A WEAKEST PRECONDITION SEMANTICS FOR COMMUNICATING PROCESSES

by

Tzilla Elrad and Nissim Francez

Technical Report #244
May 1982
A WEAKEST PRECONDITION SEMANTICS FOR COMMUNICATING PROCESSES(*)

by

Tzillä Efirad(1) and Nissim Fräncez(2)

Dept. of Computer Science
Technion - Israel Institute of Technology
Haifa 32000, Israel

SUMMARY

A weakest precondition semantics for communicating processes is presented, based on a centralized one level approach. Semantic equations are given for the CSP constructs and their continuity is proved. The representation of various operational concepts, including delay, is discussed. Several examples of applying the rules are given.

Key Words and Concepts:
Weakest precondition, semantics, communicating processes, distributed programming, non-deciderminism, termination, deadlock.

CR Categories: 5.24, 4.32.

(1) The work of the first author was supported by NSF grant MCS-80-17577.
(2) The work of the second author was supported by a grant by IBM-Israel.
(*) A preliminary version of this paper was presented in the 5th International Conference on Programming, Torino, April 1982.
I. INTRODUCTION

The importance of the axiomatic approach to formal definitions of the semantics of languages for concurrent programming is by now widely recognized [OG, AFR, FS, LS, LG].

The purpose of this paper is to investigate the use of WP (Weakest Precondition) semantics as a tool for the formal definition of the semantics of languages for concurrent and distributed programming.

As far as we know, all the previous attempts to use WP semantics were by means of reduction to sequential nondeterminism [LS, FS]. Here we aim at a direct concurrent semantics, preserving the processes structure of the program.

Due to recent developments in the technology of micro-processors, there is an increasing trend towards the use of languages supporting distributed activity involving communication; e.g. CSP [H], PL/TS [FE], Distributed Processes [BH] and, recently, ADA [ADA].

As a model language for our investigation, CSP has been chosen. This language (and the model of concurrency on which it is based) already have been given other formal definitions ([FHLR], [FLP], [CH], [AFR], [CC], [CM], [LG], [P]) and attracted considerable attention.

An important feature of CSP is its emphasis on terminating concurrent programs, as opposed to [MM] or [MI], for example, where non-termination is the rule. This fits nicely with the use of WP which also emphasizes termination.

In the denotational semantics already given for CSP [FHLR], [FLP], [CC], as well as in the various proof rules for partial correctness, no attempt was made to characterize properly terminating programs.

We show that properties like freedom of deadlocks are also naturally reflected in a WP semantics.
Some aspects of CSP which need to be clarified by formal definition of their semantics, are:

a) Stress on simultaneity rather than on mutual exclusion as the synchronization means.

b) The function of the communication primitives of input and output (traditionally known as send and receive) as a choice mechanism and repetition control mechanism. This is an extension of Dijkstra's guarded commands language [D] allowing two kinds of non-deterministic resolutions: local (within a process) and global (among several processes).

c) The distributed termination convention, by means of which the global property of termination (depending on the state of the whole program) is distributed to the various processes. By this convention a process will either terminate by itself or its termination will be induced by other processes with which it communicates.

However, in view of the many semantic definitions of CSP already published (only some of which have been referred to above), one could ask a natural question: why is another semantic definition, though using a method not applied so far, needed? To justify the definition suggested here, we would like to draw attention to the following phenomenon:

In principle, one can envisage two approaches to attributing a semantics to language for concurrent programs. According to the one approach, some a priori semantics is attributed separately to each process, and then those meanings are bound together to yield the semantics of the whole program. This approach was used in [FHLR], [FLP], and [MI], [M] and in the denotational setting and in [OG], [AFR], [17], and [27]. In essential respects, this is...
In previous attempts by Apt, de Roever and Francez (unpublished) this approach was tried in a WP setting, and caused the use of complicated states involving histories. In order to appreciate the difficulty involved, consider the following situation: from a given state $\sigma$, in process $P_1$, two possible continuations are possible. The one involves a communication with a second process, say $P_j$, and if it happens leads to a failing computation. The second involves a communication with a third process, say $P_k$, which leads to successful termination.

Now, the question arises, should the state $\sigma$ be included in the weakest precondition of $P_1$? If it is included, then the environment may choose the "bad" communication, which makes the choice wrong. If it is excluded, again the environment may decide on the "good" communication, again making the choice wrong.

