram which is not a proper subprogram of an await -statement. Then a proof outline of {p} S {q} , where S is a component program, is the program S together with the assertions p, q and $pre(S_i)$, $post(S_i)$ for all normal subsets S_i of S which satisfy the following conditions:

a) for partial correctness

1°. $\vdash_{PC} \{pre(S_i) \wedge B\}\ R\ \{post(S_i)\}$

if S_i is await B then R end ,

2°. conditions (i) - (v) of Lemma 3.2;

b) for total correctness

3°. $\vdash_{TC} \{pre(S_i) \wedge B\}\ R\ \{post(S_i)\}$

if S_i is await B then R end ,

4°. conditions (i) - (iv) of Lemma 3.2 ,

5°. conditions (v') of Section 3.3.2.

Similarly as before it is easy to prove the following lemma:

Lemma 3.6. Let S be a component of a GP program. Then

i) $\vdash_{PC+\ rule\ 16} \{p\}\ S\ \{q\}$ iff there exists a proof outline for partial correctness of {p} S {q}.

ii) $\vdash_{TC+\ rule\ 16} \{p\}\ S\ \{q\}$ iff there exists a proof outline for total correctness of {p} S {q}. □

Also, in the same way as before we accommodate the definition of proof outlines to cater for the case of subprograms of the form <R>. We intend to use rule 15 to prove correctness of GP programs. To this purpose we have to modify appropriately the definition of interference freedom. We now say that proof outlines for components of a GP program are interference-free if no assignment or an await -statement or a subprogram of the form <R> of one component interferes with the proof outline of another component.

Using now rules 14 - 16 we can prove partial correctness of GP programs. However, to prove total correctness of GP programs it is not sufficient to use proof outlines for total correctness for the component programs - in presence of the await -statements we still have to handle the problem of *deadlock freedom.*

We follow here the approach of [OG1]. Let $S \equiv [S_1 \| ... \| S_n]$ be a GP program. An n-tuple of programs $<R_1,...,R_n>$ is called a *blocked tuple* if

i) each R_i is either an await -statement being a subprogram of S_i or E ,

ii) $\exists$ i $R_i \not\equiv$ E.

Suppose that interference-free proof outlines for partial correctness of $\{p_1\}\ S_1\ \{q_1\},...,\{p_n\}\ S_n\ \{q_n\}$ are given. With each blocked tuple $<R_1,...,R_n>$ of S we associate an n-tuple $<r_1,...,r_n>$ of assertions defined as follows:

if $R_i \equiv$ await B then R end then $r_i \equiv pre(R_i) \wedge \neg B$,

if $R_i \equiv$ E then $r_i \equiv q_i$.

Suppose now that interference-free proof outlines for total correctness of $\{p_1\}\ S_1\ \{q_1\},...,\{p_n\}\ S_n\ \{q_n\}$ are given. We then associate with each blocked tuple $<R_1,...,R_n>$ of S the corresponding n-tuple of assertions as before but we prefix each assertion with $\exists n_1 \ ... \ \exists n_k$ where $n_1,...,n_k$ are parameter variables occurring in it.

The following can be proved for proof outlines for both partial and total correctness. If S is executed in an initial state satisfying the assertions $p_1,...,p_n$ and deadlocks then the corresponding assertions $r_1,...,r_n$ associated with the reached blocked tuple are satisfied.

We say that S is *deadlock-free relative to the assertion* p if in the computations of S starting in a state satisfying p deadlock cannot arise. The following lemma is a direct consequence of the above.

Lemma 3.7. Suppose that interference-free proof outlines of $\{p_1\}\ S_1\ \{q_1\},...,\{p_n\}\ S_n\ \{q_n\}$ are given. Then $S \equiv [S_1 \| ... \| S_n]$ is deadlock-free relative to $\bigwedge_{i=1}^{n} p_i$ if for all blocked tuples $<R_1,...,R_n>$ $\neg \bigwedge_{i=1}^{n} r_i$ holds for the corresponding tuple of assertions $<r_1,...,r_n>$. □

This lemma allows us to handle the proofs of deadlock-freedom. Thus to prove total correctness of a GP program it is enough to find appropriate proof outlines for total correctness which satisfy the conditions of the above lemma. Then the conclusion of rule 15 holds in the sense of total correctness.

