Weakest environment of communicating processes

by ZHOU CHAOCHEN

Institute of Computing Technology, Academia Sinica
Peking, China

ABSTRACT

As is well known, the concept of the weakest precondition has played an important role in sequential programming. In this paper we introduce a similar concept for distributed programming. As far as partial correctness is concerned, given an overall specification of a distributed system and of a designated part of the system, we can find a minimum specification that must be met by the rest of the system in order that the whole system meet the overall specification. This minimum specification is called the weakest environment of the first designated part with respect to the overall specification. In terms of weakest environment, a calculus for the partial correctness of processes with a master-slave communication mechanism is also given.
INTRODUCTION

\( (P \wp R) \) denotes the weakest precondition for partial correctness of program \( P \) with respect to postcondition \( R \). \( (P \wp R) \) identifies the minimum condition that must be satisfied by the machine state before the execution of \( P \) if the machine state after the successful execution of \( P \) is to satisfy \( R \). Weakest precondition is an important idea in sequential programming. It has been used as a semantics of sequential programming languages, as a proving technique for the correctness of sequential programs, and also as a rigorous approach to developing sequential programs.

This paper will define a similar concept for distributed programming. In sequential programming the sequential operator ";" takes a peculiar part in combining program segments into a program. The weakest precondition can be understood in the following way. Given an overall specification \( R \) of a program and a last segment \( P \) of that program, we may ask what is the minimum specification that must be met by the other segment of the program in order that the whole program meet its specification \( R \). This is nothing other than \( (P \wp R) \).

Let \( P \) and \( Q \) be programs, and let \( R \) be an assertion of machine states. Let "\( P \) satisfies \( R \)" stand for "Starting from any initial machine state, if \( P \) terminates, then the resultant machine state satisfies \( R \)." Then we can define \( (P \wp R) \) as follows: \( (P \wp R) \) is an assertion of machine states such that \( (Q; P) \) satisfies \( R \) iff \( Q \) satisfies \( (P \wp R) \).

In a communicating process, a process may be constructed from a group of parallel subprocesses. So the parallel combinator \( || \) can combine subprocesses into a process even as the sequential operator does for sequential programming.

Let \( P \) and \( Q \) be processes. Then \( (P || Q) \) is a process constituted by \( P \) and \( Q \) in parallel. \( P \) and \( Q \) regard each other as the environment within which they fulfill a certain task. Thus a similar question arises: Given a specification \( R \) of the overall task and a subprocess \( P \), can we formulate a minimum condition that must be met by the environment of \( P \) in order that the process constituted by \( P \) and its environment can meet \( R \)? This minimum condition is called the weakest environment and is denoted as

\( (P \wedge R) \)

So \( (P \wedge R) \) is a specification such that \( (P \parallel Q) \) satisfies \( R \) iff \( Q \) satisfies \( (P \wedge R) \).

In the following sections we give an answer to this question.

The programming notation for communicating processes presented in the first section is oriented to the master-slave communication structure. In this structure communication occurs only between a master and its slave. A master may have several slaves, but each slave belongs to only one master. A master may serve a supermaster as its slave, and a slave may employ its own slaves as well. Of course it is not allowed that a slave is also a supermaster of its master in this rank system. The denotational semantics of the notation is given in the same way as it was in Zhou and Hoare.²

In Section 2, process predicate is used as specification language. Process predicate is something like the channel predicate of Zhou and Hoare,² but it uses a process name instead of a channel name. A process name stands for a message sequence that records all messages that have passed so far to (or from) the process of this name.

In Section 3, a formal definition of weakest environment is introduced. Weakest environment identifies the minimum condition a master (or a slave) must meet in order to meet a given overall specification with its designated slave (or its designated master).

A calculus for the partial correctness of communicating processes is also developed in terms of weakest environment in Section 4.

The last section, Section 5, gives a proof of partial correctness in this calculus.

COMMUNICATING PROCESSES

We briefly recall the programming notation of communicating processes and its denotational semantics in this section.

A process is constructed from a group of subprocesses, intercommunicating on a network of master-slave structures. Communication is the atomic action of a process.

For a process there are two different kinds of communications. One occurs between the process and its slaves, and the other one between the process and its master. Since a master can have several slaves, a process calls its slaves by their names. The pair \( s.e \) is used to denote a communication occurring between a process and one of its slaves, where \( s \) is the name of the slave engaged in this communication and \( e \) is the value of the message being passed in this communication. But any process can have only one master; that is, the master of a process is unique to the process. Hence it is not necessary to use names to identify masters. The pair \( \text{master}.e \) is generally used to denote a communication between a process and its master, where \( \text{master} \) is a special name that is different from any slave name, and \( e \) is the value of the message in the communication.

