# A Proof System for Asynchronously Communicating Deterministic Processes *

F.S. de Boer and M. van Hulst

Utrecht University
Dept. of Comp. Sc.
P.O. Box 80089
3508 TB Utrecht, The Netherlands

**Abstract**

We introduce in this paper new communication and synchronization constructs which allow deterministic processes, communicating asynchronously via unbounded FIFO buffers, to cope with an indeterminate environment. We develop for the resulting parallel programming language, which subsumes deterministic dataflow, a simple compositional proof system. Reasoning about communication and synchronization is formalized in terms of input/output variables which record for each buffer the sequence of values sent and received. These input/output variables provide an abstraction of the usual notion of history variables which denote sequences of communication events. History variables are in general necessary for compositional reasoning about the correctness of distributed systems composed of non-deterministic processes.

## 1 Introduction

Hoare logics have been used successfully for reasoning about correctness of a variety of distributed systems [OG76, AFdR80, ZdRvEB85, Pan88, HdR86]. In general, proof systems for distributed systems based on some kind of Hoare logic formalize reasoning about communication and synchronization in terms of sequences of communication events called *histories*.

Distributed systems based on synchronous communication allow an elegant compositional proof theory [Zwi88] essentially because there exists a simple criterion for deciding when the local histories of the processes of a system are compatible, that is, can be combined into a global history of the entire system. This criterion consists of checking whether the local histories can be obtained as some kind of projection of some global history.

On the other hand distributed systems based on asynchronous communication do not allow such a simple criterion: To check the compatibility of the local histories one has in general to consider all possible interleavings [Pan88]. As such its logical formulation will involve quantification over histories, and this will obviously complicate the reasoning process.

The recent book on program correctness by Francez [Fra92] contains a section on *non-deterministic* processes which communicate asynchronously via FIFO buffers, featuring a proof system that uses a logic based on *input/output variables* instead of histories. A buffer is logically represented by an

*Part of this work is published in the Proceedings of Mathematical Foundations of Computer Science 1994 (MFCS'94), volume 841 of Lecture Notes in Computer Science

input variable which records the sequence of values read from the buffer and by an output variable which records the sequence of values sent to the buffer. The difference between input/output variables and histories is that in the former information of the relative ordering of communication events on different buffers is lost. However, it can be shown that this logic is incomplete for non-deterministic processes; the information expressible by input/output variables only is insufficient to obtain a complete specification of an entire system by composing the local specifications of its constituent processes.

The main contribution of this paper consists of showing that distributed systems composed of *deterministic* processes which communicate asynchronously via (unbounded) FIFO buffers, however do allow a simple complete compositional proof theory based on input/output variables. In order to endow a deterministic process with the capability of responding to an indeterminate environment we introduce communication and synchronization constructs which allow a process to test the contents of a buffer and to synchronize on a set of input buffers simultaneously. The resulting programming language subsumes deterministic dataflow. Thus despite the restriction to deterministic processes we obtain a powerful parallel programming language which still allows a simple compositional proof theory based on input/output variables.

The rest of this paper is organized as follows: In section 2, the programming language is defined. Then, in section 3, an operational semantics and a definition of correctness formulas and their semantics is given. In section 4, the proof system is presented, followed by the proofs of soundness (section 5) and completeness (section 6). Section 7 discusses an extension of the language which provides a process with the full means to cope with an indeterminate environment. Finally, section 8 contains some concluding remarks and observations.

## 2 The programming language

In this section, we define the syntax of the programming language. The language describes the behaviour of asynchronously communicating deterministic sequential processes. Processes interact only via communication channels which are implemented by (unbounded) FIFO-buffers. A process can send a value along a channel or it can input a value from a channel. The value sent will be appended to the buffer, whereas reading a value from a buffer consists of retrieving its first element. Thus the values will be read in the order in which they have been sent. A process will be suspended when it tries to read a value from an empty buffer. Since buffers are assumed to be unbounded sending values can always take place. Additionally we introduce constructs which allow testing whether a buffer is empty or not.

We assume given a set of program variables *Var*, with typical elements $x, y, \ldots$. Channels are denoted by $c, d, \ldots$.

**Definition 1** *The syntax of a statement $S$ which describes the behaviour of a (deterministic) sequential process, is defined by*

$$
\begin{array}{rcl}
S & ::= & \mathsf{skip} \\
 & \mid & x := e \\
 & \mid & c??x \mid c!!e \\
 & \mid & S_1; S_2 \\
 & \mid & \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \\
 & \mid & \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} \\
 & \mid & \mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \\
 & \mid & \mathsf{while}\ c??x\ \mathsf{do}\ S\ \mathsf{od} \\
 & \mid & \mathsf{repeat}\ S\ \mathsf{until}\ c??x
\end{array}
$$

In the above definition skip denotes the 'empty' statement. In the assignment statement $x := e$ we restrict for technical convenience to arithmetical expressions $e$. Sending a value of an (arithmetical) expression $e$ along channel $c$ is described by $c!!e$, whereas storing a value read from a channel $c$ in a variable $x$ is described by $c??x$. The execution of $c??x$ is suspended in case the corresponding buffer is empty. Furthermore we have the usual sequential control structures of sequential composition, choice and iteration ($b$ denotes a boolean expression). Additionally we allow as tests in the choice and while construct an input statement $c??x$. The execution of a statement if $c??x$ then $S_1$ else $S_2$ fi consists of reading a value from channel $c$, in case its corresponding buffer is non-empty, storing it in $x$ and proceeding subsequently with $S_1$. In case the buffer is empty control moves on to $S_2$. The execution of a statement while $c??x$ do $S$ od consists of alternatingly reading a value from channel $c$ and executing $S$ until the corresponding buffer is empty. Finally repeat $S$ until $c??x$ models a form of busy waiting: repeat $S$ for as long as no value can be read from channel $c$. Note that $c??x$ is equivalent to repeat skip until $c??x$, this corresponds to the 'idle waiting' inherent in $c??x$. To resolve possible ambiguities in the grammar we assign to sequential composition the lowest binding priority.

**Definition 2** *A parallel program $P$ is of the form $[S_1 \parallel ... \parallel S_n]$, where we assume the following restrictions: the statements $S_i$ do not share program variables, channels are unidirectional and connect exactly one sender and one receiver.*

# 3 Semantics

In this section we define the operational semantics of the programming language and an appropriate notion of program correctness.

First we need to define the notion of *state* which assigns values to program variables and associates a FIFO buffer to each channel. For the formal justification of the compositional proof system it will appear to be convenient to introduce for each channel $c$ variables $c??$ and $c!!$ which record the sequence of values read from channel $c$ and the sequence of values sent along $c$. The values read from a channel will also include a special value $\perp$ which results from testing an empty buffer. For example a sequence $\langle 1, 2, 3, \perp, 4, 5 \rangle$ representing the values read from a channel indicates that after $1, 2$ and 3 have been read the process tested the contents of the buffer when it was empty. Subsequent read operations on the channel resulted in the values 4 and 5. A variable $c??$ ($c!!$) is also called an input (output) variable. We denote the set of variables $c??$ and $c!!$ by $IO$.

**Definition 3** *Restricting ourselves to the domain of values consisting of integers only, denoted by $\mathbb{Z}$, the set of states $\Sigma$, with typical element $\sigma$, is defined as $\Sigma = \langle Var \rightarrow \mathbb{Z}, IO \rightarrow \mathbb{Z}_\perp^\star \rangle$*

In the above definition $\mathbb{Z}_\perp^\star$ denotes all finite sequences of elements of the set $\mathbb{Z}_\perp = \mathbb{Z} \cup \{\perp\}$. We introduce the following operations on sequences. The empty sequence is denoted by $\epsilon$. Given a sequence $s \in \mathbb{Z}_\perp^\star$, its first element will be denoted by $f(s)$, and the subsequence of $s$ consisting of elements of $\mathbb{Z}$ only we denote by $r_\perp(s)$. The result of appending an element $d$ to a sequence $s$ is denoted by $s \cdot d$. We define $s \prec s'$ iff $s$ is a prefix of $s'$. By $s' - s$ we denote the suffix of $s'$ determined by its prefix given by $s$ (so it is defined only if $s \prec s'$). The buffer corresponding to a channel $c$ in a state $\sigma$, that is, the sequence of values sent along $c$ but not yet read, which we denote by $\sigma(c)$, is given by $\sigma(c!!) - r_\perp(\sigma(c??))$. For example, if $\sigma(c!!) = \langle 1, 2, 3 \rangle$ and $\sigma(c??) = \langle 1, \perp, 2 \rangle$ then $\sigma(c) = \langle 3 \rangle$.

The value of a variable $v$, which might be either a program variable $x$ or an input/output variable $c??$, $c!!$ in a state $\sigma$ will be simply denoted by $\sigma(v)$. Given a state $\sigma$, a variable $v$ and a value $d$ (of corresponding type), we define the state $\sigma\{d/v\}$ as follows:

$$\sigma\{d/v\}(v') = \begin{cases} \sigma(v') & \text{if } v \neq v' \\ d & \text{otherwise} \end{cases}$$

The value of an arithmetical expression $e$ (boolean expression $b$) we denote by $\sigma(e)$ ($\sigma(b)$).

