Research report 96

SPECIFICATION AND VERIFICATION OF TOTAL CORRECTNESS OF DISTRIBUTED PROGRAMS

Mathai Joseph
Paritosh Pandya*

(RR96)

Abstract

This paper describes a compositional specification and proof system for networks of distributed processes. Each process in a network is specified using first order logic in terms of a *presupposition* P and an *affirmation* A as a triple {P} S {A}. For purely sequential programs, these triples reduce to the familiar Hoare triples extended for total correctness. In distributed programs, P-A triples allow the internal behaviour of a process to be specified in the context of the communications of the other processes in the network. Communications may either by synchronous or asynchronous. Properties such as termination, the absence of deadlock and the absence of livelock can be verified. As the technique is syntax-directed and allows network abstraction, proofs follow the structure of the program and a subnetwork within a network can be replaced by a single process. It also allows proof of the properties of non-terminating processes, such as servers.

Department of Computer Science
University of Warwick
Coventry
CV4 7AL, UK

* and
Tata Institute of Fundamental Research
Bombay
India

Feb 1987

Specification and Verification of Total Correctness of Distributed Programs

*Paritosh K. Pandya**

Tata Institute of Fundamental Research, Bombay

Mathai Joseph

University of Warwick, Coventry

ABSTRACT

This paper describes a compositional specification and proof system for networks of distributed processes. Each process in a network is specified using first order logic in terms of a *presupposition* P and an *affirmation* A as a triple {P} S {A}. For purely sequential programs, these triples reduce to the familiar Hoare triples extended for total correctness. In distributed programs, P-A triples allow the internal behaviour of a process to be specified in the context of the communications of the other processes in the network. Communications may either be synchronous or asynchronous. Properties such as termination, the absence of deadlock and the absence of livelock can be verified. As the technique is syntax-directed and allows network abstraction, proofs follow the structure of the program and a subnetwork within a network can be replaced by a single process. It also allows proof of the properties of non-terminating processes, such as servers.

1. Introduction

There has been much interest over the last 10 years in the class of distributed programming languages designed to have close relation with formal methods for the design and verification of programs. Among the best known of such languages are CSP [8] and its several variants such as occam13 which gave rise to a number of partial correctness proof systems for establishing the so-called safety properties of programs (eg, Apt et al [1], Levin and Gries [12], Zhou and Hoare[5], Misra and Chandy [14], Soundararajan [22] etc.). While alternative techniques were then explored for proving total correctness and liveness properties (Hoare[9], Barringer et al[3]) new forms of partial correctness proof systems were sought to allow programs and proofs to be developed compositionally (see de Roever[21], and Hooman and de Roever[10]).

Apart from needing close association between program development and proof, practical work in distributed systems also demands the use of richer communication primitives than the purely synchronous communication permitted in CSP and proof systems are now required to consider for example the use of asynchronous and broadcast communication between processes, and to provide means of specifying non-terminating (server) processes. Ideally, this must be done in a compositional proof system dealing with total correctness and liveness properties and handling the attendant requirements of establishing deadlock freedom, termination and divergence freedom.

The wealth of past work provides a range of techniques for proving properties of distributed programming, and different proof techniques are distinguished by the elegance with which they can be used for solving one kind of problem or another. Unfortunately, there is no simple way in which all the desirable features of these proof systems can be combined without introducing defects not present in the original systems and no single proof system has as yet been constructed with the properties that a software engineer would regard as essential in the complex task of writing correct, large, distributed programs.

In practice, large distributed programs consist mainly of sequential code and there is the well-established

* Supported in part by Visiting Fellowships awarded by the Science and Engineering Research Council (research grant GR/D 90918) and the British Council.

and well-studied Hoare logic of sequential programs available for developing such code. Though, as it stands, Hoare logic is insufficient for dealing with the non-local properties of distributed programs its motivations are intuitively compelling and it is attractive to consider the use of a similar proof framework for all the constructs of such programs. This has been partly accomplished (see for example, Apt et al[1] or Levin and Gries[12]) but only by losing the ability to partition the development and proof of a distributed program into separate units and the costs of establishing that proofs "cooperate" or that they are free from interference are rather substantial. It is important to retain the same modularity in the proof as there is in the program and, with this, to be able to reason about deadlock freedom, livelock freedom and global termination. It should be possible to abstract from the environment of a distributed program the properties that effect the behaviour of a process and then to base the development and the proof of the process on this restricted abstract view.