A communication is an atomic action of a process, so a finite sequence of communications is used to record the behavior of a process from its beginning up to some moment in time. Such a sequence of communications is called the trace of the behavior of a process. Finally, a process is defined as the set of all traces of its behavior.
Let us now take a buffer as an example, to show some possible traces of it. This buffer gets messages from its slave, named servant, and provides its master with them:

.servant ----> buffer ----> master

1. < > (empty sequence) records the behavior before it actually evolves.
2. < servant. 3 > is a trace recording the behavior when it has input a message of value 3 from slave.
3. < servant. 3, master 3 > records the behavior when it has input value 3 and then output it.
4. < servant. 3, servant. 4, master. 3, servant. 1, master. 4 > is another possible trace.

The programming notation presented in the following is an applicative language including the basic constructs: output, input, alternation, naming, parallelism, and recursion. The semantics of processes is defined by a function \( \varsigma \) that maps a process into its trace set.

### 1.1 STOP

The process \( \text{STOP} \) is one that communicates neither with its master nor with its slaves; i.e.,

\[
\varsigma[\text{STOP}] = \omega[\{ < > \}]
\]

### 1.2 Output

Let \( z \) be a process name (a slave name or "master") and \( e \) be a message, and let \( P \) be a process. Then \( (z \cdot e \to P) \) is also a process, which just outputs \( e \) to Process \( z \), and then behaves like \( P \); i.e., the possible trace of \( (z \cdot e \to P) \) is headed by \( z \cdot e \), and the rest of the sequence is a trace of \( P \).

So \( \varsigma[z \cdot e \to P] = \omega[\{ < > \} \cup \{ z \cdot e \} \cup \varsigma[P]] \).

### 1.3 Input

Let \( z \) be a process name, \( x \) be a variable, and \( M \) be a message set, and let \( P(x) \) be a process for any \( x \) in \( M \). Then \( (z^?x : M \to P(x)) \) is a process that first inputs any message \( x \) of type \( M \) from process \( z \), and then behaves like \( P(x) \).

Thus

\[
\varsigma[z^?x : M \to P(x)] = \omega[\{ < > \} \cup \{ z^?x \} \cup \varsigma[P(x)]].
\]

### 1.4 Alternation

Let \( P \) and \( Q \) be processes. Then \( (P|Q) \) is a process that behaves either like \( P \) or like \( Q \).

\[
\varsigma[P|Q] = \omega[\varsigma[P] \cup \varsigma[Q]]
\]

### 1.5 Naming

A process may employ another process as a slave by giving a slave name, say \( s \), to the employed process. In this case the named process should be prepared to communicate with its master, which will call it by the name \( s \). Thus, for synchronization, the communication of the named process with its master (which has the form \( m.e \)) must be regarded as the same event as the communication of its master with the named slave (which has the form \( s.e \)). This can be realized by replacing each occurrence of \( m \) by an occurrence of \( s \) in the named process.

Let \( P \) be a process, and let \( s \) be a slave name that does not occur in \( P \). Then \( (s:P) \) is also a process, and

\[
\varsigma[s:P] = \omega[\varsigma[P][s/m]]
\]

where \( m \) is an abbreviation of \( \text{master} \).

### 1.6 Parallelism

Let \( A \) be the set of names of slaves that Process \( P \) wants to communicate with, and let \( B \) be the set of names of slaves that Process \( Q \) wants to communicate with, where \( A \cap B = \emptyset \) (since no slave is allowed to serve more than one master). Suppose that \( P \) employs \( Q \) as its slave by giving \( Q \) a slave name, \( s \), of \( A \). Then \( (P|A^B \cdot s : Q) \) is a new process, which is constructed by \( P \) employing \( Q \) as its slave. This new process has a new list of slave names that it still wants to communicate with: \( (A \cup B) - \{ s \} \). Thus, in \( (P|A^B \cdot s : Q) \), each communication between \( P \) and its slave \( s \) must be synchronized by the communication between \( Q \) and its master. But each possible communication between \( (P|A^B \cdot s : Q) \) and its master or its slaves in \( (A - \{ s \}) \) can be made by \( P \) itself and has nothing to do with \( Q \). The communication between \( (P|A^B \cdot s : Q) \) and its slaves in \( B \), which actually are slaves of \( Q \), is still dominated by \( Q \) and has nothing to do with \( P \). So the traces of \( (P|A^B \cdot s : Q) \), before canceling \( s \) from the slave list, should be

\[
T = \{ t|\epsilon \in (A \cup B \cup \{ m \})^* \} \cup t|\epsilon \in \varsigma[P] \& t|\epsilon \in \varsigma[Q] \}
\]

where \( m \) is an abbreviation of \( \text{master} \), \( X^* \) stands for all the finite sequences of communications prefixed by the process names in \( X \), and \( t \cdot X \) stands for the sequence obtained from \( t \) by canceling all the communications prefixed by a process name out of \( X \). Then canceling \( s \) from the slave list, which \( (P|A^B \cdot s : Q) \) still wants to communicate with, can be realized by

\[
\varsigma[(P|A^B \cdot s : Q)] = \omega[\{ v|\epsilon\} | \epsilon \in T]
\]

where \( v|\epsilon \) is obtained from \( t \) by canceling the communication prefixed by \( s \). Sometime we use \( T|\{ s \} \) to stand for the right-hand side.
1.7 Recursion

A process expression is an expression made by process variables \((p, q, \text{etc.})\) and the previous six operators. Then \(p \triangle F(p)\) is a single recursion, where \(F(p)\) is a process expression including process variable \(p\). The expression \(p[i:S] \triangle F(p)\) is a vector form of mutual recursion, where for any \(i \in S\), \(p[i]\) is a process variable, and \(F(p)[i]\) is a process expression including some of the process variables \(p[j](j \in S)\).