**Definition 4** *A* configuration *is a pair* $\langle [S_1 \parallel ... \parallel S_n], \sigma\rangle$, *where* $S_i$ *is either a statement or equals* $E$ *which denotes termination.*

We now define a transition relation $\rightarrow$ between configurations. For convenience, we identify the statements $S$ and $S; E$ for any statement $S$.

**Definition 5** *The relation* $\rightarrow$ *between configurations is the smallest relation satisfying:*

1. $\langle [... \parallel \mathsf{skip}; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S \parallel ...], \sigma\rangle$
2. $\langle [... \parallel x := e; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S \parallel ...], \sigma\{\sigma(e)/x\}\rangle$
3. $\langle [... \parallel c!!e; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S \parallel ...], \sigma\{\sigma(c!!) \cdot \sigma(e)/c!!\}\rangle$
4. $\langle [... \parallel c??x; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S \parallel ...], \sigma\{\sigma(c??) \cdot d/c??, d/x\}\rangle$,
   *provided* $\sigma(c) \neq \epsilon$ *and* $d = f(\sigma(c))$.
5. $\langle [... \parallel \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S_1; S \parallel ...], \sigma\rangle$,
   *provided* $\sigma(b) = true$.
6. $\langle [... \parallel \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S_2; S \parallel ...], \sigma\rangle$,
   *provided* $\sigma(b) = false$.
7. $\langle [... \parallel \mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S_1; S \parallel ...], \sigma\{\sigma(c??) \cdot d/c??, d/x\}\rangle$,
   *provided* $\sigma(c) \neq \epsilon$ *and* $d = f(\sigma(c)))$.
8. $\langle [... \parallel \mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}; S \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S_2; S \parallel ...], \sigma\{\sigma(c??) \cdot \bot/c??\}\rangle$,
   *provided* $\sigma(c) = \epsilon$.
9. $\langle [... \parallel \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S; \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\rangle$,
   *provided* $\sigma(b) = true$.
10. $\langle [... \parallel \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S' \parallel ...], \sigma\rangle$,
    *provided* $\sigma(b) = false$.
11. $\langle [... \parallel \mathsf{while}\ c??x\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S; \mathsf{while}\ c??x\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\{\sigma(c??) \cdot d/c??, d/x\}\rangle$,
    *provided* $\sigma(c) \neq \epsilon$ *and* $d = f(\sigma(c))$.
12. $\langle [... \parallel \mathsf{while}\ c??x\ \mathsf{do}\ S\ \mathsf{od}; S' \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S' \parallel ...], \sigma\{\sigma(c??) \cdot \bot/c??\}\rangle$,
    *provided* $\sigma(c) = \epsilon$.
13. $\langle [... \parallel \mathsf{repeat}\ S\ \mathsf{until}\ c??x; S' \parallel ...], \sigma\rangle \rightarrow \langle [... \parallel S; \mathsf{if}\ c??x\ \mathsf{then\ skip\ else\ repeat}\ S\ \mathsf{until}\ c??x\ \mathsf{fi}; S' \parallel ...], \sigma\}\rangle$

It is worthwhile noticing that, although not strictly necessary for the definition of the operational semantics, recording tests on an empty buffer allow one to determine the local behaviour of a process. Consider the following two statements: $S = \mathsf{if}\ c??x\ \mathsf{then}\ d!!0; R_1\ \mathsf{else}\ d!!0; c??x; R_2\ \mathsf{fi}$ and $S' = \mathsf{if}\ d??y\ \mathsf{then}\ c!!0; R_1'\ \mathsf{else}\ c!!0; d??y; R_2'\ \mathsf{fi}$. In case one does not record tests on an empty buffer we cannot determine on the basis of the values of the input/output variables whether $R_1$ or $R_2$ (and, symmetrically, $R'_1$ or $R'_2$) is reached in a computation of $[S \parallel S']$, which starts with empty buffers $c$ and $d$. This observation will play an important role in the completeness proof of the proof system.

**Definition 6** *A* computation sequence *of a program* $P = [S_1 \parallel \ldots \parallel S_n]$ *is a finite or infinite sequence* $C_0 \rightarrow C_1 \rightarrow \ldots$ *of configurations such that* $C_0 = \langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle$ *for some state* $\sigma$*.*

We are now ready to define the semantics of a program:

**Definition 7** *The semantics* $[\![P]\!](\sigma)$ *of program* $P$ *in state* $\sigma$ *is defined as* $\{CS \mid CS$ *is a computation sequence of* $P$*, and the state-component of the first configuration of* $CS$ *equals* $\sigma\}$*.*

**Definition 8** *A computation sequence of program* $P$ *is* terminating *iff it is finite and its last configuration is* $\langle [E \parallel \ldots \parallel E], \sigma \rangle$*, for some state* $\sigma$*.*

To reason about the correctness of a program we assume given some first-order logic (so we allow only quantification over *Var*, not over *IO*) to describe properties of states. Thus the vocabulary of the logic includes the standard arithmetical operations and relations. Additionally we assume the logic to include operations and relations on sequences like append, prefixing, etc. Sequence terms of the logic are then constructed from the basic sequence terms $c??$ and $c!!$ using the included repertoire of sequence operations. The sequence term $c$ representing the buffer associated with channel $c$ we introduce as an abbreviation of the term $c!! - r_{\perp}(c??)$. The truth of an assertion $\phi$ in a state $\sigma$ is denoted by $\sigma \models \phi$. For example, $\sigma \models c?? \prec c!!$ iff $\sigma(c??) \prec \sigma(c!!)$.

The correctness of a program $P$ will be specified in terms of formulas of the form $I : \{\phi\}P\{\psi\}$, where $I$, $\phi$ and $\psi$ are assertions (of the given first-order logic). The assertions $\phi$ and $\psi$ are called the precondition and postcondition, respectively. The assertion $I$ is called the (global) invariant, it expresses some invariant properties of the communication structure of a computation. As such a global invariant in general refers to sequence terms, no references to variables occurring in the program are allowed. Moreover, in $I$, $\phi$ and $\psi$ no quantification over the program variables of $P$ may occur. Intuitively the meaning of a correctness formula $I : \{\phi\}P\{\psi\}$ can be rendered as follows:

> Every state of a computation of $P$ starting in a state which satisfies both $I$ and $\phi$ satisfies $I$, and upon termination $\psi$ is guaranteed to hold.

Thus the formalism used here is a variant of *I-logic*, as introduced by Pandya [Pan88]. The following definition gives a more formal account of the semantics of correctness formulas:

**Definition 9** *Given a correctness formula* $I : \{\phi\}P\{\psi\}$ *such that* $I$ *and* $P$ *do not have program variables in common, we define* $\models I : \{\phi\}P\{\psi\}$ *iff for any* $\sigma$*, if* $\sigma \models I \wedge \phi$ *then for all finite* $CS \in [\![P]\!](\sigma)$*,* $\sigma' \models I$ *holds, where* $\sigma'$ *is the second component of the last configuration of* $CS$*; moreover if* $CS$ *is terminating, then* $\sigma' \models I \wedge \psi$ *holds.*

## 4 The proof system

In this section we present the proof system for deriving correctness formulas. In order to reason about the correctness of a program $P = [S_1 \parallel \ldots \parallel S_n]$ *compositionally*, that is, in terms of the correctness of its parallel components $S_i$, we introduce *local* correctness formulas of the form $I : \{p\}S\{q\}$, where $I$ is a global invariant which does not contain occurrences of program variables of $S$, and $p$ and $q$ are assertions which are allowed only to refer to variables occurring in $S$. The set of variables of a statement $S$ consists of its program variables and those input/output variables $c??$ ($c!!$) for which $c$ is an input channel of $S$ ($c$ is an output channel of $S$). We define the semantics of local correctness formulas axiomatically in terms of the following axioms and rules:

**Axiom 1** *(skip)* $I : \{p\}\mathsf{skip}\{p\}$

**Axiom 2** *(assignment)* $I : \{p[e/x]\}x := e\{p\}$

**Rule 1** *(output)*

$$\frac{(I \wedge p) \rightarrow (I \wedge q)[c!! \cdot e/c!!]}{I : \{p\}c!!e\{q\}}$$

**Rule 2** *(input)*

$$\frac{(I \wedge p \wedge c \neq \epsilon) \rightarrow (I \wedge q)[f(c)/x, c?? \cdot f(c)/c??]}{I : \{p\}c??x\{q\}}$$

**Rule 3** *(sequential composition)*

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

**Rule 4** *(conditional)*

$$\frac{I : \{p \wedge b\}S_1\{q\},\ I : \{p \wedge \neg b\}S_2\{q\}}{I : \{p\}\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\{q\}}$$

**Rule 5** *(while loop)*

$$\frac{I : \{p \wedge b\}S_1\{p\}}{I : \{p\}\mathsf{while}\ b\ \mathsf{do}\ S_1\ \mathsf{od}\{p \wedge \neg b\}}$$

**Rule 6** *(input conditional)*

$$\frac{I : \{p\}c??x; S_1\{q\},\ (I \wedge p \wedge c = \epsilon) \rightarrow (I \wedge r)[c?? \cdot \perp/c??],\ I : \{r\}S_2\{q\}}{I : \{p\}\mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}\{q\}}$$

**Rule 7** *(input while loop)*

$$\frac{I : \{p\}c??x; S\{p\},\ (I \wedge p \wedge c = \epsilon) \rightarrow (I \wedge q)[c?? \cdot \perp/c??]}{I : \{p\}\mathsf{while}\ c??x\ \mathsf{do}\ S_1\ \mathsf{od}\{q\}}$$