3.5. Justice and fairness

3.5.1 Justice

Consider the program

$$b\text{: = true; [while } b \text{ do skip od } \| \ b\text{: = false].}$$

Under the assumption of justice this program always terminates and without this assumption termination is not guaranteed. Thus in the presence of shared variables there is a difference between total correctness of parallel programs under and without the assumption of justice. To prove total correctness of parallel programs under the assumption of justice we follow the approach of Olderog and Apt [OA] and as in the case of nondeterministic programs we use program transformations.

Let $S \equiv S_0;[S_1 \| \ \ldots \ \| S_n]$ be a parallel program with shared variables. We define $T_{\text{just}}(S)$ as the program obtained from S by the following steps:

1. prefix S with an initialization part

$$\text{INIT} \equiv z_1\text{: = ?; } \ldots \text{ ; } z_n\text{: = ?; end}_1\text{: = false; } \ldots \text{ ; end}_n\text{: = false,}$$

2. replace every loop **while** B **do** R **od** of a component S_i by

```
while B do
        await z̄ ≥ 1  then z_i: = ?;
                      for j ≠ i do
                    [ ¬ end_j → z_j: = z_j − 1 □ end_j → skip]
                              od;
                      end;
        R
        od,
```

3. suffix every component S_i by

$$\text{END}_i \equiv \text{end}_i\text{: = true.}$$

We assume that none of the variables in the set $Z = \{z_1, \ldots, z_n, \mathrm{end}_1, \ldots, \mathrm{end}_n\}$ occurs in the original program S.

The following lemma clarifies the relation between the programs S and $T_{\mathrm{just}}(S)$:

Lemma 3.8. For every parallel program S and a state σ

$$\mathcal{M}_{\mathrm{just}}[[S]](\sigma) = \mathcal{M}_{\mathrm{tot}}[[T_{\mathrm{just}}(S)]](\sigma) - \{\Delta\} \bmod Z. \ \square$$

A proof can be found in [OA]. The following corollary is immediate.

Corollary 3.1. For all assertions p and q without free variables from the set Z

$$\models_{\mathrm{just}} \{p\}S\{q\} \textit{ iff } \models_{\mathrm{tot}-\Delta} \{p\}T_{\mathrm{just}}(S)\{q\}. \ \square$$

Here the subscript "tot-Δ" refers to total correctness modulo deadlocks - a notion obtained by disregarding possible deadlocks in the definition of total correctness.

Now, to prove total correctness modulo deadlocks of $T_{\mathrm{just}}(S)$ we first have to take care of random assignments. To this purpose it is necessary to refine the notion of a proof outline for total correctness in the presence of random assignments. While proving total correctness of the component programs we shall now use repetitive command rule III instead of repetitive command rule II. This requires replacement of the conditions concerning the while-subprograms ((v′) of subsection 3.3.2) by the following ones:

(v″) $r(\alpha) \wedge B \rightarrow \mathrm{pre}(S_j)$, $\mathrm{post}(S_j) \rightarrow \exists \beta < \alpha r(\beta)$, $\mathrm{pre}(S_i) \rightarrow \exists \alpha r(\alpha)$, $\exists \alpha r(\alpha) \wedge \neg B \rightarrow \mathrm{post}(S_i)$

if S_i is while B do S_j, od, where $r(\alpha)$ is an assertion with a free variable α which does not appear in S_i and ranges over ordinals,

and add the following one concerning random assignments:

(vi) $\mathrm{pre}(x{:}=?) \rightarrow \forall x \geq 0\ \mathrm{post}(x{:}=?)$.