The process (or array of processes) defined by recursion \(p \triangle F(p)\) (or \(p i:S \triangle F(p)\)) is the least fixed point of the recursion, denoted as \(\mu p. F\) (or \(\mu p[i:S]. F\)).

In Zhou and Hoare\(^2\) a partial order of trace sets has been defined by the inclusion relation of sets, and the limit of a monotonic sequence of trace sets has been defined as the infinite union of this sequence:

\[
\lim \delta[P_i] = \delta \cup \delta[P_i]
\]

where \(\forall i. (\delta[P_i] \subseteq \delta[P_i])\).

Furthermore, we have shown that all the operators are continuous. So the least fixed point of recursion can be defined as

\[
\delta[\mu p. F] = \delta \cup \delta[F(\text{STOP})]
\]

and \(\delta[\mu p[i:S]. F[j]] = \delta \cup \delta[F(\text{STOP})][j] (j \in S)\).

In what follows we always use \(p\) as an abbreviation of the least fixed point defined by recursion \(p \triangle F(p)\), where it will not cause confusion.

Example. A Matrix Multiplier

To realize a multiplication for a matrix of three rows by a three-dimensional vector

\[
(v_1, v_2, v_3) \begin{pmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{pmatrix}
\]

it is desirable to input three numbers at a time, and multiply these numbers simultaneously also. So an algorithm is suggested as

\[
((\text{adder} \mid \text{multi}[1]; p[1]) \mid \text{multi}[2]; p[2]) \mid \text{multi}[3]; p[3])
\]

where \(B_1 = \{\text{row}[i]\}^3_{i=1,2,3}\), \(A_1 = \{\text{multi}[1]\}, \text{multi}[2]\},\text{multi}[3]\},\)

\(A_2 = (A_1 \cup B_1) - \{\text{multi}[1]\}\) and \(A_3 = (A_2 \cup B_2) - \{\text{multi}[2]\}\),

and

\[
\text{adder} \triangle \text{multi}[1]?x: \text{NAT} \rightarrow \text{multi}[2]?y: \text{NAT} \rightarrow \text{multi}[3]?z: \text{NAT} \rightarrow \text{master}(x + y + z) \rightarrow \text{adder},
\]

\[
p[i:1..3] \triangle \text{row}[i]?x: \text{NAT} \rightarrow \text{master}(v[i] \times x) \rightarrow p[i].
\]

SPECIFICATIONS

We adopt process predicates as assertions. A process predicate is a predicate with process names as free variables, where a process name denotes the sequence of messages communicated by the process of this name up to some moment in time. For example, let \(u\) and \(v\) be strings, and let \(u \leq v\) mean that \(u\) is an initial segment of \(v\). Then the assertion

\[
\text{master} \equiv \text{servant}
\]

means that the message sequence communicated by a process to its master is a copy of the message sequence communicated by the process to its slave, called \text{servant}. This assertion should always be true of a buffer process that gets messages from its slave, \text{servant}, and provides its master with them; therefore we can take it as a specification of the buffer process.

Let \(P\) be a process and \(R(m, z_1, \ldots, z_n)\) be a process predicate with the free process variables \text{master} and \(z\), \(i = 1, \ldots, n\). Then we define \(P \text{ sat } R\) as meaning that \(R\) is true of any possible behavior of \(P\). Since we use a trace to record the behavior of \(P\), the definition of \(P \text{ sat } R\) is

\[
P \text{ sat } R (m, z_1, \ldots, z_n) = R(\forall \text{master}, t'z_1, \ldots, t'z_n),
\]

where \(tz\) stands for a message sequence that is obtained from \(t\) by leaving out only the communication of process \(z\) and then dropping the process name \(z\) to get a pure message sequence. (Note: \(t'z\) and \(t(z)\) are different. The first is a sequence of messages, whereas the second is a sequence of communications.)

Thus, for example, we can hope that

\[
\text{buffer sat } (\text{master} \leq \text{servant}),
\]

and

\[
\text{multiplier sat } \forall e. (1 \ldots \# m).
\]

where \# is the length of sequence \(t\) and \(e_i\) is the \(i\)th member of sequence \(t\).

One of the disadvantages of the process predicate is that we cannot use it to describe liveness of a process, just as partial correctness of a sequential program cannot deal with termination of a program. So process predicate, too, is only concerned with the partial correctness of a distributed program.