**Rule 8** *(repeat)*

$$\frac{I : \{p\}S\{r\},\ (I \wedge r \wedge c = \epsilon) \rightarrow (I \wedge p)[c?? \cdot \perp/c??],\ I : \{r\}c??x\{q\}}{I : \{p\}\mathsf{repeat}\ S\ \mathsf{until}\ c??x\{q\}}$$

**Rule 9** *(local consequence)*

$$\frac{I \wedge p \rightarrow p',\ I : \{p'\}S\{q'\},\ I \wedge q' \rightarrow q}{I : \{p\}S\{q\}}$$

Note that we have assertions occurring as premises, that is, we assume as additional axioms all valid first-order assertions. In the axiom for the assignment statement we only need to substitute the expression $e$ for $x$ in the postcondition $p$ since $I$ is not allowed to refer to program variables. Thus $I$ is on purely syntactical grounds invariant over assignments. The output statement $c!!e$ is modeled as an assignment to the corresponding output variable $c!!$ which consists of appending the value sent to the sequence $c!!$. Since the global invariant may refer to the variable $c!!$ we have to apply the corresponding substitution also to $I$ itself. In a similar manner an input statement $c??x$ is modeled as a (multiple) assignment to the variable $x$ and the input variable $c??$. To obtain a complete rule the enabledness condition of the input action, namely that the buffer $c$ is non-empty, is included. (Note that the term $c$ is actually an abbreviation of the term $c!! - r_{\perp}(c??)$.) The following rules for the conditional and while statement are straightforward extensions of the usual rules. The rules for the conditional, while and repeat statement with an input statement as a test combine naturally the standard rules and the rule for input statements. The main additional feature is the recording of a test on an empty buffer. Finally, the local consequence rule extends in an obvious way the classical rule.

Local correctness formulas can be combined into correctness formulas of an entire program as follows:

**Rule 10** *(parallel composition)*

$$\frac{I : \{p_i\}S_i\{q_i\}(i = 1, ..., n)}{I : \{\bigwedge_i p_i\}[S_1 \parallel ... \parallel S_n]\{\bigwedge_i q_i\}}$$

For completeness we need the following rules:

**Rule 11** *(consequence)*

$$\frac{I \wedge \phi \rightarrow I' \wedge \phi',\ I' : \{\phi'\}P\{\psi'\},\ I' \wedge \psi' \rightarrow \psi,\ I' \rightarrow I}{I : \{\phi\}P\{\psi\}}$$

**Rule 12** *(substitution)* Let $v$ be a variable not occurring in $P$ or $q$, and $t$ be a sequence term if $v$ is an input/output variable, and an arithmetical term, otherwise.

$$\frac{I : \{p\}P\{q\}}{I : \{p[t/v]\}P\{q\}}$$

# 5 Soundness

In this section we will prove the soundness of the proof system, that is, we will argue that every derivable correctness formula $I : \{\phi\}P\{\psi\}$ is valid. We will consider only the soundness of the parallel composition rule, the soundness of the consequence rule and the substitution rule being straightforward. Since the meaning of a local correctness formula is defined only axiomatically we introduce the notion of a local *proof outline*, and prove the soundness of the parallel composition rule by induction on the length of the computation of the program using information about the components as given by the local proof outlines.

In the following, we will need the syntactical continuations $after(R, S)$ and $before(R, S)$, where $R$ is a substatement of $S$. Informally, they denote the part of $S$ that remains to be executed after (before) $R$ has been executed.

**Definition 10** *Let $S$ be some statement, and $R$ some substatement of $S$. Define the syntactical continuations $after(R, S)$ and $before(R, S)$ as follows:*

- *If $R = S$ then $after(R, S) = E$*
- *If $S = \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$ or $S = \mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi}$ and $R$ is a substatement of $S_i$ ($i \in \{1, 2\}$) then $after(R, S) = after(R, S_i)$*
- *If $S = \mathsf{while}\ b\ \mathsf{do}\ S_1\ \mathsf{od}$ or $S = \mathsf{while}\ c??x\ \mathsf{do}\ S_1\ \mathsf{od}$ and $R$ is a substatement of $S_1$ then $after(R, S) = after(R, S_1); S$*
- *If $S = \mathsf{repeat}\ S_1\ \mathsf{until}\ c??x$ and $R$ is a substatement of $S_1$ then $after(R, S) = after(R, S_1);$ $\mathsf{if}\ c??x\ \mathsf{then\ skip\ else}\ S\ \mathsf{fi}$*
- *If $S = S_1; S_2$ and $R$ is a substatement of $S_1$ then $after(R, S) = after(R, S_1); S_2$; if $R$ is a substatement of $S_2$ then $after(R, S) = after(R, S_2)$*
- *$before(R, S) = R; after(R, S)$*

**Note:** We do not consider input commands occurring as tests in statement $S$ to be substatements of $S$.

**Definition 11** *Given a statement $S$, an invariant $I$ and pre- and postcondition $p$ and $q$, where $p$ and $q$ only contain variables occurring in $S$, a* proof outline *$pfo(I, p, S, q)$ consists of the program text of $S$ together with an assignment $pre(R)$ and $post(R)$ to all substatements $R$ of $S$ such that the following assertions are valid:*

1. *$(I \wedge p) \rightarrow pre(S)$ and $(I \wedge post(S)) \rightarrow q$*
2. *$(I \wedge pre(R)) \rightarrow post(R)$, for $R = \mathsf{skip}$*
3. *$(I \wedge pre(R)) \rightarrow post(R)[e/x]$, for $R = x := e$*
4. *$(I \wedge pre(R)) \rightarrow (I \wedge post(R))[c!! \cdot e/c!!]$, for $R = c!!e$*
5. *$(I \wedge pre(R) \wedge c \neq \epsilon) \rightarrow (I \wedge post(R))[f(c)/x, c?? \cdot f(c)/c??]$ for $R = c??x$*
6. *$(I \wedge pre(R)) \rightarrow pre(R_1)$, $(I \wedge post(R_1)) \rightarrow pre(R_2)$ and $(I \wedge post(R_2)) \rightarrow post(R)$, for $R = R_1; R_2$*
7. *$(I \wedge pre(R) \wedge b) \rightarrow pre(R_1)$,*
   *$(I \wedge pre(R) \wedge \neg b) \rightarrow pre(R_2)$ and $I \wedge post(R_i) \rightarrow post(R)(i = 1, 2)$,*
   *for $R = \mathsf{if}\ b\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2\ \mathsf{fi}$*
8. *$(I \wedge pre(R) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(R_1))[f(c)/x, c?? \cdot f(c)/c??]$,*
   *$(I \wedge pre(R) \wedge c = \epsilon) \rightarrow (I \wedge pre(R_2))[c?? \cdot \perp/c??]$, and*
   *$I \wedge post(R_i) \rightarrow post(R)(i = 1, 2)$, for $R = \mathsf{if}\ c??x\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2\ \mathsf{fi}$*
9. *$(I \wedge pre(R) \wedge b) \rightarrow pre(R_1)$, $(I \wedge post(R_1)) \rightarrow pre(R)$, and $(I \wedge pre(R) \wedge \neg b) \rightarrow post(R)$ for $R = \mathsf{while}\ b\ \mathsf{do}\ R_1\ \mathsf{od}$*

10. $(I \wedge pre(R) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(R_1))[f(c)/x, c?? \cdot f(c)/c??]$,
$(I \wedge post(R_1)) \rightarrow pre(R)$, *and* $(I \wedge pre(R) \wedge c = \epsilon) \rightarrow (I \wedge post(R))[c?? \cdot \perp/c??]$,
*for* $R =$ while $c??x$ do $R_1$ od

11. $(I \wedge pre(R)) \rightarrow pre(R_1)$,
$(I \wedge post(R_1) \wedge c = \epsilon) \rightarrow (I \wedge pre(R))[c?? \cdot \perp/c??]$, *and*
$(I \wedge post(R_1) \wedge c \neq \epsilon) \rightarrow (I \wedge post(R))[f(c)/x, c?? \cdot f(c)/c??]$,
*for* $R =$ repeat $R_1$ until $c??x$

The following lemma relates a proof of a local correctness formula with the existence of a local proof outline.

**Lemma 1** *A local correctness formula* $I : \{p\}S\{q\}$ *is derivable iff there exists a proof outline* $pfo(I, p, S, q)$.

**Proof** Only if: induction on the length of the derivation of $I : \{p\}S\{q\}$. The base cases (skip, assignment, input and output) are immediate. We consider some rules:

- sequential composition. Consider a proof ending with an application of the sequential composition rule, yielding $I : \{p\}S_1; S_2\{q\}$. Then we also have proofs of $I : \{p\}S_1\{r\}$ and $I : \{r\}S_2\{q\}$, for some $r$. By induction hypothesis there exist proofoutlines $pfo(I, p, S_1, r)$ and $pfo(I, r, S_2, q)$. In particular, we have $(I \wedge post(S_1)) \rightarrow r$ and $(I \wedge r) \rightarrow pre(S_2)$. From this we deduce $(I \wedge post(S_1)) \rightarrow pre(S_2)$. Choosing $pre(S_1; S_2) = pre(S_1)$ and $post(S_1; S_2) = post(S_2)$ then gives us a proofoutline $pfo(I, p, S_1; S_2, q)$.