Thus to prove total correctness of a parallel program S under the assumption of justice it is enough to:

1. Find proof outlines for the total correctness of the component programs of $T_{\mathrm{just}}(S)$. They are to satisfy the conditions listed under b) in Section 3.4 but with (v′) replaced by (v″), and (vi) added.

2. Prove that they are interference free.

3. Apply the rule of parallel composition (rule 15) and possibly the composition rule, the rule of auxiliary variables, reduction rule and consequence rule.

4. Apply the following rule.

RULE 18: JUSTICE RULE

$$\frac{\{p\}\ T_{just}(S)\ \{q\}}{\{p\}\ S\ \{q\}}$$

Observe that in contrast to the case of nondeterministic programs we did not "absorb" the transformation $T_{just}(S)$ into the assertions of existing rules. For parallel programs the idea of absorption does not work properly. This is due to the interface freedom test which has to deal with the assignments to the auxiliary variables from the set Z and perhaps some other ones. Even if assignments to these variables were absorbed into the assertions of the proof outlines for the components of S, they would reappear in the final test of interference freedom. Therefore, we rather propose to apply the transformations as a part of the correctness proofs.

3.5.2. Weak fairness

In the presence of the await-statements the assumption of justice is not any more an appropriate one. To see this consider the program

$$S \equiv \textbf{[while true do skip od } \| \textbf{ await false then skip end]}.$$

Then for any state σ

$$\mathcal{M}_{just}[[S]](\sigma) = \phi$$

i.e. the program S neither diverges nor converges! Intuitively, the program S should diverge as there is no way to activate its second component and the first one diverges.

An appropriate notion of fairness is obtained by replacing in the definition of justice the notion of termination of a component by the notion of *enabledness*.

Definition 3.6

Let $S \equiv S_0;[S_1 \| \ldots \| S_n]$ be a GP program and let ξ be a computation of S.

i) We say that the i-th component is *enabled* in the configuration $< [T_1 \| \ldots \| T_n], \sigma >$ if T_i is not terminated, i.e. $T_i \not\equiv E$ and whenever T_i is of the form **await** B **then** R **end**; T'_i then $\models B(\sigma)$ holds.

ii) We say that ξ is *weakly unfair* if it is infinite and some component is from a certain moment on continuously enabled, but is only finitely many times active in ξ.

iii) We say that ξ is *weakly fair* if it is not weakly unfair.

Note the close correspondence between the notions of weak fairness for nondeterministic and general parallel programs. We now define a weakly fair semantics of GP programs by putting

$$\begin{aligned}\mathcal{M}_{\text{wfair}}[[S]](\sigma) = \mathcal{M}[[S]](\sigma) &\\ \cup \{\perp: S \text{ can diverge from } \sigma \text{ by} &\\ \quad \text{a weakly fair computation}\} &\\ \cup \{\Delta: S \text{ can deadlock from } \sigma\}. &\end{aligned}$$

Observe that under the assumption of weak fairness the program S considered above diverges, i.e.

$$\mathcal{M}_{\text{wfair}}[[S]](\sigma) = \{\perp\}.$$

To prove total correctness of GP programs under the assumption of weak fairness similarly as before we first exhibit an appropriate program transformation provided in [OA].

Let $S \equiv S_0;[S_1 \| \ldots \| S_n]$ be a GP program. We first need to formalize the notion of enabledness. To this purpose we introduce new auxiliary variables $pc_1, \ldots, pc_n$ which will be used as a restricted form of program counters indicating when the component S_i is in front of an await-statement, and if so in front of which one. To this end, we assign to every occurrence of an await-statement in S_i a unique number $\ell \geq 1$ as a label. Let L_i denote the set of all these labels for S_i and let B_ℓ denote the Boolean guard of the await-statement labeled by ℓ. We assume that $0 \notin L_i$.