---

From the collection of the Computer History Museum (www.computerhistory.org)
Let \( P \) be a process and \( \{z_1, \ldots, z_n\} \) be the slave list of \( P \). Then what is the most precise specification of \( P \) that we can define in terms of process predicate? This should be nothing but just all the possible message sequences to its master and slaves. Let \( P(m, z_1, \ldots, z_n) \) denote the required process predicate. Then

\[
P(x_0, x_1, \ldots, x_n) = \forall t \exists e \in \mathcal{P}. x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i
\]

This claim can be justified by the following theorem:

Theorem.
If \( P \) is a process of slaves \( \{z_1, \ldots, z_n\} \), and \( R \) is an assertion of process names \( \{m, z_1, \ldots, z_n\} \), then

\[
P sat R \iff \forall x_0, \ldots, x_n \ P(x_0, \ldots, x_n) \Rightarrow R(x_0, \ldots, x_n)
\]

Proof. \((\Rightarrow)\) Suppose \( P sat R \); i.e.,

\[
R(t \upharpoonright m, \ldots, t \upharpoonright z_n)
\]

is true for any \( t \) of \( \mathcal{P} \).

Then

\[
P(x_0, \ldots, x_n) \Rightarrow \exists e \in \mathcal{P}. x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i \quad \text{ (def.)}
\]

\((\Leftarrow)\) Suppose \( \forall x_0, \ldots, x_n \ P(x_0, \ldots, x_n) \Rightarrow R(x_0, \ldots, x_n). \)

Then for any \( e \in \mathcal{P} \), \( R(t \upharpoonright m, \ldots, t \upharpoonright z_n) \) is true by the assumption, since \( P(t \upharpoonright m, \ldots, t \upharpoonright z_n) \) is true; i.e., \((P sat R)\) is true.

We now derive the structural formulas for process specification.

Theorems.

2.1 STOP

STOP is an inactive process, so it always leaves all communications with its master and slaves empty:

\[
STOP(m, Z) = m = < > \land Z = < >
\]

where \( Z \) stands for \( \{z_1, \ldots, z_n\} \), and \( Z = < > \) stands for \& \( z_i = < > \).

Proof.

\[
STOP(x_0, \ldots, x_n) = \exists e \in \mathcal{P}. STOP. x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i \quad \text{ (def.)}
\]

\[
= \exists e \{ < > \}. x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i \quad \text{ (1.1)}
\]

\[
x_0 = < > \land x_i = < >
\]

\[
x_0 = < > \land x_i = < >.
\]

\[
STOP(m, Z) = STOP(x_0, \ldots, x_n) [m/x_0][z_i/x_i] = m = < > \land Z = < >.
\]

2.2 Output

The possible message sequence of \((z!e \rightarrow P)\) is empty (before its evolution), or has value \( e \) prefixed to the sequence of \( z \), whereas the rest is from \( P \).

\[
(z!e \rightarrow P)(m, Z) = STOP(m, Z) \lor (hd(z) = e \land P(m, Z)[tl(z)/z]),
\]

where \( z \in \{m\} \cup Z \), and \( hd(t) \) stands for the first element of \( t \) and \( tl(t) \) is the tail of \( t \); i.e., \( t = hd(t) \land tl(t) \).

Proof.

\[
(z!e \rightarrow P)(x_0, \ldots, x_n) = \exists e \in \mathcal{P} \{ e \rightarrow P \}
\]

\[
x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i \quad \text{ (def.)}
\]

\[
= \exists e \{ < > \} \lor \{ z, e \rightarrow t \in \mathcal{P} \}.
\]

\[
x_0 = t \upharpoonright m \land x_i = t \upharpoonright z_i \quad \text{ (1.2)}
\]

\[
= (x_0 = < > \land x_i = < >) \lor \exists e \in \mathcal{P}.
\]

\[
x_0 = < > \land x_i = < >.
\]

\[
STOP(x_0, \ldots, x_n) \lor \exists e \in \mathcal{P}.
\]

\[
x_0 = t \upharpoonright m
\]

\[
\land x_i = t \upharpoonright z_i \land hd(x_i) = e \land tl(x_i) = t \upharpoonright z_i
\]

\[
= STOP(x_0, \ldots, x_n) \lor (hd(x_i) = e \land P(x_0, \ldots, d(x_i), \ldots, x_n)).
\]

2.3 Input

\((z?x:M \rightarrow P(x))\) is like \((z!e \rightarrow P)\), except that the value prefixed to the sequence \( z \) can be any one of the set \( M \).

\[
(z?x:M \rightarrow P(x))(m, Z) = STOP(m, Z) \lor (hd(z)e = m \land P(hd(z))(m, Z)[tl(z)/z])
\]

where \( z \in \{m\} \cup Z \).

Proof. Omitted; it is similar to the proof of 2.2.

2.4 Alternation

The possible message sequences of \((P | Q)\) are the union of the message sequences of \( P \) and \( Q \). \((P | Q)(m, Z) = P(m, Z) \lor Q(m, Z)\).