- while rule. Consider a proof ending with an application of the while rule, yielding $I : \{p\}$while $b$ do $S_1$ od$\{p \wedge \neg b\}$. Then we also have a proof of $I : \{p \wedge b\}S_1\{p\}$. By induction hypothesis there exists a proofoutline $pfo(I, p \wedge b, S_1, p)$. Choosing $pre($while $b$ do $S_1$ od$) = p$ and $post($while $b$ do $S_1$ od$) = p \wedge \neg b$ then gives us a proofoutline $pfo(I, p,$ while $b$ do $S_1$ od$, q)$, because $(I \wedge p \wedge b) \rightarrow pre(S_1)$, $(I \wedge post(S_1)) \rightarrow p$ and $(I \wedge p \wedge \neg b) \rightarrow (p \wedge \neg b)$.

- input conditional. Consider a proof ending with an application of the input conditional rule, yielding $I : \{p\}$if $c??x$ then $S_1$ else $S_2$ fi$\{p \wedge \neg b\}$. Then there exists $r$ such that we have a proof of $I : \{p\}c??x; S_1\{q\}$ and of $I : \{r\}S_2\{q\}$, and $(I \wedge p \wedge c = \epsilon) \rightarrow (I \wedge r)[c?? \cdot \perp/c??]$ is valid. By induction hypothesis there exist proofoutlines $pfo(I, p, c??x; S_1, q)$ and $pfo(I, r, S_2, q)$. Thus, there are $pre(c??x; S_1)$, $post(c??x; S_1)$, $pre(c??x)$, $post(c??x)$, $pre(S_1)$, $post(S_1)$, $pre(S_2)$, and $post(S_2)$ such that $(I \wedge p) \rightarrow pre(c??x; S_1)$, $(I \wedge post(c??x; S_1)) \rightarrow q$, $(I \wedge pre(c??x; S_1)) \rightarrow pre(c??x)$, $(I \wedge post(c??x)) \rightarrow pre(S_1)$, $(I \wedge post(S_1)) \rightarrow post(c??x; S_1)$, $(I \wedge pre(c??x) \wedge c \neq \epsilon) \rightarrow (I \wedge post(c??x))[f(c)/x, c?? \cdot f(c)/c??]$, $(I \wedge r) \rightarrow pre(S_2)$ and $(I \wedge post(S_2)) \rightarrow q$.

  Choosing $pre($if $c??x$ then $S_1$ else $S_2$ fi$) = p$ and $post($if $c??x$ then $S_1$ else $S_2$ fi$) = q$ then gives us a proofoutline $pfo(I, p,$ while $b$ do $S_1$ od$, q)$, as can be seen by verifying $(I \wedge p \wedge c \neq \epsilon) \rightarrow (I \wedge pre(S_1))[f(c)/x, c?? \cdot f(c)/c??]$, $(I \wedge p \wedge c = \epsilon) \rightarrow (I \wedge pre(S_2))[c?? \cdot \perp/c??]$ and $(I \wedge post(S_i) \rightarrow q$ $(i = 1,2)$ from the above.

The if-part is proven with induction on the structure of $S$ (the input and output cases are omitted, being very similar to the assignment case):

- $S \equiv$ skip Suppose we have a proof outline $pfo(I, p, S, q)$. According to the definition of a proofoutline we then have assertions $pre(S)$ and $post(S)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, and $(I \wedge pre(S)) \rightarrow post(S)$. We next show a derivation of $I : \{p\}S\{q\}$:

(1) $I : \{p\}S\{p\}$, by Axiom 1
(2) $I : \{p\}S\{pre(S)\}$, by Rule 8, (1) and $(I \wedge p) \rightarrow pre(S)$
(3) $I : \{p\}S\{post(S)\}$, by Rule 8, (2) and $(I \wedge pre(S)) \rightarrow post(S)$
(4) $I : \{p\}S\{q\}$, by Rule 8, (3) and $(I \wedge post(S)) \rightarrow q$

- $S \equiv x := e$ Suppose we have a proof outline $pfo(I, p, S, q)$. We now have assertions $pre(S)$ and $post(S)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, and $(I \wedge pre(S)) \rightarrow post(S)[e/x]$. We next show a derivation of $I : \{p\}S\{q\}$:

  (1) $I : \{q[e/x]\}S\{q\}$, by Axiom 2
  (2) $I : \{post(S)[e/x]\}S\{q\}$, by Rule 8, (1) and $(I \wedge post(S)) \rightarrow q$, hence also $(I \wedge post(S)[e/x]) \rightarrow q[e/x]$
  (3) $I : \{pre(S)\}S\{q\}$, by Rule 8, (2) and $(I \wedge pre(S)) \rightarrow post(S)[e/x]$
  (4) $I : \{p\}S\{q\}$, by Rule 8, (3) and $(I \wedge p) \rightarrow pre(S)$

- $S \equiv S_1; S_2$ Suppose we have a proof outline $pfo(I, p, S, q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, $post(S_1)$, $pre(S_2)$, and $post(S_2)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S)) \rightarrow pre(S_1)$, $(I \wedge post(S_1)) \rightarrow pre(S_2)$, and $(I \wedge post(S_2)) \rightarrow post(S)$.

  From these facts we can easily derive $(I \wedge p) \rightarrow pre(S_1)$ and $(I \wedge post(S_1)) \rightarrow pre(S_2)$, and also $(I \wedge pre(S_2)) \rightarrow pre(S_2)$ and $(I \wedge post(S_2)) \rightarrow q$. This means we have proofoutlines $pfo(I, p, S_1, pre(S_2))$ and $pfo(I, pre(S_2), S_2, q)$. Using the induction hypothesis twice and the rule for sequential composition we arrive at $\vdash I : \{p\}S\{q\}$.

- $S \equiv \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi}$ Suppose we have a proof outline $pfo(I, p, S, q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, $post(S_1)$, $pre(S_2)$, and $post(S_2)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S) \wedge b) \rightarrow pre(S_1)$, $(I \wedge pre(S) \wedge \neg b) \rightarrow pre(S_2)$, $(I \wedge post(S_1)) \rightarrow post(S)$, and $(I \wedge post(S_2)) \rightarrow post(S)$.

  From these facts we derive $(I \wedge p \wedge b) \rightarrow pre(S_1)$ and $(I \wedge post(S_1)) \rightarrow q$, and also $(I \wedge p \wedge b) \rightarrow pre(S_2)$ and $(I \wedge post(S_2)) \rightarrow q$. This means we have proofoutlines $pfo(I, p \wedge b, S_1, q)$ and $pfo(I, p \wedge \neg b, S_2, q)$. Using the induction hypothesis twice and the rule for conditionals we arrive at $\vdash I : \{p\}S\{q\}$.

- $S \equiv \textsf{while } b \textsf{ do } S_1 \textsf{ od}$ Suppose we have a proof outline $pfo(I, p, S, q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, and $post(S_1)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S) \wedge b) \rightarrow pre(S_1)$, $(I \wedge post(S_1)) \rightarrow pre(S)$, and $(I \wedge pre(S) \wedge \neg b) \rightarrow post(S)$.

  From these facts we immediately derive the existence of a proofoutline $pfo(I, pre(S) \wedge b, S_1, pre(S))$. Using the induction hypothesis we obtain $\vdash I : \{pre(S) \wedge b\}S_1\{pre(S)\}$, and hence, by application of the while rule, $\vdash I : \{pre(S)\}S\{pre(S) \wedge \neg b\}$. Finally, we apply the consequence rule, using the facts $(I \wedge pre(S) \wedge \neg b) \rightarrow q$ and $(I \wedge p) \rightarrow pre(S)$, which follow from the above, to obtain $\vdash I : \{p\}S\{q\}$.

- $S \equiv \textsf{if } c??x \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi}$ Suppose we have a proof outline $pfo(I, p, S, q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, $post(S_1)$, $pre(S_2)$, and $post(S_2)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(S_1))[f(c)/x, c?? \cdot f(c)/c??]$, $(I \wedge pre(S) \wedge c = \epsilon) \rightarrow (I \wedge pre(S_2))[c?? \cdot \perp/c??]$, $(I \wedge post(S_1)) \rightarrow post(S)$, and $(I \wedge post(S_2)) \rightarrow post(S)$.

  From these facts we derive $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S)) \rightarrow pre(S)$, $(I \wedge pre(S_1)) \rightarrow pre(S_1)$, $(I \wedge post(S)) \rightarrow post(S)$, and $(I \wedge pre(S) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(S_1))[f(c)/x, c?? \cdot f(c)/c??]$. This means we have a proofoutline $pfo(I, p, c??x; S_1, q)$. Similarly, we derive $(I \wedge pre(S_2)) \rightarrow pre(S_2)$ and $(I \wedge post(S_2)) \rightarrow q$, giving us $pfo(I, pre(S_2), S_2, q)$ Using the induction hypothesis twice and $(I \wedge p \wedge c = \epsilon) \rightarrow (I \wedge pre(S_2))[c?? \cdot \perp/c??]$ the rule for input conditionals gives us $\vdash I : \{p\}S\{q\}$.