We now put

$$\text{enabled}_i \equiv \neg\text{end}_i \wedge \bigwedge_{\ell \in L_i} (pc_i = \ell \rightarrow B_\ell).$$

In contrast to the case of justice we have to check here the enabledness of a component in front of every while-statement or an atomic statement (see [OA]). More precisely we need the following notion.

Definition 3.7 By an *immediate atomic statement* of a loop **while** B **do** R **od** we mean an atomic statement, an await-statement or a while-statement which is a subprogram of R but which lies outside any while-statements within R.

For example in the program

while B **do while** C **do** x: = 1 **od od**

the assignment x: = 1 is an immediate atomic statement of the inner loop and **while** C **do** x : = 1 **od** is the only immediate atomic statement of the outer loop.

The program $T_{\text{wfair}}(S)$ is now obtained from S by

1. prefixing S with an initialization part

$$\begin{aligned} \text{INIT} \equiv\ & z_1 := ? \ ; \ldots ; z_n := ?; \\ & \text{end}_1 := ? ; \ldots ; \text{end}_n := ?; \\ & \text{pc}_1 := 0 \ ; \ldots ; \text{pc}_n := 0, \end{aligned}$$

2. replacing every substatement ℓ: **await** B_ℓ **then** R **end** of S_i by

$$\text{pc}_i := \ell;\ \textbf{await}\ B_\ell\ \textbf{then}\ R;\ \text{pc}_i := 0\ \textbf{end},$$

3. inserting in every loop **while** B **do** R **od** of a component S_i

   a. in front of the first immediate atomic statement of R:

$\text{TEST}_i \equiv$ **await** $\bar{z} \geq 1$ **then** $z_i := ?$;
for $j \neq i$ **do**
[enabled$_j \rightarrow z_j := z_j - 1 \; \square \; \neg$ enabled$\rightarrow z_j := ?$]
od
end,

b. in front of every other immediate atomic statement of R:

$\text{RESET}_i \equiv$ **await true then**
for $j \neq i$ **do**
[enabled$_j \rightarrow$ skip $\square \; \neg$ enabled$_j \rightarrow z_j := ?$]
od
end

4. suffixing every component S_i of S with $\text{END}_i \equiv \text{end}_i := \text{true}$.

We assume that none of the variables in the set $z = \{z_1, \ldots, z_n, \text{end}_1, \ldots, \text{end}_n, \text{pc}_1, \ldots, \text{pc}_n\}$ occurs in the original program S.

The following lemma whose proof can be found in [OA] relates the programs S and $T_{\text{wfair}}(S)$:

Lemma 3.9. For every GP program S and a state σ

$$\mathcal{M}_{\text{wfair}}[[S]](\sigma) - \{\Delta\} = \mathcal{M}[[T_{\text{wfair}}(S)]](\sigma) - \{\Delta\} \text{ mod } Z. \; \square$$

Corollary 3.2. For all assertions p and q without free variables from the set Z

$$\models_{\text{wfair}-\Delta} \{p\}S\{q\} \text{ iff } \models_{\text{tot}-\Delta} \{p\}T_{\text{wfair}}(S)\{q\}. \; \square$$

Here the subscript "wfair-Δ" refers to total correctness under the assumption of weak fairness but modulo deadlocks.

This corollary shows that in order to prove total correctness of a GP program under the assumption of weak fairness it is enough to

1. Prove total correctness modulo deadlocks of T_{wfair} in the same way as in the case of justice.
2. Prove deadlock freedom of S using lemma 3.5 from section 3.4.

3.5.3. Strong fairness

Analogously to the case of nondeterministic programs there is another natural fairness assumption concerning GP programs - that of strong fairness. According to this hypothesis a component of a GP program will be activated if it is infinitely often enabled. This is a stronger requirement than that of weak fairness which guarantees activation of a component only if it is continuously enabled. More precisely we adopt the following definition.

Definition 3.8

Let $S \equiv S_0;[S_1 \| \ \ldots \ \| S_n]$ be a GP program and let ξ be a computation of S.

i) We say that ξ is *strongly unfair* if it is infinite and some component is infinitely often enabled but is only finitely many times active in ξ.

ii) We say that ξ is *strongly fair* if it is not strongly unfair.