This paper describes a proof system which attempts to meet all these requirements. From one point of view, we would be justified in saying as Apt did in another context [2] that the work contains "hardly any new ideas". But it would be deceptive to suggest that our proof system results merely from the combinations of ideas from other proof systems: rather, we consider it as part of an effort being made at several places to devise effective and usable proof systems for distributed programming languages. Without much hesitation we can put forward the following claims about this proof system: that it offers a specification and verification technique for establishing total correctness of distributed programs written in a language permitting synchronous and asynchronous communication, for proving absence of deadlock and livelock, and for dealing with server processes. The proof system is syntax-directed and compositional, allows network abstraction, and supports a top-down decomposition of a problem into a distributed programming solution.

2. An Informal Introduction to Process Specification

A distributed program consists of a number of disjoint processes which communicate with each other. For the program as a whole to reach a final objective, each process must be designed to exhibit a behaviour which depends both on its local properties (or its implementation) and on its interactions with its environment. It should be possible to determine this behaviour from a study of the syntactic structure of a process, given a specification of its global interactions. One way of accomplishing this is to formally characterize the *dependence* of a process on its environment in its specification.

Let us define the specification of a process S as the triple:

{Pre} S {Aff}

where the *affirmation* Aff defines the desired behaviour of the process and the *presupposition* Pre abstractly characterizes the dependence of the process on its environment for achieving this behaviour. For sequential programs, this form of specification reduces to the well-known Hoare triples

{P} S {Q}

where P and Q are formulae in first order logic with program variables. This specifies that program S starting in a state satisfying precondition P will terminate in a state satisfying postcondition Q (hence, the specification is really in Hoare logic extended for total correctness). Q specifies the desired final values of the variables and P gives the necessary initial conditions under which this final state may be achieved.

Preconditions and postconditions can be used to specify the desired behaviour of a distributed process if these assertions are extended to include the *communication histories* of processes, where a communication history preserves the sequence of values transmitted along each channel between pairs of processes. For example, the communication history for a channel C could be C=3^4^5, which says that the sequence of values 3, 4 and 5 were sent along the channel in that order.

We shall use sequences to represent histories over channels, with the following operations on sequences.

$Seq_1 \preceq Seq_2$ *iff* Seq_1 is an initial subsequence of Seq_2

$Seq_1 \prec Seq_2$ *iff* $Seq_1 \preceq Seq_2 \wedge Seq_1 \neq Seq_2$

#Seq is the length of sequence Seq

λ is the empty sequence

Seq[i] gives the *i* th element of the sequence

$Seq_1 \wedge Seq_2$ is the catenation of Seq_1 and Seq_2

There are some obvious properties of channels: eg, the channels of a process are empty when it begins execution, and its channel histories will not change after it terminates; each communication along a channel adds a value to the channel history.

Example: A process D sends a sequence of n integers along a channel in_1 be sorted in descending order by a sorter S. It then interrogates S along channel out_1 to read back the sorted integers: when it has received all n values, it terminates.

The sorter S is organized internally as an array of n processes S_1 .. S_n. Each process S_i receives a sequence of n-i+1 integers from its left neighbour along channel in_i, keeps the largest number it receives and sends the rest to its right neighbour along channel in_{i+1}. Sorted integers are returned along the channel out_i: process S_i will first send its own stored value and then the values returned through its right neighbour along channel out_{i+1}.