- $S \equiv \textsf{while } c??x \textsf{ do } S_1 \textsf{ od}$ Suppose we have a proof outline $pfo(I,p,S,q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, and $post(S_1)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(S_1))[f(c)/x, c?? \cdot f(c)/c??]$, $(I \wedge pre(S) \wedge c = \epsilon) \rightarrow (I \wedge post(S))[c?? \cdot \perp/c??]$, and $(I \wedge post(S_1)) \rightarrow pre(S)$.

  From the facts $(I \wedge pre(S)) \rightarrow pre(S)$, $(I \wedge pre(S_1)) \rightarrow pre(S_1)$ and $(I \wedge pre(S) \wedge c \neq \epsilon) \rightarrow (I \wedge pre(S_1))[f(c)/x, c?? \cdot f(c)/c??]$ we derive a proofoutline $pfo(I, pre(S), c??x; S_1, pre(S))$ (Somewhat more precisely: if we choose assertions $pre(c??x) = pre(c??x; S) = post(c??; S) = post'(S_1) = pre(S)$, and $post(c??x) = pre'(S_1) = pre(S_1)$, all requirements of the proofoutline are met. We use $pre'(S_1)$ and $post'(S_1)$ to distinguish from the assertions $pre(S_1)$ and $post(S_1)$ from the proofoutline $pfo(I,p,S,q)$.) By induction hypothesis, we have a proof of $I : \{pre(S)\}c??x; S_1\{pre(S)\}$. Now we can apply the input while loop rule, using $(I \wedge pre(S) \wedge c = \epsilon) \rightarrow (I \wedge post(S))[c?? \cdot \perp/c??]$, to obtain $\vdash I : \{pre(S)\}S\{post(S)\}$, which after application of the consequence rule (using $(I \wedge p) \rightarrow pre(S)$ and $(I \wedge post(S)) \rightarrow q$) gives $\vdash I : \{p\}S\{q\}$.

- $S \equiv \textsf{repeat } S_1 \textsf{ until } c??x$ Suppose we have a proof outline $pfo(I,p,S,q)$. We now have assertions $pre(S)$, $post(S)$, $pre(S_1)$, and $post(S_1)$ such that the following hold: $(I \wedge p) \rightarrow pre(S)$, $(I \wedge post(S)) \rightarrow q$, $(I \wedge pre(S)) \rightarrow pre(S_1)$, $(I \wedge post(S_1) \wedge c = \epsilon) \rightarrow (I \wedge pre(S))[c?? \cdot \perp/c??]$, and $(I \wedge post(S_1) \wedge c \neq \epsilon) \rightarrow (I \wedge post(S))[f(c)/x, c?? \cdot f(c)/c??]$.

  From $(I \wedge pre(S)) \rightarrow pre(S_1)$ we directly obtain a proofoutline $pfo(I, pre(S), S_1, post(S_1))$, and from $(I \wedge post(S)) \rightarrow q$ and $(I \wedge post(S_1) \wedge c \neq \epsilon) \rightarrow (I \wedge post(S))[f(c)/x, c?? \cdot f(c)/c??]$ we obtain a proofoutline $pfo(I, post(S_1), c??x, q)$ (take $pre(c??x) = post(S_1)$ and $post(c??x) = post(S)$). Thus, using the induction hypothesis we have proofs of $I : \{pre(S)\}S_1\{post(S_1)\}$ and $I : \{post(S_1)\}c??x\{q\}$. Now we apply the repeat rule using $(I \wedge post(S_1) \wedge c = \epsilon) \rightarrow (I \wedge pre(S))[c?? \cdot \perp/c??]$ to obtain a proof of $I : \{pre(S)\}S\{q\}$. Finally, applying the consequence rule using $(I \wedge p) \rightarrow pre(S)$ gives the desired result.

Given lemma 1 it is not difficult to derive as a corollary from theorem 1 below the soundness of the parallel composition rule.

**Theorem 1** *Given local proof outlines $pfo(I, p_i, S_i, q_i)$ $(1 \leq i \leq n)$, a state $\sigma$ such that $\sigma \models I \wedge \bigwedge_i p_i$, and a computation $\langle [S_1 \parallel ... \parallel S_n], \sigma\rangle \rightarrow^* \langle [R_1 \parallel ... \parallel R_n], \sigma'\rangle$ ($\rightarrow^*$ denotes the reflexive, transitive closure of the transition relation $\rightarrow$), we have*

1. *$\sigma' \models I$*
2. *if $R_i$ is $before(R, S_i)$ then $\sigma' \models pre(R)$*
3. *if $R_i$ is $after(R, S_i)$ then $\sigma' \models post(R)$*

**Proof** The proof proceeds with induction on the length of the computation.

The base case derives immediately from the validity of $(I \wedge p_i) \rightarrow pre(S_i)$ which follows from the local proof outlines $pfo(I, p_i, S_i, q_i)$.

With respect to the induction step, we treat some representative cases:

- $\langle [S_1 \parallel ... \parallel S_n], \sigma\rangle \rightarrow^* \langle [R_1 \parallel ... \parallel c??x; R_j \parallel ... \parallel R_n], \sigma''\rangle \rightarrow \langle [R_1 \parallel ... \parallel R_j \parallel ... \parallel R_n], \sigma'\rangle$.

  From the induction hypothesis we derive that $\sigma'' \models I$, $\sigma'' \models pre(R)$, if $R_i = before(R, S_i)$ or $\sigma'' \models post(R)$, if $R_i = after(R, S_i)$ $(i \neq j)$, and $\sigma'' \models pre(c??x)$. Now, since the local assertion $pre(R)$ $(post(R))$ refers only to the variables of $S_i$, it follows that $\sigma' \models pre(R)$ $(\sigma' \models post(R))$, for $i \neq j$. Furthermore, it is not difficult to derive from the proof outline $pfo(I, p_j, S_j, q_j)$ the validity of $(I \wedge pre(c??x) \wedge c \neq \epsilon) \rightarrow (I \wedge p)[f(c)/x, c?? \cdot f(c)/c??]$, where $p = pre(R)$ in case $R_j = before(R, S_j)$, for some substatement $R$ of $S_j$, and $p = post(R)$ in case $R_j = after(R, S_j)$, for some substatement $R$ of $S_j$.) We have that $\sigma'' \models I \wedge pre(c??x) \wedge c \neq \epsilon$, ($\sigma'' \models c \neq \epsilon$ follows from the existence of the last transition) and thus $\sigma'' \models (I \wedge p)[f(c)/x, c?? \cdot f(c)/c??]$. Since $\sigma' = \sigma''\{f(\sigma''(c))/x, \sigma''(c??) \cdot f(\sigma''(c))/c??\}$ we conclude that $\sigma' \models (I \wedge p)$.

- $\langle [S_1 \parallel ... \parallel S_n], \sigma\rangle \to^* \langle [R_1 \parallel ... \parallel \textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi}; R_j \parallel ... \parallel R_n], \sigma''\rangle \to \langle [R_1 \parallel ... \parallel S_1; R_j \parallel ... \parallel R_n], \sigma'\rangle$.

  From the induction hypothesis we again derive $\sigma' \models pre(R)(post(R))$ for $i \neq j$. Also by induction hypothesis we have $\sigma'' \models I \wedge pre(\textsf{if } b \textsf{ then } S_1 \textsf{ else } S_2 \textsf{ fi})$. Furthermore, by the last step we know $\sigma'' \models b$, so by $j$'s proof outline we derive $\sigma'' \models pre(S_1)$, and hence, because $\sigma'' = \sigma'$ also $\sigma' \models pre(S_1)$ (the other choice possibility is completely analogous).

- $\langle [S_1 \parallel ... \parallel S_n], \sigma\rangle \to^* \langle [R_1 \parallel ... \parallel \textsf{repeat } S_1 \textsf{ until } c??x; R_j \parallel ... \parallel R_n], \sigma''\rangle \to \langle [R_1 \parallel ... \parallel S_1; \textsf{if } c??x \textsf{ then skip else repeat } S_1 \textsf{ until } c??x \textsf{ fi}; R_j \parallel ... \parallel R_n], \sigma'\rangle$.

  From the induction hypothesis we again derive $\sigma' \models pre(R)(post(R))$ for $i \neq j$. Also by induction hypothesis we have $\sigma'' \models I \wedge pre(\textsf{repeat } S_1 \textsf{ until } c??x)$. By $j'$s proof outline we derive $\sigma'' \models pre(S_1)$, and hence, because $\sigma'' = \sigma'$ also $\sigma' \models pre(S_1)$.

□

**Corollary 1** *The parallel composition rule is sound.*

**Proof** We show that any derivation ending in an application of the parallel composition rule leads to a valid formula. Suppose we have a derived $I : \{\bigwedge_i p_i\}[S_1 \parallel ... \parallel S_n]\{\bigwedge_i q_i\}$, using the parallel composition rule in the last step. Then, for all $i$, we have a derivation of $I : \{p_i\}S_i\{q_i\}$. By lemma 1 we have proofoutlines $pfo(I, p_i, S_i, q_i)$ for all $i$.