The strongly fair semantics $\mathcal{M}_{\text{sfair}}$ is defined in an analogous manner as the $\mathcal{M}_{\text{wfair}}$ semantics. To prove total correctness of GP programs under the strong fairness assumption we proceed through the same steps as in the case of weak fairness. The corresponding transformation $T_{\text{sfair}}(S)$ is defined by applying steps 1, 2 and 4 of $T_{\text{wfair}}(S)$ but with the following new step 3.

3. Insert in front of every immediate atomic statement of every **while**-loop of S_i

$$\begin{aligned}\text{TEST}_i \equiv\ & \textbf{await } \bar{z} \geq 1 \textbf{ then } z_i := ?; \\ & \textbf{for } j \neq i \textbf{ do} \\ & [\text{enabled}_j \rightarrow z_j := z_j - 1 \ \square \ \neg \text{ enabled} \rightarrow z_j := ?] \\ & \quad \textbf{od} \\ & \textbf{end.}\end{aligned}$$

As in the case of weak fairness the following lemma relates the programs S and $T_{\text{sfair}}(S)$ (see [OA]):

Lemma 3.10. For every GP program S and a state σ

$$\mathcal{M}_{\text{sfair}}[[S]](\sigma) - \{\Delta\} = \mathcal{M}[[T_{\text{sfair}}(S)]](\sigma) - \{\Delta\} \bmod Z.$$

where Z is defined as in the previous subsection. □

The remaining steps in proving total correctness under the strong fairness are exactly the same as in the previous section and are omitted.

Further Reading

The approach to program correctness studied in this paper has been applied to several other classes of concurrent programs. For the benefit of the reader we now provide a number of pointers to the literature.

The await - statement considered in sections 3.4 and 3.5 is a very inefficient synchronization construct. A more efficient synchronization statement - the conditional critical region statement coupled with the use of resources, originally suggested in Hoare [H2], is studied from the point of view of program correctness in Owicki and Gries [OG2]. Clarke [C] discusses the issue of systematic construction of resource invariants for the programs written in the above language.

The correctness of programs written in CSP - *Communicating Sequential Processes,* a language introduced in Hoare [H3], has been studied in several papers. A proof system motivated by the proof system from section 3.4 for the GP programs is introduced in Levin and Gries [LG]. A proof system motivated by the proof system of [OG2] for the language using resources, is presented in Apt, Francez and De Roever [AFR]. Apt [A5] provides a simpler and structured exposition of the latter system and lists other entries to the literature on the subject of correctness of CSP programs.

The correctness of programs written in DP - *Distributed Processes,* a language introduced in Brinch Hansen [B], has been studied in Gerth, De Roever and Roncken [GRR1].

The correctness of a fragment of ADA involving tasks has been most extensively studied in Gerth and De Roever [GR]. Alternative proof systems were presented in Barringer and Mearns [BM] and Schlichting and Schneider [SS]. In the last paper also proof rules for asynchronous message passing are introduced.

In Francez, Hailpern and Taubenfeld [FHT] a communication abstraction mechanism called Script is introduced and proof rules for programs using it are presented. Finally, De Roever [R] provides an overview of the proof systems introduced in Apt, Francez and De Roever [AFR], Gerth, De Roever and Roncken [GRR1] and Gerth and De Roever [GR].

In none of the above papers the issue of fairness has been considered. Correctness of CSP programs under the assumption of fairness is studied in Grumberg, Francez and Katz [GFK].

Correctness proofs of nontrivial concurrent programs using the above methods have been given in various papers. We only mention here correctness proofs of a parallel garbage collector given in Gries [Gr], of a distributed algorithm maintaining message - routing tables in a network given in Lamport [L2], of the "Dutch National Torus," a program written in DP, given in [GRR2], and of a solution to the distributed termination problem of Francez [F] given in Apt [A5].