Initially, all the channels are empty.

$$pre_D \triangleq in = out = \lambda$$

Process D will terminate after the n values have been received sorted in descending order:

$$post_D \triangleq out = sort(in)$$

This postcondition can however be realized by D only by assuming that the sorter will perform correctly: that its output is a sorted permutation of its input. This assumption $assume_D$ must be specified as an invariant for the environment as it holds at each point in the execution of the process.

$$assume_D \triangleq out \preceq sort(in)$$

Considering now the sorter S, it too will start with all its channels empty.

$$pre_S \triangleq in = out = \lambda$$

S will behave correctly only if it is sent all the numbers to be sorted before it is interrogated, and if no more than n numbers are sent. This constraint can be represented by the invariant

$$pump(in,out,n) \triangleq \#in \leq n \wedge \#out > 0 \Rightarrow \#in = n$$

This must be the assumption $assume_S$ for S and process D must make a commitment $commit_D$ that it is met:

$$assume_S \triangleq pump(in,out,n)$$

$$commit_D \triangleq pump(in,out,n)$$

A commitment is an invariant describing the behaviour of a process at *any* point in its execution, while the postcondition describes only the final state of the process. So, a commitment for the sorter would be

$$commit_S \triangleq out \preceq sort(in)$$

These specifications show that process D and the sorter are intended to operate cooperatively, with their commitments satisfying their mutual assumptions. Also, process D will terminate in a state satisfying the postcondition provided it does not deadlock, ie wait indefinitely for a communication that will not take place. So the specifications are adequate to establish *weak total correctness:* to show that any execution of a process for which its assumption is invariantly true and its precondition is initially true *will* maintain its commitment as an invariant and *will* terminate satisfying the postcondition unless it deadlocks.

Freedom from deadlock is a property that must be established from the global interactions between processes. This requires a specification of all the communication actions that a process is willing to perform at each point in its execution and is defined in its *liveness invariant.*

Let en(in?) be true for process S if it is ready to read a value on channel in and let en(out!) be true if S is willing to send a value on channel out. Assume u_S is true when S has terminated. Liveness invariants for process D and S can be written in terms of these functions:

$$L_D \triangleq \begin{cases} \#in < n \Rightarrow en(in!) \wedge \\ \#in = n \wedge \#out < n \Rightarrow en(out?) \wedge \\ \#in = \#out = n \Rightarrow u_D \end{cases}$$

$$L_S \triangleq \begin{cases} \#in < n \Rightarrow en(in?) \wedge \\ \#in - \#out > 0 \Rightarrow en(out!) \end{cases}$$

The complete *presupposition* of a process consists of a precondition and an assumption. The *affirmation* consists of a postcondition, a commitment and a liveness invariant, and the specification gives weak total correctness.

In some cases, processes are designed to execute for ever, for example in operating systems and implementations of network protocols. Clearly, weak total correctness which requires that the process execution be finite is not adequate for specifying such nonterminating processes. The specification must allow the process to communicate indefinitely. Let *divergence* be defined as an infinite sequence of consecutive internal transitions. Then for nonterminating processes we must ensure that their execution is divergence-free: ie, that each process will be ready to communicate within a finite time. Thus, for infinite processes we use a divergence-correctness specification: any execution of the process for which the precondition holds initially, and the assumption is invariantly true will maintain its commitment and the liveness invariant; and *will* be divergence free. The postcondition will hold if the process terminates.

Later sections give a precise formal interpretation of such a specification and show how, given the specification of each process in a network, it is possible to check that the assumption of each process is satisfied, and how it can be shown that deadlocks do not occur. The same form of specification allows a subnetwork to be considered as a single process by hiding the internal communications in the network.

3. The Distributed Programming Notation

Consider a notation for distributed programming based on the notion of a *process.* The simplest processes consist of one of the commands *skip, assignment, input* and *output* and these are called primitive processes. Initially, we shall assume that input and output are synchronous processes whose actions occur simultaneously and indivisibly. (We shall later consider asynchronously communicating processes).

Primitive processes can be combined into more complex processes using constructors: for example, the constructors *sequential composition,* **if** and **while** can be used to build new processes out of simpler processes. If the simpler processes are sequential processes, then the new processes are also sequential

processes. But parallel composition is also a constructor and can be used like any other constructor to build a new process: in this case, the component processes are executed in parallel. Constructors can thus be freely intermixed to create new sequential or parallel processes.