The conclusion is, that the (simple) state does not contain enough information to facilitate a definition of WP of a single process, and extensions of the state to include communication histories (or futures) are needed. Even with such extensions, a proper weakest precondition semantics for a single process could not be found, since any such semantics records only positive information about successful computations, and disregards unsuccessful paths, which still may match with another process' failing path and thus create a global failure.
Here we have chosen the second approach, where a centralized semantics is given directly to the whole program, thereby avoiding the consideration of histories as part of the state but paying the price of giving up full induction on the syntax of the program. We would like to pose the problem of devising a two-levelled WP-semantics for CSP as a challenge to researchers in semantics: is it inherently impossible (as believed by Dijkstra [private communication]) or just needs some new insight, missed in previous attempts?)
II. THE SEMANTIC EQUATIONS

We start with some preliminaries: Let $P := [P_1; \ldots ; P_i; \ldots ; P_n]$ be a program with (variable disjoint) communicating processes $P_i$, $i = 1, \ldots , n$. The symbol "$;$" denotes concurrent composition. We refer the reader unfamiliar with SCP to [H] for an informal and detailed description of the language. Let $Q$ be a predicate over the disjoint union of all the (local) states of the $P_i$'s, to be called the global state.

We denote by $\Lambda$ an empty process (with no instructions). We assume that each non-empty process is structured as $P_i := S_i; P_i'$, with $P_i'$ possibly $\Lambda$. We call $P_i'$ the rest of process $P_i$, and it has a major significance in the definition of the "rest of the (whole) program", a central concept in what follows.

We denote by $\text{SEQ}$ all non-empty sequential program sections, i.e. those sections which contain no communication commands. We assume $\text{WP}[S, Q]$ as known [D] for $S \in \text{SEQ}$.

We denote by $\text{IF}$ all guarded selection statements. For notational convenience, we shall assume that each guard $g$ has two components: a boolean component $b$ (we take $b = \text{true}$ if it is not included) and a communication component $c$ (we take, by convention, $c = \text{skip}$, in case it is not included [LG]). Thus, if $S_i \in \text{IF}$, $S_i$ has the form

$$S_i := [b_i; c_i \rightarrow T_i]$$

Also, $\text{DO}$ denotes the set of all repetitive statements. A subscript in a statement denotes the index of the process to which the statement belongs. For simplicity, we refrain from another index which would
distinguish between different IFs or DOs in the same process.

For convenience, we use the following syntactic (meaning preserving) transformation:

\[
P_j ? x \rightarrow [P_j ? x \rightarrow \text{skip}] \\
P_j ! x \rightarrow [P_j ! x \rightarrow \text{skip}]
\]

and thus have to consider I/O operations only as guards and never as simple commands. We consider this to be a simpler solution (to the problem of the double role of I/O commands) then the distinction between weak and strong guarding as suggested in \[P\].

We also use the (syntactic) predicate \( \mu(c_i,c_j) \) where \( c_i, c_j \) are I/O components of guards:

\[
\mu(c_i,c_j) = \begin{cases} 
\text{true} & \text{if } c_i = [P_j ? x]_i, \; c_j = [P_j ! y]_j, \; \text{type}(x) = \text{type}(y) \\
\text{false otherwise.} & 
\end{cases}
\]

The predicate \( \mu(c_i,c_j) \) means that \( c_i \) and \( c_j \) are syntactically matching communication guards taken from \( P_i \) and \( P_j \) respectively.

Note that this definition hints that messages are strongly typed, and communication is always to named target processes, determined syntactically (at compile time) which are important features of CSP, for an extension of the language to allow dynamic targets, see \[F\]. We use also the notation target\((c)\), where

\[
\text{target}(P_j ? x) = \text{target}(P_j ! y) = j.
\]

Also, by convention, target\((\text{[skip]}_j)\) = \( j \).

We would like to extend the weakest precondition semantics for nondeterministic sequential programs \[D\] to deal with communicating processes. Obviously, we need to add a basic definition of the meaning
of a single communication. We define:

\[ WP[[P_j ? x]_j \parallel [P_i ! y]_j, Q] = Q^x_y, \]

where \( Q^x_y \) is the predicate obtained by substituting \( y \) for all free occurrences of \( x \) in \( Q \). Operationally, this means that a single communication acts as a global assignment, relating variables of two (disjoint) processes. Note that \( WP \) is not defined for non-matching communication guards, as the equations will prevent its application in such cases.

Assume that for some \( 1 \leq \ell \leq n, S_\ell \in SEQ \), so \( S_\ell \) contains no input/output guards. In that case \( S_\ell \) might unconditionally be chosen for execution. An execution of \( S_\ell \) should result in a state which satisfies the weakest precondition for a successful execution of the rest of the program in such a way that \( Q \) will hold at the end.