Finally, the issue of soundness and completeness of the above systems has been studied - for the language studied in Section 3.4 in Apt [A2], for the language using critical section statement in Owicki [O1], for CSP in Apt [A3] and for a fragment of ADA in Gerth [G].

Acknowledgements. E. Börger talked me into writing this paper. The exposition owes much to the joint work done with E.-R. Olderog and profited from the discussions with A. Pnueli. N. Perry and B. White typed the paper fast and efficiently.

References

[A1] Apt., K.R., "*Ten years of Hoare's logic, a survey - part I*", ACM TOPLAS 3(4), pp. 431-483, 1981.

[A2] Apt., K.R., "*Recursive assertions and parallel programs*", Acta Informatica 15, pp. 219-232, 1984.

[A3] Apt., K.R., "*Formal justification of a proof system for Communicating Sequential Processes*", Journal ACM 30 (1) pp. 197-216, 1983.

[A4] Apt., K.R., "*Ten years of Hoare's logic, a survey - nondeterminism, part II*", Theoretical Computer Science 28, pp. 83-109, 1984.

[A5] Apt., K.R., "*Proving correctness of CSP programs - a tutorial*", in: Proc. International Summer School "*Control Flow and Data Flow: Concepts of Distributed Programming*", Marktoberdorf, Springer-Verlag, to appear.

[A6] Apt., K.R., "*Correctness proofs of distributed termination algorithms*", in: Proc. Advanced Course on Logics and Models for Verification and Specification of Concurrent Systems, La Colle-sur-Loup, Springer-Verlag, to appear.

[AFR] Apt., K.R., Francez, N. and de Roever, W.P., "*A proof system for Communicating Sequential Processes*", ACM TOPLAS 2(3), pp. 359-385, 1981.

[AD] Apt., K.R. and Olderog, E.-R., "*Proof rules and transformations dealing with fairness*", Science of Computer Programming 3, pp. 65-100, 1983.

[APS] Apt., K.R., Pnueli, A. and Stavi, J., "*Fair termination revisited - with delay*", Theoretical Computer Science 33, pp. 65-84, 1984.

[BM] Barringer, H. and Mearns, I., "*Axioms and proof rules for Ada tasks*", IEEE Proc. 129, Part E, 2, pp. 38-48, 1982.

[B] Brinch, Hansen P., "*Distributed process: a concurrent programming concept*", Communications ACM 21(11), pp. 934-941, 1978.

[C] Clarke, E.M., Jr., "*Synthesis of resource invariants for concurrent programs*", ACM TOPLAS 2(3), pp. 338-358, 1980.

[D1] Dijkstra, E.W., *"Guarded commands, nondeterminacy and formal derivation of programs"*, Communications ACM **18(8)**, pp. 453-457, 1975.

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

[F] Francez, N., *"Distributed termination"*, ACM TOPLAS **2(1)**, pp. 42-55, 1980.

[FHT] Francez, N., Hailpern, B., and Taubenfeld, G., *"Script: A communication mechanism and its verification"*, Science of Computer Programming, to appear.

[G] Gerth, R., *"A sound and complete Hoare axiomatization of the Ada-rendezvous"*, in: Proc. ICALP 82, Lecture Notes in Computer Science, vol. 140, Springer-Verlag, pp. 252-264, 1982.

[GR] Gerth, R., and De Roever, W.P., *"A proof system for concurrent ADA programs"*, Science of Computer Programming, **4(2)**, pp. 159-205, 1984.

[GRR1] Gerth, R., De Roever, W.P., and Roncken, M., *"Procedures and concurrency: a study in proof"*, in Proc. 5th International Symposium on Programming, Lecture Notes in Computer Science, vol. 137, Springer-Verlag, pp. 132-163, 1982.

[GRR2] Gerth, R., De Roever, W.P., and Roncken, M., *"A study in Distributed Systems and Dutch Patriotism"*, in: Proc. 2nd Conference FCT and TCS, Bangalore, 1982.