Now consider a state $\sigma$ such that $\sigma \models I \wedge \bigwedge_i p_i$. Let $CS$ be a finite terminating computation sequence of $[S_1 \parallel ... \parallel S_n]$, of which $\sigma'$ is the second component of the last configuration. According to theorem 1, we have that $\sigma' \models I \wedge \bigwedge_i post(S_i)$, and hence, using $(I \wedge post(S_i)) \to q_i$ from the proofoutline of $S_i$ also $\sigma' \models I \wedge \bigwedge_i q_i$. We therefore conclude $\models I : \{\bigwedge_i p_i\}[S_1 \parallel ... \parallel S_n]\{\bigwedge_i q_i\}$ □

# 6 Completeness

We prove completeness in the sense that every valid global correctness formula is derivable. Let $I : \{\phi\}P\{\psi\}$ be a valid correctness formula, with $P = [S_1 \parallel ... \parallel S_n]$. We will sketch a proof of the derivability of $I : \{\phi\}P\{\psi\}$. Let $\bar{v}$ be the variables of $P$ (both the program variables and those input/output variables $c??, c!!$, for which $c$ is a channel of $P$). By $\bar{v}_i$ we denote the variables of $S_i$. To be more precise, $\bar{v}_i$ consists of all the program variables of $S_i$ and the input (output) variables $c??$ ($c!!$) with $c$ an input (output) channel of $S_i$. Finally, the set of input/output variables of $P$ we denote by $\bar{c}$ (so $\bar{c}$ consists of the variables $c??$, $c!!$, with $c$ a channel of $P$).

We first construct local proof outlines of the components $S_i$ by introducing local assertions $pre(R)$ and $post(R)$, for all substatements $R$ of $S_i$. Roughly speaking, an assertion $pre(R)$ characterizes all those intermediate states of a computation of the program $P$ such that process $i$ is about to execute $R$ (or has just finished executing $R$, in case of $post(R)$). Let $\phi' = \phi \wedge \bar{v} = \bar{z}$, where $\bar{z}$ are new variables which are introduced to 'freeze' the initial values of $\bar{v}$.

**Definition 12** *For $R$ a substatement of $S_i$ we define:*

$$\begin{array}{lll} \sigma \models pre(R) \text{ iff } & \exists \sigma', \sigma'', R_1, ..., R_n \textit{ such that } & \sigma' \models I \wedge \phi' \\ & & \langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R_1 \parallel ... \parallel R_n], \sigma'' \rangle \\ & & \sigma(\bar{v}_i) = \sigma''(\bar{v}_i) \textit{ and } \sigma(\bar{z}) = \sigma''(\bar{z}) \\ & & R_i = \textit{before}(R, S_i) \end{array}$$

*and*

$$\begin{array}{lll} \sigma \models post(R) \text{ iff } & \exists \sigma', \sigma'', R_1, ..., R_n \textit{ such that } & \sigma' \models I \wedge \phi' \\ & & \langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R_1 \parallel ... \parallel R_n], \sigma'' \rangle \\ & & \sigma(\bar{v}_i) = \sigma''(\bar{v}_i) \textit{ and } \sigma(\bar{z}) = \sigma''(\bar{z}) \\ & & R_i = \textit{after}(R, S_i) \end{array}$$

Next we define the following global invariant $I'$:

**Definition 13** *Let $I'$ be an assertion such that:*

$$\begin{array}{lll} \sigma \models I' \text{ iff } & \exists \sigma', \sigma'', R_1, ..., R_n \textit{ such that } & \sigma' \models I \wedge \phi' \\ & & \langle [S_1, ..., S_n], \sigma' \rangle \rightarrow^* \langle [R_1, ..., R_n], \sigma'' \rangle \\ & & \sigma(\bar{c}) = \sigma''(\bar{c}) \textit{ and } \sigma(\bar{z}) = \sigma''(\bar{z}) \end{array}$$

In words, $I'$ asserts that $\sigma$'s valuation of the input/output variables of $P$ can actually be obtained by some computation of $[S_1 \parallel ... \parallel S_n]$ starting from a state satisfying $I \wedge \phi'$.

By standard techniques [TZ88], $I'$, $pre(R)$ and $post(R)$ can be shown to be expressible in our assertion language such that the variables of $pre(R)$ and $post(R)$ ($R$ a substatement of $S_i$) are among the variables $\bar{v}_i$ and $\bar{z}$, and the variables of $I'$ are among the variables $\bar{c}$ and the variables $\bar{z}$.

In order to prove that the assertions $pre(R)$, $post(R)$ and $I'$ as above, with $R$ a substatement of $S_i$, define a proof outline of $S_i$ we need the following lemma:

**Lemma 2 (merging lemma)** *Let, for $i = 1, \ldots, n$, be given the computations*

$$\langle [S_1 \parallel ... \parallel S_n], \sigma \rangle \rightarrow^* \langle [... \parallel R_i \parallel ...], \sigma_i \rangle$$

*Then for any computation*

$$\langle [S_1 \parallel ... \parallel S_n], \sigma \rangle \rightarrow^* \langle [R'_1 \parallel ... \parallel R'_n], \sigma' \rangle$$

*such that $\sigma'$ and $\sigma_i$ agree with respect to the input/output variables of $S_i$, there exists a computation*

$$\langle [S_1 \parallel ... \parallel S_n], \sigma \rangle \rightarrow^* \langle [R_1 \parallel ... \parallel R_n], \sigma'' \rangle$$

*with $\sigma''(\bar{v}_i) = \sigma_i(\bar{v}_i)$ $(i = 1, \ldots, n)$.*

The above lemma follows in a straightforward manner from the fact that the input/output behaviour of a statement $S_i$ as given by its input/output variables completely determines its local behaviour up to internal, i.e. non-communication actions ($S_i$ being deterministic). A formal proof is given below. Meanwhile, it is worthwhile to point out that the above lemma holds because of the recording of tests on an empty buffer. Consider the following two statements:

$$S = \textsf{if } c??x \textsf{ then } d!!0; R_1 \textsf{ else } d!!0; c??x; R_2 \textsf{ fi}$$

and

$$S' = \textsf{if } d??y \textsf{ then } c!!0; R'_1 \textsf{ else } c!!0; d??y; R'_2 \textsf{ fi}$$

Given an initial state with empty buffers $c$ and $d$, there exists a computation of $[S \parallel S']$ which reaches $R_1$ and there exists a computation of $[S \parallel S']$ which reaches $R'_1$. But there does not exist a computation which reaches both $R_1$ and $R'_1$. This does not invalidate the merging lemma, because there is no computation ending in a state which agrees with the two computations with respect to the input/output variables; for the first computation reaches a state $\sigma$ with $\sigma(c??) = \sigma(c!!) = \sigma(d!!) = \langle 0 \rangle$, $\sigma(d??) = \langle \perp, 0 \rangle$ and the second computation reaches a state $\sigma'$ with $\sigma'(c??) = \langle \perp, 0 \rangle$, $\sigma'(d??) = \sigma'(d!!) = \sigma'(c!!) = \langle 0 \rangle$. If we would not record tests on empty buffers there *is* a computation ending in a state which agrees with the two computations (take one of them), which would invalidate the merging lemma.

**Proof of merging lemma** Let $|\sigma(c??)|$ ($|\sigma(c!!)|$) denote the length of the sequence that is assigned by $\sigma$ to channel variable $c??$ ($c!!$). Note that we also take into account the special symbol $\perp$. We prove the merging lemma with induction on $\Sigma_{c \in IO}(|\sigma'(c??)| + |\sigma'(c!!)|)$, that is to say the total number of all input and output actions that have taken place in the computation $CS \stackrel{\text{def}}{=} \langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [R'_1 \parallel \ldots \parallel R'_n], \sigma' \rangle$ from the lemma.

- $\Sigma_{c \in IO}(|\sigma'(c??)| + |\sigma'(c!!)|) = 0$: then for all $i$, $\sigma'(c??) = \sigma'(c!!) = \epsilon$. Hence, $\sigma_i(c??) = \sigma_i(d!!) = \epsilon$, where $c??, d!! \in IO_i$. So, for every $i$, $CS_i \stackrel{\text{def}}{=} \langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\ldots \parallel R_i \parallel \ldots], \sigma_i \rangle$ contains 0 $i - IO$ transitions. Therefore, we can leave out all non-$i$ transitions from $CS_i$ (updating the intermediate global states in order to obtain a correct global computation) for all $i$, obtaining $CS'_i$, and then "glue" all $CS'_i$ together, executing them one after another, again revising the global states accordingly. Evidently, this gives the desired computation.

- $\Sigma_{c \in IO}(|\sigma'(c??)| + |\sigma'(c!!)|) > 0$: Suppose that the last $IO$-transition in $CS$ was due to execution of an input $c??x$ in process $i$ (replacing $c??x$ by $d!!e$ is completely analogous). Thus, $CS$ has the following form:

  $\langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\hat{R'}_1 \parallel \ldots \parallel c??x; \hat{R'}_i \parallel \ldots \parallel \hat{R'_n}], \hat{\sigma} \rangle \to \langle [\hat{R'}_1 \parallel \ldots \parallel \hat{R'_n}], \hat{\sigma'} \rangle \to^* \langle [R'_1 \parallel \ldots \parallel R'_n], \sigma' \rangle$. Define $CS'$ as the prefix of $CS$ up to and including the configuration $\langle [\hat{R'}_1 \parallel \ldots \parallel c??x; \hat{R'}_i \parallel \ldots \parallel \hat{R'_n}], \hat{\sigma} \rangle$, i.e. up to the point where $c??x$ is about to be executed. By lemma 3 we know that there exists $CS'_i = \langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\ldots \parallel c??x; \hat{R}'_i \parallel \ldots], \sigma''_i \rangle \to \langle [\ldots \parallel \hat{R}'_i \parallel \ldots], \sigma'''_i \rangle \to^* \langle [\ldots \parallel R_i \parallel \ldots], \sigma'_i \rangle$ where $\sigma'_i(\bar{v}_j) = \sigma_i(\bar{v}_j)$, and all transitions after the input $c??x$ are internal $i$-transitions. Define $CS''_i$ as the prefix of $CS'_i$ up to and including the configuration $\langle [\ldots \parallel c??x; \hat{R}'_i \parallel \ldots], \sigma''_i \rangle$.