Thus we must have

\[ WP[P, Q] \Rightarrow WP[S_\ell, Q'] \]

where

\[ Q' = WP[[P_1 \parallel \ldots \parallel P_\ell \parallel \ldots \parallel P_n], Q]. \]

It can be proved [E] that we can define: for some \( 1 \leq \ell \leq n, S_\ell \in SEQ, \)

\[ WP[P, Q] = WP[S_\ell, WP[[P_1 \parallel \ldots \parallel P_\ell \parallel \ldots \parallel P_n], Q]]. \]

i.e. giving preference to sequential program segments preserves the meaning. Recall that \( P'_\ell \) is the rest of process \( P_\ell \), whose first section is sequential (and might be \( \Lambda \)).

But what if there is no \( 1 \leq \ell \leq n: S_\ell \in SEQ \)? Then, the program is in a state in which each of the processes (not yet terminated) is willing to communicate; some of the communication commands serve as iteration guards, and some others as selection guards. In such a situation the
following three properties must hold for any state belonging to the WP:

Property a: Execution of any of the passable (to be defined below) guarded commands will result in a state which satisfies the weakest precondition for successfully executing the rest of the program, with respect to the (same) post-condition Q.

Property b: Any exit from a repetitive statement (loop) should result in a state satisfying the weakest precondition for successfully executing the rest of the program, with respect to Q.

Property c: At least one of the guards is passable, or at least one of the loops must terminate.

None of the three alternatives holding means that the program is in a deadlock-state in which no pair of matching i/o guards exist, nor does a true local (boolean) guard exist. In that case we must have WP[P,Q] = false.

We proceed by giving a formal presentation of properties a, b and c.

Property a: Communication commands as a choice mechanism

Since for all 1 ≤ i ≤ n S_i is either a selection (IF) command or an iteration (DO) command, all S_i's have the form:

\[ S_i:::[b_i^1; c_i^1 \rightarrow T_i^1] \quad \text{or} \quad S_i:::*[b_i^1; c_i^1 \rightarrow T_i^1] \]

\[ \vdots \]

\[ \vdots \]

\[ b_i^n_i; c_i^n_i \rightarrow T_i^n_i \] \quad \text{or} \quad \[ b_i^n_i; c_i^n_i \rightarrow T_i^n_i \]

Our centralized approach enables us to determine (syntactically) all possible matching communications. The predicate \[ \mu(c_i^k, c_j^k) \land b_i^k \land b_j^k \] means that the corresponding communication path can be followed and
\( T^k_1, T^k_j \) may be extended. In this case the guards \( b^k_i \); \( c^k_j \) (in \( S_1 \)) and \( b^{k'}_j \); \( c^{k'}_j \) (in \( S_j \)) are said to be \textit{passable}.

The following will define the meaning of the communication primitives as a choice mechanism and express \textit{global nondeterminism}:

\[
i, j, k, k' : \mu(k^k, c^k_j) \quad b^k_i \land b^{k'}_j \Rightarrow \text{WP}[c^k_i \land c^{k'}_j \land \text{WP}[p^k_i, k', Q]]
\]

whose meaning is: For any passable pair of matching guards, the execution of the communication commands will result in a state satisfying the weakest precondition for successfully executing the rest of the program (denoted by \( p^k_{i,j} \)) with respect to \( Q \). An exact definition of \( p^k_{i,j} \) follows below.

The condition \( (c^k_i = \text{skip}) \land b^k_i \) indicates that \( S_1 \) is ready to choose some guard with no communication request, i.e. the guard \( b^k_i; \text{skip} \) (in \( S_1 \)) is \textit{passable}. Here we have \textit{local nondeterminism}:

\[
i, k : c^k_i = \text{skip} \quad b^k_i \Rightarrow \text{WP}[\text{skip}, \text{WP}[p^k_{i,j}, Q]]
\]

whose meaning is:

For any local nondeterministic choice, passing this guard will successfully end satisfying the weakest precondition of the rest of the program (denoted \( p^k_i \)) with respect to \( Q \).

We will have to define \( p^k_{i,j} \) and \( p^k_i \), which represent the "rest of the program". The ability to consider this "rest of the program" is a major difference from the two-levelled approaches, which considers one process at a time.

Note that so far we did not care whether the i/o guards \( c^k_i \) are taken from an IF command or from a DO command. This distinction is
expressed in the definition of $\tilde{T}^k_i$ below. We let

$$\tilde{T}^k_i = [P_1 \| \ldots \| \tilde{T}^k_i \| \ldots \| \tilde{T}^k_j \| \ldots \| \tilde{T}^k_{j-1} \| \ldots \| P_n],$$

for $1 \leq i < 1 \leq n$, and

$$\tilde{T}^k_i = [P_1 \| \ldots \| \tilde{T}^k_i \| \ldots \| \tilde{T}^k_{j-1} \| \ldots \| P_n],$$

for $1 \leq i \leq n$.

where

$$\tilde{T}^k_i = \begin{cases} \tilde{T}^k_i & S_i \in IF \\ \tilde{T}^k_i ; S_i & S_i \in DO, \end{cases} \quad 1 \leq i \leq n, \quad 1 \leq k \leq n.$$

The definition of $\tilde{T}^k_i$ reflects the meaning of a repetitive statement as once executing the loop body and then again the whole loop, as is usual in definitions using the least fixed point approach. However, the definition reflects also the fact that between two consecutive executions of a loop $S_i$, statement in some $P_j$, $j \neq i$, may be executed.