The notation described here is closely related to Occam (INMOS) and CSP (Hoare, 1978) so we call it Occam-S. It allows us to describe programs consisting of disjoint processes communicating with each other through named channels. An Occam-S program is itself a process, but it is called a *closed* process as it refers to no processes other than its components. Thus, it can be considered as a hierarchy of processes, and the structure of the hierarchy is syntactically determined.

The **if** process is similar to Dijkstra's nondeterministic alternative command: it consists of a number of guarded processes, each having a guard b_i and a component process S_i. The guards are executed and one of those that evaluate to true is chosen nondeterministically and its component process is then executed. If all the guards of an **if** process evaluate to false then the process *deadlocks*. The **while** process is executed repeatedly as long as its boolean condition is true.

The parallel composition constructor // is used to produce a *network:* for example, $S_1//...S_k//[S_{k+1}]//...//[S_n]$, where processes $[S_{k+1}],...,[S_n]$ are called servers whose component processes S_1 .. S_n are executed in parallel. The network terminates when all its nonserver component processes have terminated. Servers, like monitors in concurrent programs, provide services to other processes in the network through their communications. Servers are local to a network: they are created with the network, they die with the network and they can communicate only with the processes of the network. Servers do not have to be programmed to terminate as they are automatically destroyed with the network.

Processes communicate with each other by sending and receiving data through *ports* which are connected to named *channels*. The input process C?x reads a value into a variable x from its port C? which is connected to channel C. The output process D!e sends the value of the expression e through its port D! along channel D. A channel in Occam-S connects exactly two processes through an input port of one process (eg, C?) to an output port of another process (eg, D!) and can carry data in only one direction. Channels may differ in their characteristics: here we consider communication which is *synchronous*, i.e. input and output over a channel occurring at the same time. This means that the sending process must wait until the receiving process is ready to receive data, and vice versa.

The **alt** constructor enables a process to be ready to communicate through any one of a number of ports. The components of an **alt** process are guarded processes each having a guard $b_i;c_i$ and a component process S_i. A guarded process is *enabled* if its condition b_i is true; it is *ready to execute* if it is enabled and its communication c_i can be performed. The **alt** process waits if none of its guarded processes is ready to execute. If one or more of its guarded processes is ready to execute, exactly one is executed: the communication of this guarded process is followed by the execution of its component process. If none of the guarded processes is enabled, the **alt** process deadlocks.

The ports and variables of a par-process can be accessed by its component processes provided each such port and variable is accessed by at most one such process. For example, a component process can communicate with a process outside its network using a port inherited from its par-process and connected to an external channel. A channel connecting two processes within a network is called an internal channel.

We can now define an abstract syntax for Occam-S. Let S range over processes, e over expressions, x over program variables PVAR, $\bar{x}$ over variable lists, e over integer expressions, b over boolean expressions, C,D over ports and c_i over communication commands C?x and C!E.

<process> ::= **skip** | x:=e | C?x | C!e |

seq S_1 ; S_2 ...; S_n **end** |

if $b_1 \rightarrow S_1 \,\square\, \cdots \,\square\, b_n \rightarrow S_n$ **fi** |

while b **do** S **od** |

alt $b_1 ; c_1 \rightarrow S_1 \,\square\, \cdots \,\square\, b_n ; c_n \rightarrow S_n$ **end** |

par NET **end**

<network> ::= $S_1 \,//\, \cdots \,//\, S_k \,//\, [S_{k+1}] \,//\, \ldots \,//\, [S_n]$

where $0<k\leq n$.