Proof. Omitted; it can be immediately obtained from the definition and (1.4).
2.5 Naming

$(s:P)$ is derived from $P$ by replacing $m$ with $s$; i.e.,

$$(s:P)(s, Z) = P(m, Z)[s/m]$$

where $s \in Z$.

Proof. Omitted.

2.6 Parallelism

$(P \parallel s:Q)(m,Z) = P(m,Z)[s 1m]$ and $(P II s:Q)(m,Z) = P(m,A) & Q(s,B)$, where $A \cap B = 0$, $s \in A$, and $Z = (A \cup B) - \{s\}$.

Proof. Suppose $A = \{z_1, \ldots, z_n, s\}$ and $B = \{z_{n+1}, \ldots, z_k\}$.

$$(\Rightarrow) (P \parallel s:Q)(x_o, \ldots, x_n) = \exists t \in T. (P \parallel [s:Q](x_o, \ldots, x_n).$$

2.7 Recursion

The possible message sequences of the least fixed point of $p \triangleleft F(p)$ are the infinite union of the message sequences of $F^\infty$.

$$(\mu p.F)(m,Z) = \exists s. F^\infty(STOP)(m,Z)$$

and

$$(\mu [i:S].F)[j](m,Z) = \exists s. F^\infty(STOP)[j](m,Z) \quad (j \in S)$$

Proof. Omitted, since it can immediately be obtained from the definition and (1.7).

**WEAKEST ENVIRONMENT**

We now have got enough to give a formal definition of weakest environment.

In designing a process with master-slave structure to meet an overall specification, we may choose the design of the master part of the process at first, and then inquire what is the minimum specification that must be met by the slave part of the process in order that the whole process meet the overall specification. The required specification is called the weakest environment of the designed master.

Let $R$ be an overall specification with free process variables $m$ and $Z$, and let $M$ be a process with slaves $A$, and $A - Z = \{s\}$. Let $M \triangleleft R$ denote a process predicate with free process variables $s$ and $Z - A$, which represents the weakest

$x_o = t_1[m_{i=1}^n & x_i = t_1[z_i & y = t_1[s (where $t_1 = t_1[A \cup \{m\}]$)

$y = t_1[m_{i=1}^n & x_i = t_1[z_i & y = t_1[s (where $t_1 = t_1[A \cup \{m\}]$)

$y = t_1[s_{i=1}^n & x_i = t_1[z_i (where $t_1 = t_1[A \cup \{m\}]$)

$y = t_1[s_{i=1}^n & x_i = t_1[z_i (where $t_1 = t_1[A \cup \{m\}]$)
environment of \( M \) with respect to \( R \); i.e., for any process \( S \) with slaves \( Z - A \)

\[
(M \parallel_{A \subset Z - A} s:S) \text{ sat } R \iff (s:S) \text{ sat } (P \text{ we } R) \quad \text{(Figure 1)}.
\]

The formal definition can be introduced from the theorems of Section 2 as follows.

By the theorem in Section 2, since \((M \parallel_{A \subset Z - A} s:S)\) has the
slave list \( Z \),

\[
(M \parallel_{A \subset Z - A} s:S) \text{ sat } R(m, Z) \iff \forall m, Z. (M \parallel_{A \subset Z - A} s:S)(m, Z) \Rightarrow R(m, Z).
\]

But \((M \parallel_{A \subset Z - A} s:S)(m, Z) = \exists s. M(m, A) \& S(s, Z - A)\)
by Theorem 2.6. So

\[
(M \parallel_{A \subset Z - A} s:S) \text{ sat } R(m, Z) = \forall m, Z, (\exists s. M(m, A) \& S(s, Z - A)) \Rightarrow R(m, Z)
\]

\[
= \forall m, s, Z. (M(m, A) \& S(s, Z - A)) \Rightarrow R(m, Z)
\]

(since \( s \) does not occur in \( R \))

\[
= \forall m, s, Z. s, Z - A \Rightarrow (M(m, A) \Rightarrow R(m, Z))
\]

\[
(C \Rightarrow (B \Rightarrow D)) = (C \Rightarrow (B \Rightarrow D))
\]

\[
= \forall s, Z - A. S(s, Z - A) \Rightarrow \forall m, A \cap Z.
\]

\[
M(m, A) \Rightarrow R(m, Z)
\]

(since \( A \cap Z \) does not occur in \( S \))

\[
= s:S \text{ sat } (\forall m, A \cap Z. M(m, A) \Rightarrow R(m, Z))
\]

(since Theorem in Section 2 and 2.6).

Thus \((M \parallel_{A \subset Z - A} s:S)\) can be suggested to be

\[
\forall m, A \cap Z. M(m, A) \Rightarrow R(m, Z).
\]

Similarly, if we design the slave part of a process first, then we can inquire about the weakest environment of the slave with respect to an overall specification.