  Then, we can use the induction hypothesis, using the existence of $CS_j$, for $j \neq i$, $CS''_i$ and $CS'$, yielding a computation sequence $\overline{CS}$: $\langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [R_1 \parallel \ldots \parallel \hat{R}'_i \parallel \ldots \parallel R_n], \overline{\sigma} \rangle$ with $\overline{\sigma}(\bar{v_j}) = \sigma_j(\bar{v_j})$, $j \neq i$ and $\overline{\sigma}(\bar{v}_i) = \sigma''_i(\bar{v}_i)$.

  Now we can extend $CS'$ with the transition sequence $CS_i$ - $CS''_i$, again adjusting the global states accordingly, to get the desired run.

□

**Lemma 3** *Suppose there exists a computation sequence* $\langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\ldots \parallel R_i \parallel \ldots], \sigma_i \rangle$. *Suppose furthermore there exists a c.s.* $\langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\ldots \parallel c??x; R'_i \parallel \ldots], \sigma'' \rangle \to \langle [\ldots \parallel R'_i \parallel \ldots], \sigma''' \rangle \to^* \langle [R'_1 \parallel \ldots \parallel R'_i \parallel \ldots \parallel R'_n], \sigma' \rangle$ *such that* $\sigma'$ *agrees with* $\sigma_i$ *on the input/output variables of process* $i$, *and furthermore* $c??x$ *is the last input/output command that is executed. Then, there exists a c.s.* $\langle [S_1 \parallel \ldots \parallel S_n], \sigma \rangle \to^* \langle [\ldots \parallel c??x; R'_i \parallel \ldots], \sigma''_i \rangle \to \langle [\ldots \parallel R'_i \parallel \ldots], \sigma'''_i \rangle \to^* \langle [\ldots \parallel R_i \parallel \ldots], \sigma'_i \rangle$ *where* $\sigma'_i(\bar{v}_i) = \sigma_i(\bar{v}_i)$, $c??x$ *is the last input/output command that is executed and after* $c??x$ *only* $i$*-transitions occur. (Again, this lemma also holds in the case that* $c??x$ *is replaced by* $d!!e$*.)*

**Proof** By the fact that $\sigma_i$ agrees with $\sigma'$ with respect to the input/output variables of process $i$, we know that the last $i$-IO-transition in the first transition sequence is due to execution of $c??x$ in $S_i$. This is because the ordering of input and output actions is uniquely determined by the value of the input/output variables of a (deterministic!) process. In order to obtain from this computation sequence the desired computation sequence, all we have to do is dismiss all subsequent non-$i$ transitions from it, adjusting the states accordingly. Note that this is possible because none of the subsequent $i$-transitions depends on any non-$i$ transition that is being removed $\Box$

Now we are ready for the following key lemma of the completeness proof:

**Lemma 4** *The assignment of local assertions $pre(R)$ and $post(R)$ to any substatement $R$ of $S_i$ defines a proof outline $pfo(I', pre(S_i), S_i, post(S_i))$.*

**Proof** We only illustrate the following case from definition 11. Consider condition 5: $(I' \wedge pre(R) \wedge c \neq \epsilon) \rightarrow (I' \wedge post(c??x))[f(c)/x, c?? \cdot f(c)/c??]$, with $c??x$ occurring in $S_i$. Suppose for some $\sigma$ we have $\sigma \models I' \wedge pre(R) \wedge c \neq \epsilon$. By definition of $pre(c??x)$ and $I'$ there exist computations

$$\langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R_1 \parallel ... \parallel c??x; R \parallel ...R_n], \sigma_i \rangle$$

and

$$\langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R'_1 \parallel ... \parallel R'_n], \sigma'' \rangle$$

such that $\sigma' \models I \wedge \phi'$, $\sigma(\bar{v}_i) = \sigma_i(\bar{v}_i)$, $\sigma(\bar{z}) = \sigma_i(\bar{z})$, $\sigma(\bar{c}) = \sigma''(\bar{c})$, and $\sigma(\bar{z}) = \sigma''(\bar{z})$. Note that the introduction of the freeze variables $\bar{z}$ and the additional conjunct $\bar{v} = \bar{z}$ of $\phi'$ allows us to assume without loss of generality that the above computations start indeed with the same initial state.

Now, since the states $\sigma_i$ and $\sigma''$ agree with respect to the input/output variables of $S_i$, we have by lemma 2 that there exists a computation

$$\langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R'_1 \parallel ... \parallel c??x; R \parallel ...R'_n], \sigma''' \rangle$$

such that $\sigma_i(\bar{v}_i) = \sigma'''(\bar{v}_i)$. Furthermore, since $\sigma$ and $\sigma'''$ agree with respect to the input/output variables (this is because $\sigma(\bar{v}_i) = \sigma_i(\bar{v}_i) = \sigma'''(\bar{v}_i)$ and $\sigma'''(\bar{v}_j) = \sigma''(\bar{v}_j)$ for $i \neq j$ and $\sigma''$ agrees with $\sigma$ with respect to $\bar{c}$) and $\sigma(c) \neq \epsilon$, it also follows that $\sigma'''(c) \neq \epsilon$, so that the input action $c??x$ indeed is enabled in $\sigma'''$. Thus we have by definition of $I'$ and $post(c??x)$ that $\sigma'''\{f(c)/x, \sigma'''(c??) \cdot f(c)/c??\} \models I' \wedge post(c??x)$, that is, $\sigma''' \models (I' \wedge post(c??x))[f(c)/x, c?? \cdot f(c)/c??]$. From this we conclude that $\sigma \models (I' \wedge post(c??x))[f(c)/x, c?? \cdot f(c)/c??]$ (note that $\sigma$ and $\sigma'''$ agree with respect to the variables $\bar{c}$ and $\bar{v}_i$). $\Box$

By the above lemma and lemma 1, we thus infer $\vdash I' : \{pre(S_i)\}S_i\{post(S_i)\}$, for all $i$. So, using the parallel composition rule, we derive $\vdash I' : \{\bigwedge_i pre(S_i)\}P\{\bigwedge_i post(S_i)\}$.

To proceed we need the following propositions:

**Proposition 1** *We have $\models (I \wedge \phi') \rightarrow (I' \wedge \bigwedge_i pre(S_i))$.*

The above proposition follows immediately from the definition of $I'$ and $pre(S_i)$.

**Proposition 2** *We have $\models (I' \wedge \bigwedge_i post(S_i)) \rightarrow \psi$.*

**Proof** Suppose $\sigma \models (I' \wedge \bigwedge_i post(S_i))$, then by definition of $I'$ and $post(S_i)$ we have a computation $\langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [R'_1 \parallel ... \parallel R'_n], \sigma'' \rangle$ and, for all $i$, a computation $\langle [S_1 \parallel ... \parallel S_n], \sigma' \rangle \rightarrow^* \langle [... \parallel E \parallel ...], \sigma_i \rangle$ such that $\sigma' \models I \wedge \phi'$, $\sigma(\bar{v}_i) = \sigma_i(\bar{v}_i)$ and $\sigma(\bar{c}) = \sigma''(\bar{c})$ (again we may assume a common initial state). Hence, by lemma 2 we have a computation $\langle[S_1 \parallel ... \parallel S_n],\sigma'\rangle \rightarrow^* \langle[E \parallel ... \parallel E],\sigma'''\rangle$ such that $\sigma'''(\bar{v}_i) = \sigma_i(\bar{v}_i)$. So we have a terminating computation sequence of $[S_1 \parallel ... \parallel S_n]$, starting in a state $\sigma'$ that satisfies $\sigma' \models I \wedge \phi$ (even $\sigma' \models I \wedge \phi'$), and ending in $\sigma'''$, hence by $\models I : \{\phi\}P\{\psi\}$ we derive $\sigma''' \models \psi$ and hence $\sigma \models \psi$ □

**Proposition 3** *We have* $\models I' \rightarrow I$.

This proposition follows immediately from the definition of $I'$ and the validity of $I : \{\phi\}P\{\psi\}$.

We conclude with the following corollary:

**Corollary 2** *We have* $\vdash I : \{\phi\}P\{\psi\}$.

**Proof** By an application of the consequence rule, using the above propositions we obtain the derivability of $I : \{\phi'\}P\{\psi\}$. An application of the substitution rule (substituting $\bar{v}$ for $\bar{z}$) then gives $\vdash I : \{\phi\}P\{\psi\}$ □

# 7 Extending synchronization

In the programming language discussed so far synchronization is modeled by the input statement $c??x$ (note that input commands as tests in choice and while constructs model only communication and do not incorporate synchronization). However since an input command $c??x$ only checks for the channel $c$ (and suspends in case its corresponding buffer is empty) its proper execution implicitly requires a predictable and determinate environment which is guaranteed to send eventually along channel $c$.