Property b: The distributed termination convention

A loop $S_i$ terminates either because all the boolean components $b^k_i$ are false or because all the target processes referred to in its guards have terminated, or a combination of both.

The termination condition for a loop $S_i$ (denoted by $LE_i$) can be expressed as follows:

$$LE_i = \bigwedge_{k : \text{target}(c^k_i) \neq \emptyset} \sim b^k_i$$

$LE_i$ means: each of the guards of $S_i$ is either non-passable (having $b^k_i = \text{false}$) or is trying to communicate with a terminated target process.

Property b can be formally expressed as:

$$\bigwedge_{i : S_i \in DO} (LE_i = WP[[P_1 \| \ldots \| P_j \| \ldots \| P_n], Q]).$$
For each of the loops which satisfies \( LE_i \) (i.e. are ready to terminate) the program is in a state which satisfies the weakest precondition for successfully executing the rest of the program with respect to \( Q \).

Here the rest of the program is obtained by deleting \( S_i \) completely from the program.

**Property c: Freedom from deadlock**

The predicate

\[
BB_1 = \bigwedge_{i,j,k,k':\leq}(i,c_i,k') \land b_j^k
\]

means: at least one pair of matching communications is (jointly) passable. Also, the predicate

\[
BB_2 = \bigwedge_{i,k: c_i^k = \text{skip}} b_i^k
\]

means: at least one of the local nondeterministic guards is passable.

Let

\[
BB = BB_1 \lor BB_2
\]

BB means: at least one guard is passable (either a global nondeterministic choice can be made, or some local nondeterministic choice is possible).

In case of \( \neg BB \) we must require that at least one of the loops is ready to terminate, as denoted by:

\[
\bigwedge_{i:S_i \in DO} LE_i
\]

Deadlock freedom can be thus expressed by:

\[
BB \lor \bigwedge_{i:S_i \in DO} LE_i
\]

i.e. either a communication occurs, or a local guard was passed, or,
else, a loop exit occurred. Note that no sequential move $S_i \in \text{SEQ}$ is present, by assumption.

Formal presentation of the semantic equations

We sum up the preceding discussion with the following equations defining WP[P,Q] by cases, as shown in Table 1:

WP EQUATIONS FOR CSP

0) $\Lambda$-rule

$$\text{WP}[\text{All} \ldots \text{Il} \Lambda], Q] = Q$$

1) SEQ-rule

For some $1 \leq i \leq n$, $S_i \in \text{SEQ}$:

$$\text{WP}[P, Q] = \text{WP}[S_i, \text{WP}[[P_1] \ldots \|P_i] \ldots \|P_n], Q]]$$

2) Communication-rule (COM)

For no $1 \leq i \leq n$, $S_i \in \text{SEQ}$:

$$\text{WP}[P, Q] =$$

$$\left( \text{LE}_i \Rightarrow \text{WP}[[P_1] \ldots \|P_i] \ldots \|P_n], Q]] \right)^{\wedge}$$

If BB then

$$\left( \text{CC} \right)$$

$$\left( b_i^k \land b_j^{k'} \Rightarrow \text{WP}[c_i^k \| c_j^{k'}, \text{WP}[P_i^k, J], Q]] \right)^{\wedge}$$

$$\left( \text{NP} \right)$$

else

$$\left( \text{NP} \right)$$
Note: In case there are no loops within the \( S_i \)'s, \( i = 1, \ldots, n \), then
\[
\bigwedge_{i:S_i \in DO} (LE_i \Rightarrow WP[[P_1||\ldots||P_i||\ldots||P_n],Q]) = \phi = \text{true}
\]
and
\[
\bigwedge_{i:S_i \in DO} LE_i = \bigwedge_{\phi} = \text{false}.
\]

Therefore, we obtain:

**IF-Theorem** If for all \( 1 \leq i \leq n \), \( S_i \neq \emptyset \Rightarrow S_i \in IF \),
\[
WP[P,Q] =
\]
\[
\begin{array}{c}
\hat{\wedge} \\
(b_k \land b_j \Rightarrow WP[c_i^k \land c_j^k, WP[P_i^k, Q]])
\end{array}
\]
\[
\begin{array}{c}
i,j,k,k':(c_i^k, c_j^k) \\
\hat{\wedge}
\end{array}
\]
\[
\begin{array}{c}
i,k;c_i^k = \text{skip} \\
\hat{\wedge}
\end{array}
\]

Thus, in case there are no loops the communication-rule is reduced to this expected IF-rule, which is a natural extension of the SEQ-IF rule in [D].