Let \( R \) be an overall specification with free process variables \( m \) and \( Z \), and let \( S \) be a process with slaves \( B \), and \( A = (Z - B) \cup \{ s \} \). Let \((s:S \text{ we } R)\) denote the weakest environment with free variables \( m \) and \( A \) such that for any process \( M \) with slaves \( A \)

\[
(M \parallel_{AB} s:S) \text{ sat } R \iff M \text{ sat } (s:S \text{ we } R) \quad \text{(Figure 2)}.
\]

Similar reasoning applies; we can therefore suggest

\((s:S \text{ we } R)\) as \( \forall B \cap Z. S(s, B) \Rightarrow R(m, Z) \).

In general, for any process \( P \) with process names \( X \) and any assertion \( R \) with process variables \( Y \), we define the weakest environment of \( P \) with respect to \( R \) as

\[
P \text{ we } R = \forall X \cap Y. P(X) \Rightarrow R(Y).
\]

So \( P \text{ we } R \) is a predicate of process variables \( X \cap Y \), where \( X \cap Y = (X \cup Y) - (X \cap Y) \).

If \( M \) is a process with process names \( m \) and \( A \), and \( R \) is an assertion with process variables \( m \) and \( Z \), then

\[
M \text{ we } R
\]

\[
= \forall (\{ m \} \cup A) \cap (\{ m \} \cup Z). \ (M(m, A) \Rightarrow R(m, Z)
\]

\[
= \forall m, A \cap Z. \ (M(m, A) \Rightarrow R(m, Z)
\]

as suggested.

If \((s:S)\) is a process, where \( S \) has process names \( m \) and \( B \), and \( s \notin B \); \( R \) is an assertion with process names \( m \) and \( Z \); and \( A = (Z - B) \cup \{ s \} \), then \((s:S)\) has process names \( \{ s \} \cup B \), and

\[
(s:S) \text{ we } R
\]

\[
= \forall (\{ s \} \cup B) \cap (\{ m \} \cup Z). \ (s:S)(s, B) \Rightarrow R(m, Z)
\]

\[
= \forall B \cap Z. \ S(s, B) \Rightarrow R(m, Z)
\]

as suggested also.

Hence the required theorem follows:

Theorem 2.

\[
(M \parallel_{A \cap Z - A} s:S) \text{ sat } R(m, Z)
\]

iff (1) \( M \text{ sat } (S \text{ we } R) \)

or (2) \( (s:S) \text{ sat } (M \text{ we } R) \).

where \( Z = (A - \{ s \}) \cup B \).

Proof. As given above.

The intention to develop a calculus of the partial correctness of communicating processes in terms of weakest environment is based on the fact that a process satisfies an assertion iff the weakest environment of the process with respect to the assertion is a tautology.

Theorem 2.

If \( P \) is a process with process names \( X \) and \( R \) is an assertion of process variables \( X \), then

\[
P \text{ sat } R \iff (P \text{ we } R) = \text{ true}
\]

Proof.

\[
P \text{ we } R = \forall X. \ P(X) \Rightarrow R(X) \quad \text{(def.)}
\]

\[
= P \text{ sat } R \quad \text{(Theorem of 2)}
\]
PROOF RULES

We now develop a set of structural rules for proving the theorems of the form of \( P \text{ we } R \). The validity of the proof rules is established by proving that each of them is consistent with the semantics.

In what follows we use \( X \) to represent the process names mentioned by process \( P \), and \( Y \) to represent the process variables of assertion \( R \).

4.0 General Rules (Healthiness Conditions)

(a) \( (P \text{ we true}) = \text{true} \)
(b) \( P \text{ we false} = \text{false} \), provided \( P \) and false have same process names; i.e. \( X = Y \)
(c) If \( R \Rightarrow T \) is a theorem, then \( (P \text{ we } R) \Rightarrow (P \text{ we } T) \)
(d) \( (\forall y. P \text{ we } R(y)) = P \text{ we } \forall y. R(y) \),

provided \( y \) is not a process variable and \( y \) does not occur in \( P \) freely.

The proofs for their consistency:

(a) \( P \text{ we true} = \forall X \text{ we } P(X) \Rightarrow \text{true} \)
(b) \( P \text{ we false} = \forall X \text{ we } P(X) \Rightarrow \text{false} \)
(c) \( P \text{ we } R \Rightarrow \forall X \text{ we } (P(X) \Rightarrow R(Y)) \)
(d) \( P \text{ we } R \Rightarrow \forall X \text{ we } (P(X) \Rightarrow R(Y)) \)

Since for any process \( P \), the empty sequence is always one of its traces, i.e., \( P(\text{< >}) = P \),

\( P \text{ we false} = \forall X \text{ we } P(X) \Rightarrow \text{true} \)

Actually, even if they mention different process names,

\( (P \text{ we false})(\text{< >}) \)

\( = (\forall X \text{ we } P(X))[\text{< >}/X-Y] \)

\( = \text{false} \). Hence, in any case no process can satisfy \( (P \text{ we false}) \). In this sense \( (P \text{ we false}) \) is always equivalent to false.