In order to increase the capability of a deterministic process to respond to an indeterminate environment we introduce a natural generalization of the input command: the input command $C??x$, with $C$ a non-empty finite *set* of channels, which allows a process to scan the channels of $C$ simultaneously. More precisely, the execution of an input command $C??x$ consists of selecting non-deterministically a non-empty channel $c \in C$, and reading a value from $c$ (which is then stored in $x$). The execution of $C??x$ suspends when all the channels of $C$ are empty. Formally, we have the following new syntax of a sequential process:

$$
\begin{array}{rcl}
S & ::= & \mathsf{skip} \\
& | & x := e \\
& | & C??x \mid c!!e \\
& | & S_1; S_2 \\
& | & \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \\
& | & \mathsf{while}\ b\ \mathsf{do}\ S\ \mathsf{od} \\
& | & \mathsf{if}\ c??x\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{fi} \\
& | & \mathsf{while}\ c??x\ \mathsf{do}\ S\ \mathsf{od} \\
& | & \mathsf{repeat}\ S\ \mathsf{until}\ c??x
\end{array}
$$

where $C$ denotes a (non-empty) finite set of channels. The (simple) input command $c??x$ will be interpreted as an abbreviation of $\{c\}??x$. For technical convenience we only allow simple input commands to occur as tests in the choice and while construct, although the proof theory to be presented below can easily be extended to the general case.

It is not so difficult to see that recording for each channel the values sent and received, respectively, is not sufficient anymore to determine the local behaviour of a process completely. Consider for example the process

$$\{c,d\}??x; \mathsf{if}\ x=0\ \mathsf{then}\ \{c,d\}??x; R_1\ \mathsf{else}\ \{c,d\}??x; x := x+1; R_2\ \mathsf{fi}$$

Suppose that the value 0 has been sent first along channel $c$ and that the value 1 has been sent first along $d$. Recording only for the channels $c$ and $d$ the values sent and received, respectively, we would not be able to determine whether the above process is about to execute $R_1$ or $R_2$: namely, either the process could have read first from $c$ and then from $d$ or vica versa.

In order to be able to determine the local behaviour of a process we introduce for each process $S_i$ (of a program $P = [S_1 \parallel \ldots \parallel S_n]$) a local *channel* variable $h_i$ which records the sequence of channels that have been selected in a generalized input command. Axiomatically, this is formalized as follows. Let $P = [S_1 \parallel \ldots \parallel S_n]$. A local correctness formula is of the form $I : \{p\}R\{q\}$, with $R$ a substatement of $S_i$ $(i = 1, \ldots, n)$, and $I$ is an assertion which is not allowed to refer to the program variables of $P$ (so it is allowed to refer to the channel variables), $p$ and $q$ are assertions which are allowed to refer only to the set of variables of $S_i$, which now additionally contains, besides its program variables and its input/output variables, the channel variable $h_i$ (the input variables of a statement $S$ consist of all those variables $c?$ for which there exists a substatement $C??x$ of $S$ such that $c \in C$). We have the following proof rule for an input command $C??x$:

$$\frac{(I \wedge p \wedge c \neq \epsilon) \rightarrow (I \wedge q)[f(c)/x, c? \cdot f(c)/c?, h_i \cdot c/h_i]}{I : \{p\}C??x\{q\}}$$

Here $c \in C$, and $C??x$ is understood to occur in $S_i$ of the program $P = [S_1 \parallel \ldots \parallel S_n]$. Since we allow a channel variable $h_i$ to occur in the global invariant $I$ we have to include $I$ in the scope of the newly added substitution $[h_i \cdot c/h_i]$, which models appending the channel $c$ to the sequence of channels $h_i$. We assume in the above rule that $C$ is not a singleton set. For $\{c\}??x$, that is $c??x$, we have the same proof rule as before:

$$\frac{(I \wedge p \wedge c \neq \epsilon) \rightarrow (I \wedge q)[f(c)/x, c? \cdot f(c)/c?]}{I : \{p\}c??x\{q\}}$$

For the formal justification of the resulting proof system we have to include in a state a valuation of the new channel variables $h_i$. The semantics of an input command $C??x$ is then described in terms of the input/output variables and the corresponding channel variable. For $C??x$ in $S_i$ of the program $P = [S_1 \parallel \ldots \parallel S_n]$ we have the following semantic description corresponding to the above proof rule:

$$\langle [\ldots \parallel C??x; S \parallel \ldots], \sigma\rangle \rightarrow \langle [\ldots \parallel S \parallel \ldots], \sigma\{\sigma(c?) \cdot d/c?, d/x, \sigma(h_i) \cdot c/h_i\}\rangle$$

provided $\sigma(c) \neq \epsilon$ and $d = f(\sigma(c))$, for $c \in C$. Again, here $C$ is assumed to be different from a singleton set. Input commands of the form $\{c\}??x$, that is $c??x$, are treated as before: they do not require an additional update to the channel variable $h_i$.

It should be noted that the additional information recorded by the channel variables still provides an abstraction from histories as sequences of communication events, where a communication event is of the form $c??d$ or $c!!d$, indicating that the value $d$ has been read from channel $c$, or that $d$ has been sent along $c$.

Soundness and completeness can be proved in essentially the same way as before. In the completeness proof both the local assertions $pre(R)$, $post(R)$ and the global invariant $I$ additionally specify the valuation of the new channel variables. Note that the set of local variables of $S_i$ (belonging to the program $P = [S_1 \parallel \ldots \parallel S_n]$) now include the channel variable $h_i$. The global invariant now specifies the values of the variables $\bar{c}$ which, besides the usual input/output variables, include the channel variables $h_i$. The main point of the completeness proof consists of the observation that also in this new case the merging lemma holds, which follows easily from the observation that the input/output variables of a process together with its channel variable completely determine its local behaviour.

## 8 Conclusion

We have shown that it is possible to obtain a compositional proof theory for distributed systems composed of asynchronously communicating processes, using input/output variables only to reason about communication, provided the programming language considered is in essence deterministic. In spite of this determinism, by providing constructs which allow a process to test the contents of a buffer we obtain a quite powerful language in which one can describe processes with the ability to respond to an indeterminate environment.

Nevertheless, it seems possible to extend the degree of nondeterminism available, as is indicated in section 7. Therefore, it is interesting to investigate the expressivity of the language thus obtained, in particular as compared to common nondeterministic languages, such as CSP.

First of all, the nondeterminism introduced by the generalized input command can be seen to be of a limited nature. This is because a process is not able to choose between subsequent actions (following the generalized input command) depending on which of the channels was read. In other words, the generalized input command could be viewed as a CSP guarded command with boolean guard parts set to **true** and empty bodies.

In the following we will indicate how, via coding messages by tagging them with the channel name of the channel over which they are sent, it is possible to express the CSP guarded command $[c_1??x_1 \rightarrow S_1[]...[]c_n??x_n \rightarrow S_n]$ in our —extended— language, provided that all channels are distinct.

Suppose that, instead of the normal values $v$, now messages of type $v_c$ are transmitted, where $c$ refers to the channel over which $v$ is sent. Also assume the operations *tag* and *val* which yield the channel and value of a message. Then we can write the above statement as follows:

$$
\begin{array}{l}
\{c_1, ..., c_n\}??x; \mathsf{if}\ tag(x) = c_1\ \mathsf{then}\ x_1 := val(x); S_1\ \mathsf{else} \\
\qquad\qquad\qquad \vdots \\
\qquad\qquad\quad\ \ \mathsf{if}\ tag(x) = c_n\ \mathsf{then}\ x_n := val(x); S_n\ \mathsf{else\ skip\ fi}
\end{array}
$$

Of course, the introduction of this new type of message along with the operations *tag* and *val* has its price: we are no longer able to reason on the same abstractness level as that of the (original, i.e. CSP) program, because we now have to hack around unravelling messages. This provides an interesting trade-off between aiming for compositionality using as little history information as possible on one hand, and staying as close to the abstractness level of the programming language on the other hand.

## References


[AFdR80] K.R. Apt, N. Francez, and W.P. de Roever. A proof system for communicating sequential processes. *ACM-TOPLAS*, 2(3):359–385, 1980.

[Fra92] N. Francez. *Program Verification.* Addison Wesley, 1992.

[HdR86] J. Hooman and W.P. de Roever. The quest goes on: a survey of proof systems for partial correctness of CSP. In *Current trends in concurrency*, volume 24 of *Lecture Notes in Computer Science*, pages 343–395. Springer-Verlag, 1986.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[Pan88] P.K. Pandya. *Compositional Verification of Distributed Programs.* PhD thesis, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400 005, INDIA, 1988.

[TZ88] J.V. Tucker and J.I. Zucker. *Program Correctness over Abstract Data Types, with Error-State Semantics.* CWI Monographs 6. North-Holland, 1988.

[ZdRvEB85] J. Zwiers, W.P. de Roever, and P. van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *Proc. ICALP'85*, volume 194 of *Lecture Notes in Computer Science.* Springer-Verlag, 1985.

[Zwi88] J. Zwiers. *Compositionality, Concurrency and Partial Correctness.* PhD thesis, Technical University Eindhoven, 1988.