We may also prove the following:

**COM-Theorem** The COM-rule is equivalent to the following rule:

for no \( \varepsilon \), \( 1 \leq \varepsilon \leq n \); \( S_{\varepsilon} \in SEQ \)
\[
WP[P,Q] =
\]
\[
\begin{array}{c}
\text{if} \\
i:S_i \in DO
\end{array}
\]
\[
\begin{array}{c}
LE_i \Rightarrow WP[[P_1||\ldots||P_i||\ldots||P_n],Q] \\
i:S_i \in DO
\end{array}
\]
\[
\begin{array}{c}
\text{else} \\
[BB]
\end{array}
\]
\[
\begin{array}{c}
\hat{\wedge} \\
i,j,k,k':(c_i^k, c_j^k) \\
\hat{\wedge}
\end{array}
\]
\[
\begin{array}{c}
(b_k \land b_j \Rightarrow WP[c_i^k \land c_j^k, WP[P_i^k, Q]])
\end{array}
\]
\[
\begin{array}{c}
i,k;c_i^k = \text{skip} \\
\hat{\wedge}
\end{array}
\]
\[
\begin{array}{c}
(b_k \Rightarrow WP[\text{skip}, WP[P_i^k, Q]])
\end{array}
\]
The operational meaning of this theorem is that in case some loop is terminating, then its termination can be given priority over communications. The reason for this is that a loop exit can only reveal new communication capabilities, but cannot cause any loss of communications. We omit a formal proof, which involves a tedious case analysis.

Note that the inverse rule, delaying loop exits until no communication is possible is not a valid rule, since possible computation sequences might be lost.

Remark. In CSP it is possible for a loop to be in a state in which none of its guards is passable, though the termination condition is false. Thus, we should not expect to have a DO-rule which is a simple extension of the sequential DO-rule. Operationally, we attribute this property to delaying the process containing the loop until some of its guarding communications are enabled. Thus, our semantic equations adequately characterize the notion of delay, which has a highly operational meaning and is hard to capture abstractly.

In order to show that the definition of WP by the above equations is legitimate, we have to establish continuity of the functional defined by the above equations. The functionality of \( \tau \) is:

\[
\tau : (\text{PROG} \to [\text{PRED} \to \text{PRED}]) \to (\text{PROG} \to [\text{PRED} \to \text{PRED}]),
\]

where PROG is the (syntactic) domain of programs, and PRED is the (semantic) domain of predicates (sets of states).

We first prove a lemma which is a slight generalization of standard theorems in denotational semantics, showing the continuity of a certain type of composing continuous functions.
Lemma. Let A, B be domains (cpo's), with typical elements a, b. Let F be a continuous function, F ∈ [A → [B → B]]. Then, the functional G defined by \( G(F) = \lambda a \cdot \lambda b \cdot F(a)(F(b)) \) is continuous.

Proof. Let \( <f_i> \) be a chain in \([A → [B → B]]\).

\[
G(\bigsqcup f_i) = \lambda a \cdot \lambda b \cdot (\bigsqcup f_i) a((\bigsqcup f_i) b)
\]

by the definition of lub of a chain of functions,

\[
= \lambda a \cdot \lambda b \cdot (\bigsqcup f_i) a(\bigsqcup f_j) b
\]

by the continuity of \((\bigsqcup f_i) a\)

\[
= \bigsqcup ((\bigsqcup f_i) a)(f_j b)
\]

by the definition of lub of functions

\[
= \bigsqcup ((\bigsqcup f_i) a)(f_j b)
\]

and again, by the definition of such an lub,

\[
= \bigsqcup \bigsqcup (f_i a)(f_j b)
\]

by lemma 5.4 \([a \cdot b]\)

\[
= \bigsqcup (f_k a)(f_k b)
\]

by the definition of \(G\)

\[
= \bigsqcup G(f_k)
\]

which proves the continuity of \(G\).

By a similar argument, we can prove a slightly stronger result, by defining \(G_i\) as

\[
G(F) = \lambda a \cdot \lambda b \cdot F(a)(F(b))
\]
where \( f, g \) are two (fixed) continuous functions in \([A \to A]\).