(c) \( P \text{ we } R \Rightarrow \forall X \text{ we } P(X) \Rightarrow R(Y) \)
(d) \( P \text{ we } R \Rightarrow \forall X \text{ we } (P(X) \Rightarrow T(Y)) \)

\( \Rightarrow \forall X \text{ we } P(X) \Rightarrow \text{false} \) \( \text{if } R \Rightarrow T \) (since \( R \Rightarrow T \))

\( \forall y. P \text{ we } R(y) = \forall y. \forall X \text{ we } P(X) \Rightarrow R(y, Y) \)

\( = \forall X \text{ we } P(X) \Rightarrow \forall y. R(y, Y) \)

\( = \text{false} \) \( \text{if } y \) is not free in \( P \)

4.1 STOP

\( \text{STOP we } R = \forall X \text{ we } R(Y) \).

The proof for the consistency:

\( \text{STOP we } R = \forall X \Rightarrow \text{false} \)

\( \Rightarrow \forall X \Rightarrow \text{false} \)

\( \Rightarrow \forall X \Rightarrow \text{true} \)

\( \Rightarrow \text{true} \).
where both STOP and \( P(x) \) still have process names \( X \).

The proofs for their consistency are omitted, since these are similar to the proofs of (4.2).

4.4 Alternation

\[
(P | Q) \text{ we } R = P \text{ we } R & Q \text{ we } R,
\]

where \( P, Q, P, \) and \( Q \) mention the same process names.

The proof for the consistence:

\[
LHS = \forall X \land Y. (P | Q)(X) \Rightarrow R(Y) \quad \text{(def.)}
\]
\[
= \forall X \land Y. (P(X) \lor Q(X) \Rightarrow R(Y) \quad \text{(2.4)}
\]
\[
= \forall X \land Y. (P(X) \Rightarrow R(Y)) & (Q(X) \Rightarrow R(Y))
\]
\[
= RHS \quad \text{(def.)}
\]

4.5 Naming

If \( s \in X - Y \) and \( m \in X \), then

\[
(s:P) \text{ we } R = (P \text{ we } R)[z/m][s/m][m/z]
\]

where \( z \in X \cup Y \) and \( P \) has the set of process names \( X[m/s] \).

The proof of the consistence:

\[
LHS = \forall X \land Y. (s:P)(X) \Rightarrow R(Y) \quad \text{(def.)}
\]
\[
= \forall X \land Y. P[X[m/s]] \Rightarrow R(Y) \quad \text{(2.5)}
\]
\[
= \forall X \land Y. P(X[m/s]) \Rightarrow R(Y) \quad \text{(since } X \land Y = X[m/s] \land Y[z/m])
\]
\[
= \forall X \land Y. P(X[m/s]) \Rightarrow R(Y[z/m])
\]
\[
= \forall X \land Y. P(X[m/s]) \Rightarrow R(Y[z/m]) \quad \text{(def.)}
\]
\[
= \forall X \land Y. R(Y[z/m]) \quad \text{(since } z \in Y, m \in X \text{ and } z \in X \cup Y.)
\]
\[
= RHS \quad \text{(def.)}
\]

4.6 Parallelism

\[
(P \parallel s:Q) \text{ we } R
\]

\[
= P \text{ we } (s:Q \text{ we } R)
\]
\[
= s:Q \text{ we } (P \text{ we } R),
\]

where \( P \) mentions process names \( [m] \cup A \), and \( s:Q \) mentions \( [s] \cup B \), and \( A \cap B = 0 \) and \( s \in X \).

Prove its consistency. Let \( X = (A - [s]) \cup B \cup [m] \), which is the set of process names of \( P \parallel s:Q \). Then

\[
LHS = \forall X \land Y. (P \parallel s:Q)(X) \Rightarrow R(Y) \quad \text{(def.)}
\]
\[
= \forall X \land Y. (\exists s.P(m, A) & Q(s, B) \Rightarrow R(Y) \quad \text{(2.6)}
\]
\[
= \forall X \land Y. (s \in Y, s. P(m, A) & Q(s, B) \Rightarrow R(Y)) \quad (s \in Y)
\]
\[
= \forall X \land Y. (s \in Y, s. P(m, A) \Rightarrow Q(s, B) \Rightarrow R(Y))
\]
\[
= \forall (s) & \in (s) \cup B \cap Y. Q(s, B) \Rightarrow R(Y) \quad \text{(since } (s) \cup B \cap Y = \emptyset)
\]
\[
= P \text{ we } (s:Q \text{ we } R)
\]

We can prove the other equivalence similarly.

Note that this rule shows that the weakest environment satisfies the usual axiom of composition as well as the weakest precondition.

4.7 Recursion

\[
\mu p. F \text{ we } R = \forall n. F^*(STOP) \text{ we } R
\]

and \( (\mu p[i:S].F)[j] \text{ we } R[j] = \forall n. F^*(STOP)[j] \text{ we } R[j] \in S) \), where \( \mu p.F \) and \( F^*(STOP) \) have same process names.