[Gr] Gries, D., *"An exercise in proving parallel programs correct"*, Communications ACM **20(12)**, pp. 921-930, 1977.

[GFK] Grumberg, O., Francez, N. and Katz, S., *"Fair termination of Communicating Processes"*, in: Proc. 3rd Annual Symposium on Principles of Distributed Computing, Vancouver, Canada, pp. 254-265, 1984

[HP] Hennessy, M.C.B. and Plotkin, G.D., *"Abstraction for a simple programming language"*, in: Proc. 8th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, vol. **74**, Springer-Verlag, pp. 108-120, 1979.

[H1] Hoare, C.A.R., *"An axiomatic basis for computer programming"*, Communications ACM **12(10)**, pp. 576-580, 583, 1969.

[H2] Hoare, C.A.R., *"Towards a theory of parallel programming"*, in: Operating Systems Techniques, pp. 61-71, (C.A.R. Hoare, R.H. Ferrat, eds.), Academic Press, 1972.

[H3] Hoare, C.A.R., "*Communicating Sequential Processes*", Communications ACM 21(8), pp. 666-677, 1978.

[L1] Lamport, L., "*Proving the correctness of multiprocessor programs*", IEEE Transactions on Software Engineering, vol. SE-3(2), pp. 125-143, 1977.

[L2] Lamport, L., "*An assertional correctness proof of a distributed algorithm*", Science of Computer Programming 2(3), pp. 175-206, 1982.

[LG] Levin, G. and Gries, D., "*A proof technique for Communicating Sequential Processes*", Acta Informatica 15(3), pp. 281-302, 1981.

[MP1] Manna, Z. and Pnueli, A., "*Verification of concurrent programs: the temporal framework*", in: The Correctness Problem in Computer Science, pp. 215-273, (R.S. Boyer and J.S. Moore, eds.), Academic Press, 1982.

[MP2] Manna, Z. and Pnueli, A., "*Verification of concurrent programs: temporal proof principles*", in: Proc. Workshop on Logic of Programs, Lecture Notes in Computer Science, vol. 131, Springer-Verlag, pp. 200-252, 1982.

[MP3] Manna, Z. and Pnueli, A., "*Proving precedence properties: temporal way*", in: Proc. ICALP '83, Lecture Notes in Computer Science, vol. 154, Springer-Verlag, pp. 491-512, 1983.

[OA] Olderog, E.-R. and Apt, K.R., "*Fairness in parallel programs: the transformational approach*", Institut fur Inform. und Prakt. Math., Christian-Albrechts-Universitat Kiel, Tech. Report 8402, 1984.

[O1] Owicki, S., "*Axiomatic proof techniques for parallel programs*", Computer Science Dept., Cornell University, PhD thesis, 1975.

[O2] Owicki, S., "*A consistent and complete deductive system for verification of parallel programs*", in: Proc. 8th Annual Symposium on Theory of Computing, pp. 73-86, 1976.

[OG1] Owicki, S. and Gries, D., "*An axiomatic proof technique for parallel programs I*", Acta Informatica 6, pp. 319-340, 1976.

[OG2] Owicki, S. and Gries, D., "*Verifying properties of parallel programs: an axiomatic approach*", Communications ACM 19(5), pp. 279-285, 1976.

[P] Pnueli, A., "*The temporal logic of programs*", in: Proc. 18th Annual Symposium on Foundations of Computer Science, pp. 46-57, 1977.

[R] De Roever, W.P., *"The cooperation test: a syntax directed verification method"*, in: Proc. Advanced Course on Logics and Models for Verification and Specification of Concurrent Systems, La Colle-sur-Loup, Springer-Verlag, to appear.

[SS] Schlichting, R.D. and Schneider, F.B., *"Using message passing for distributed programming: proof rules and disciplines"*, ACM TOPLAS 6(3), pp. 402-431, 1984.