From this follows the following:

**Proposition.** The functional \( \tau \) defined by the WP-equations is continuous.

**Proof.** By inspection we can see that the general form of \( \tau \) is, for \( W \in (\text{PROG} \to [\text{PRED} \to \text{PRED}]) \)

\[
\tau(W) = \lambda P \in \text{PROG} \cdot \lambda Q \in \text{PRED} \cdot \bigwedge_1 \text{if } \alpha_i[P] \text{ then } W[\beta_i[P], W[\gamma_i[P], Q]]
\]

where: \( \alpha_i \) are syntactic predicates, e.g. \( \exists i \leq z \leq n, S_i \in \text{SEQ} \);

\( \beta_i, \gamma_i \) are syntactic functions, e.g. computing the rest of the program, or the first sequential component of a process.

Thus, the continuity of \( \tau \) follows from the lemma above.

The semantic equations, as defined above, have an operational meaning of preference of actions according to the following order:

1. sequential (local) transitions;
2. loop exits;
3. communication.

Next, we show that, as far as WP is concerned, this semantics is equivalent to another one, giving equations that do not induce any preferences. It is also shown in [E] that the proposed WP semantics is equivalent to a natural operational semantics. However, if extra liveness properties, such as fairness, are taken into account in the semantic definition, then the definition does become sensitive to preferences of actions.
Non-preference WP-semantics

We now define another transformer, which we denote by \( \hat{\text{WP}} \), in which the above mentioned preferences are not reflected. The equation is given in Table 2.

\[
\hat{\text{WP}}[[P_1; \ldots; P_n], Q] = \begin{cases} \text{if } \bigwedge_{i=1}^{n} P_i = \top \text{ then } Q \\ \text{else} \left\{ \begin{array}{l}
\left( \begin{array}{l}
\hat{\text{WP}}[S_i, \hat{\text{WP}}[[P_1; \ldots; P_i; \ldots; P_n], Q]] \\
\wedge \\
i : S_i \in \text{SEQ} \\
\wedge \\
i : S_i \in \text{DO} \\
\wedge \\
i, j, k, k' : \mu(c_i, c_j) \\
\wedge \\
1, j, k, k' : \mu(c_i, c_j) \wedge b_i \land b_j' \\
\end{array} \right. \\
\end{cases}
\right.
\]

\textbf{Table 2.}

We now prove the following

**Proposition:** For every program \( P \) and predicate \( Q \),

\[ \hat{\text{WP}}[P, Q] = \text{WP}[P, Q]. \]

**Proof.** The inclusion \( \text{WP}[P, Q] \subseteq \hat{\text{WP}}[P, Q] \) is obvious. We proceed in a number of lemmata.

**Lemma 1.** For \( 1 \leq i < j \leq n, S_i \in \text{SEQ}, S_j \in \text{SEQ}, \)

\[ \text{WP}[S_i, \text{WP}[[[P_1; \ldots; P_i; \ldots; P_n], Q]] = \text{WP}[S_j, \text{WP}[[[P_1; \ldots; P_i; \ldots; P_n], Q]]. \]
Proof of Lemma 1.

\[ WP[S_i, WP[[P_i] \ldots ||P_i] \ldots ||P_n], Q] = (SEQ rule) \]

\[ WP[S_i, WP[S_j, WP[[P_j] \ldots ||P_j] \ldots ||P_n], Q]] = (sequential composition rule) \]

\[ WP[S_i \cdot S_j, WP[[P_i] \ldots ||P_i] \ldots ||P_n], Q] = (disjointness of \( S_i, S_j \)) \]

\[ WP[S_j, WP[S_i, WP[[P_i] \ldots ||P_i] \ldots ||P_n], Q]] = (sequential composition rule) \]

\[ WP[S_j, WP[[P_i] \ldots ||P_i] \ldots ||P_n], Q] = (SEQ rule) \]

This establishes Lemma 1, showing the commutativity of sequential program sections in different processes due to variable disjointness.

Lemma 2. For \( 1 \leq i \neq j \leq n \), \( S_i \in \text{SEQ} \), \( S_j \in \text{DO} \) and \( \text{LE}_i \) holds,

\[ WP[S_i, WP[[P_i] \ldots ||P_i] \ldots ||P_n], Q] \]

\[ WP[[P_i] \ldots ||P_j] \ldots ||P_n], Q] \]

Proof of Lemma 2 is similar to that of Lemma 1.

Lemma 3. For \( 1 \leq m \leq n \), \( S_m \in \text{SEQ} \),

\[ i,j,k,k': \mu (c_i, c_j) \rightarrow WP[c_i^{k_i}, WP[c_j^{k_j}, WP[[P_i] \ldots ||P_n], Q]] \]

\[ WP[S_m, WP[[P_i] \ldots ||P_n], Q] \]

(note that equality does not hold in this case).