The proof of the consistence is only given for single recursion:

\[
LHS = \forall X \land Z. (\mu p. F)(X) \Rightarrow R(Z) \quad \text{(def.)}
\]
\[
= \forall X \land Z. (\exists n. F^*(STOP)(X) \Rightarrow R(Z) \quad \text{(2.7)}
\]
\[
= \forall n. F^*(STOP) \text{ we } R \quad \text{(since } n \text{ does not occur in } R)
\]

From (4.7) a useful structural induction rule follows:

4.7.1 If

\[
T \Rightarrow (STOP \text{ we } R)
\]

and for any process \( P \)

\[
(T \Rightarrow (P \text{ we } R)) \Rightarrow (T \Rightarrow (F(P) \text{ we } R)),
\]

then

\[
T \Rightarrow (\mu p. F \text{ we } R),
\]

where \( STOP, P, \) and \( F(P) \) are supposed to have the same process names as \( \mu p. F \).

The proof of the consistence for this rule can be obtained as follows:

Start at \( T \Rightarrow (STOP \text{ we } R) \), and repeat to use \( (T \Rightarrow (P \text{ we } R)) \Rightarrow (T \Rightarrow (F(P) \text{ we } R)) \). Then we can get for any \( n \)

\[
T \Rightarrow (F^*(STOP) \text{ we } R).
\]

Thus \( T \Rightarrow (\forall n. F^*(STOP) \text{ we } R). \) Hence \( T \Rightarrow (\mu p. F \text{ we } R) \) by (4.7).

EXAMPLE

We end this paper by showing a proof of the partial correctness of the matrix multiplier (see Section 1). We want to prove

\[
\text{multiplier sat } i \quad 1 \ldots m \quad \text{ sat } \frac{3}{m_j} \quad j \quad \text{row j,}
\]

\[
\frac{3}{j} \quad \# \quad m \quad \# \quad \text{row j}
\]

By Theorem 2 of Section 4 this proposition is equivalent to
Lemma 1.

\( \forall i \in \{1 \ldots \# m\}, m_i = \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j], \)
\( \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j] = \text{true}, \)

where multiplier is a process with process names \( m \) and row \( \{j:1 \ldots 3\} \).

Since

\[
\text{multiplier} \; \triangleq \; (((\text{adder} \; \triangleq \; \text{mult}[1];p[1]) \; \| \; \text{mult}[2];p[2]) \; \| \; \text{mult}[3];p[3]),
\]

multiplier we \( \forall i \in \{1 \ldots \# m\}, m_i = \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j], \)
\( \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j] = \text{true} \) and (Parallelism).

We now need four lemmas.

Lemma 2.

\( \forall i \in \{1 \ldots \# m\}, m_i = \frac{\sum_{j=1}^{\# m} v[j] \cdot \text{row}[j]}{\# m} \)
\( \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j] = \text{true} \) \( \triangleq \text{row}[j] \).

Lemma 3.

\( \forall i \in \{1 \ldots \# m\}, m_i = \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j], \)
\( \sum_{j=1}^{\# m} v[j] \cdot \text{row}[j] = \text{true} \) by (Parallelism).

We now only present the proof of Lemma 4, and let \( \text{R} \) stand for the assertion on the righthand side of the previous weakest environment.

Proof.

Adder is an abbreviation for the least fixed point of the recursion

\[ \text{adder} \; \triangleq \; \text{mult} \; \triangleq \; \text{adder}. \]

and the process names which occur in adder are \( X = \{\text{mult}[1], \text{mult}[2], \text{mult}[3], m\}. \)

Let us use (4.7.1) to prove this lemma.

For (STOP \( \text{R} = \text{true} \)) it is trivial, since

\[ \text{STOP} \; \text{R} = \forall X, \text{m} = < > \; \sum_{j=1}^{\# m} \text{mult}[j], \text{STOP} \; \text{R} = < > \; \sum_{j=1}^{\# m} \text{mult}[j] = 0 \] (4.1)

Now assume \( (P \; \text{R}) = \text{true} \). Then

\[ \text{Stop} \; \text{R} = \forall X, \text{m} = < > \; \sum_{j=1}^{\# m} \text{mult}[j], \text{Stop} \; \text{R} = < > \; \sum_{j=1}^{\# m} \text{mult}[j] = 0 \] (4.1)

But

\[ \text{Stop} \; \text{R} = \forall X, \text{m} = < > \; \sum_{j=1}^{\# m} \text{mult}[j], \text{Stop} \; \text{R} = < > \; \sum_{j=1}^{\# m} \text{mult}[j] = 0 \] (4.1)

and hence the previous proposition is equivalent to truth as required.
ACKNOWLEDGMENTS

I am deeply grateful to Tony Hoare. The original idea of weakest environment appears in a joint work with him on the partial correctness of communication protocols, and the suggestion for writing this paper also came from him.

I also thank the Institute of Computing Technology, Chinese Academy of Sciences, for awarding me two years' stay in the United Kingdom, which made possible the cooperative works I published with British scholars.

REFERENCES