Proof of Lemma 3. Again a similar argument, by disjointness, \( c_i^{k_i} || c_j^{k_j} \) and \( S_m \) commute.
The equality does not hold, since executing $S_m$ might weaken:

$$i, j, k, k': \mu(c_i^k, c_j^{k'}) b_i^k \land b_j^{k'},$$

because $i = m$ or $j = m$ may now be the case.

Note, while passing, that the converse proposition, based on preferring a communication over a local action, is not true, since the local action might "hide" a communication possibility which is not taken into account.

**Lemma 4.** For $1 \leq i \leq n$, $S_i \in DO$, $LE_i$ holds

$$i, j, k, k': \mu(c_i^k, c_j^{k'}) b_i^k \land b_j^{k'} \Rightarrow WP[c_i^k \parallel c_j^{k'}, WP[P_i^k, P_j^{k'}, Q]]$$

$$= WP[[P_1 \parallel \ldots \parallel P_i^k \parallel \ldots \parallel P_n^k], Q].$$

**Proof of Lemma 4.** Similar to that of Lemma 3. Note that here also the converse does not hold.

Finally, the theorem follows from lemmata 1-4, since many of the conjuncts in the definition of \( WP \) collapse to the same sets of states, as shown by the lemmata.
III. EXAMPLES

We present some small examples of applying the rules. More extensive and complicated examples will appear in [E].

Examples: The first two examples clarify the way the communication-rule deals with loops, while the third is concerned with the distributed termination convention.

a.1 Non terminating loops ("infinite chattering", to use Hoare's expression).

Let \( P ::= [P_1 || P_2] \), where \( P_1 ::= [\text{true}; P_2 ? x \to \text{skip}] \),
and \( P_2 ::= [\text{true}; P_1 ! y \to \text{skip}] \).

Let \( Q = \text{true} \).

\[
WP[[P_1 || P_2], \text{true}] =
\]
using the communication-rule.

\[
\text{(E) false } \Rightarrow WP[[\forall P_2], \text{true}]
\]
\[
\wedge
\]
\[
\text{false } \Rightarrow WP[[P_1 || \lambda], \text{true}]
\]
\[
\wedge
\]

\[
\text{(BB) if true then}
\]
\[
\text{true } \Rightarrow WP[P_2 ? x || P_1 ! y, WP[P_1, \lambda], Q]]
\]
\[
\wedge
\]

\[
\text{(C) else}
\]

\[
\text{(ND) } \sim \text{true } \vee \sim \text{true}.
\]

Simplifying this (with \( \wedge \equiv \text{true} \)) we obtain:

\[
A = WP[P_2 ? x || P_1 ! y, WP[P_1, \lambda], Q]] .
\]

By the definition of \( \sim_{1,2} \) for a DO command:

\[
WP[P_1, \lambda] = WP[[\text{skip}; P_1 || \text{skip}; P_2], Q] .
\]
Using the SEQ-rule twice

\[ WP[[P_1 \parallel P_2], Q] = WP[[P_1, Q]] \]

Substituting the value of \( WP[\widetilde{P_{1,2}}, Q] \) back in \( A \) we obtain:

\[ WP[[P_1 \parallel P_2], Q] = WP[P_2 ? x \parallel P_1 ! y, WP[[P_1 \parallel P_2], Q]]. \]

The least fixed point of this equation is false. So:

\[ WP[[P_1 \parallel P_2], true] \neq false. \]

\[ \text{a.2 Terminating loops} \]

Let \( P := [P_1 \parallel P_2] \) where \( P_1 := \text{[true; \( P_2 ? x \rightarrow \text{skip} \)],} \)

and \( P_2 := [\text{true; } P_1 ! y \rightarrow \text{skip}] \).

Let \( Q = (x = y) \).

\[ WP[[P_1 \parallel P_2], x = y] = \]

Using the communication-rule

\[ (E) \text{ false } \Rightarrow WP[[\text{All } P_2], x = y] \]

\[ \land \]

\[ (BB) \text{ if true } \land \text{ true then} \]

\[ (CC) \text{ true } \land \text{ true } \Rightarrow WP[P_2 ? x \parallel P_1 ! y, WP[\widetilde{P_{1,2}}, Q]] \]

\[ \land \]

\[ (C) \phi \ldots \]

\[ (ND) \text{ else true.} \]

After simplification, we obtain:

\[ = WP[P_2 ? x \parallel P_1 ! y, WP[\widetilde{P_{1,2}}, Q]]. \]

Using the definition of \( \widetilde{P_{1,2}} \):

\[ WP[\widetilde{P_{1,2}}, Q] = WP[[\text{skip}; P_1 \parallel \text{skip}], x = y]. \]

(Note the difference between the IF command and the DO command)
Using the SEQ-rule twice

\[ \text{WP}[P; \text{true}, x = y]. \]

Using again the Communication-rule

\[ \text{WP}[[\text{all}A], x = y]. \]

And by simplification:

\[ \text{WP}[[\text{all}A], x = y]. \]

By using \(-\)-rule, we finally get:

\[ (x = y). \]

Substituting back the value of \( \text{WP}[P_1, Q] \) we obtain

\[ \text{WP}[[P_1 \| P_2], x = y] = \text{WP}[P_2 ? x \text{WP}_1 ! y, x = y]. \]

And by using the Global-assignment-rule and the \(-\)-rule

\[ (x = y)^x_y \equiv \text{true}; \]

so

\[ \text{WP}[[P_1 \| P_2], x = y]. = \text{true}. \]

b. **Distributed termination convention**

Let \( P:: [P_1 \| P_2], \) where

\[ P_1:: *\{x < 10; P_2 ? z \rightarrow x := z\}, \]

\[ P_2:: *\{y := 1; s := 0; \}

\[ *\{P_1 ! y \rightarrow y := y + 1; s := s + 1\}. \]
Also, let \( Q \equiv (s = 10) \).

\[
WP[P, Q] = \quad \text{(by the SEQ-rule)}
\]

\[
WP[y := 1; s := 0, WP[[P_1 \| P_2], s = 10]]
\]

Thus, we compute

\[
WP[[P_1 \| P_2], s = 10] =
\]

\[
\text{if } x \geq 10 \text{ then } WP[[\Lambda \| P_2], s = 10]
\]

\[
\text{else } x < 10 \land \text{true}
\]

\[
\forall (x < 10 \land \text{true}) \Rightarrow WP[[P_2 \| z \| P_1 ! y], WP[\tilde{P}, s = 10]]
\]

(Note that we have omitted the indices of \( \tilde{P} \), which should be clear from the context.)

By simplification of this expression, we obtain:

\[
\text{if } x \geq 10 \text{ then } WP[[\Lambda \| P_2], s = 10]
\]

\[
\text{else } WP[P_2 \| z \| P_1 ! y, WP[\tilde{P}, s = 10]]
\]

To compute the value of this expression, we proceed by considering two subcases:

1. \( x \geq 10 \):

   Since \( E_i \equiv \text{true} \), we get

   \[
   WP[[\Lambda \| P_2], s = 10] = WP[[\Lambda \| \Lambda], s = 10] = (s = 10).
   \]

2. \( x < 0 \):

   We first compute \( WP[\tilde{P}, s = 10] \)

   \[
   = WP[x := z; P_1 \| y := y + 1; s := s + 1; P_2], s = 10]
   \]

   \[
   = WP[x := z; y := y + 1; s := s + 1; WP[[P_1 \| P_2], s = 10]].
   \]

   Solving the recursive equation yields

   \[
   \exists p. P \geq 0 \land y + p \geq 10 \land s + p + 1 = 10.
   \]
By summing up we get:

\[ WP[[P_1;P_2], s=10] = \]

\[
\begin{align*}
\text{if } x &\geq 10 \quad \text{then } s=10 \\
\text{else } \exists p. p \geq 0 \land y+p \geq 10 \land s+p+1=10.
\end{align*}
\]

By substituting back and simplifying, and using the WP-rules for sequential local actions, we finally get

\[ WP[[P_1;P_2], s=10] = (x < 10). \]
IV. CONCLUSION

We have presented a definition of weakest precondition semantics (WP) for communicating processes as expressed in CSP. The approach is a centralized one, where the whole concurrent program is at hand, as opposed to two levelled approaches first defining separate meaning to processes, and then binding the separate meaning to a joint one. The weakest precondition semantics presented is capable of representing delays and deadlocks.

A similar approach was used to define WP semantics for shared variables concurrency using critical regions with the with...when...do construct, which will be reported elsewhere.

We would like to stress again the fact that in both cases of concurrency considered, a centralized approach was needed to facilitate a weakest precondition semantics. We believe this property is pertinent to the relationship between weakest preconditions and concurrency: there does not seem to be any natural method by which the predicate transformer corresponding to a collection of interacting processes will be functionally defined in terms of the transformers corresponding to the processes rather than by the processes themselves. This is a deviation from the sequential semantics, in which induction on the syntax turned always to be appropriate. An interleaving may fail even if its components (separately) succeed...

ACKNOWLEDGEMENT

We are grateful to John C. Reynolds for a continued support and many stimulating discussions that affected both this paper and the first author's Ph.D. thesis. David Gries had several suggestions that improved the presentation and readability.
REFERENCES