Let PVAR and PPORT be the sets of program variables and ports respectively. A *basis* is a triple (I,O,V) where $I,O \subseteq$ PPORT, $V \subseteq$ PVAR and I,O,V are finite. With each process we associate a basis B, denoted by B::S, indicating that all the input ports, output ports and program variables used by S are in I, O and V respectively. Note however that B may contain ports and variables not used in S. The basis of a network is designated by

$$B::NET = (B_1::S_1 // \ldots // B_n::[S_n]) \quad \text{where } B = \bigcup_{i=1}^{i=n} B_i$$

The following rules formalise the (syntactic) constraints on the use of ports and variables. A process (I,O,V)::S is said to be well-formed if each of its ports is used exclusively for either input or output.

$$I \cap O = \emptyset \tag{1a}$$

Similarly, the network $B::NET = (B_1::S_1 // \ldots // B_n::[S_n])$, where $B_i=(I_i,O_i,V_i)$ and B=(I,O,V) is well formed if

$$\begin{aligned} &\forall i,j,\ 1\leq i,j\leq n,\ i\neq j:\ B_i \cap B_j = \emptyset \\ &\forall k<j\leq n:\ V_j = NULL\ ,\quad I_j, O_j \subseteq I \cap O \end{aligned} \tag{1b}$$

That is, no input port, output port or variable can be shared between processes; all servers are local to a network. Within a network, ports with matching names (eg, C? and C!) are implicitly connected by synchronous channels of the same name (ie, C).

Let (I,O,V)::NET be a well formed network. Its par process B'::**par** NET **end** is well formed if:

$$(I-O, O-I, V) \subseteq B' \tag{1c}$$

Note that B' may contain a port with the same name as an internal channel: this will then represent a distinct external port, as the **par** constructor automatically "hides" all the internal channels.

Example

({C,E}, {D}, {x,y}):: seq C?x; D!x*x end

Basis(**seq** C?x; D!x*x **end**) = ({C}, {D}, {x})

In the rest of the paper, we shall use S to designate the well formed process (I,O,V)::S with $CH = I \cup O$. Similarly NET will designate the well formed network $B::(B_1::S_1 // \ldots // B_n::S_n)$ with B=(I,O,V), $INT = I \cap O$, $EXT=(I-O)\cup(O-I)$ and $CH=I\cup O$. In this notation, operations on sets have been used as operations on bases in a pointwise manner. For example, if B1 = (I1,O1,V1) and B2 = (I2,O2,V2) then

$B1 \cap B2 = (I1 \cap I2,\ O1 \cap O2,\ V1 \cap V2)$

Also $(\emptyset,\emptyset,\emptyset)$ will usually be abreviated to $\emptyset$.

Occam-S differs from CSP in not permitting "mixed guards". Thus, a CSP program fragment of the form:

if

$b_1 \Rightarrow S_1 \,\square \cdots \square\, b_k \Rightarrow S_k \,\square\, b_{k+1}\,;\, c_{k+1} \Rightarrow S_{k+1} \,\square \cdots \square\, b_n\,;c_n \Rightarrow S_n$

fi

would be rewritten in Occam-S as

if $b_1 \Rightarrow S_1 \,\square \cdots \square\, b_k \Rightarrow S_k$

$\square$ BB => **alt** $b_{k+1}\,;\, c_{k+1} \Rightarrow S_{k+1} \,\square \cdots \square b_n;c_n \Rightarrow S_n$ **end**

fi

where $BB = b_{k+1} \vee \cdots \vee b_n$

The Occam-S notation is as expressive as CSP but it is a subset of Occam in which features dealing with time (like WAIT, AFTER, NOW) have been removed; procedures, replicators and variable declarations have also been omitted but this is for convenience as they are not central to the main concerns of this paper. Unlike Occam, the notation does allow input and output communications in guards, channel declarations are implicit and the **if** construct has been extended to allow nondeterminism.

4. Semantics

The behaviour of OCCAM-S processes can be formally described in terms of an operational semantics [18] [19] in the style of Plotkin. The semantics is formulated as a labelled transition system, where each point in the execution of a process (or a network) is represented by a configuration and one step of the execution of the process is represented by a transition.

A configuration consists of S, which is the remainder of the program still to be executed, and the current state σ, giving value of the program variables. A special symbol "E" designates the empty (terminated) process. The set of configurations $CONF_B$ is defined below. For simplicity, we assume that all variables take integer values.

Let Z be the set of values taken by the variables. Given a basis B=(I,O,V), we define

$\sigma \in \; STATE_B = (V \rightarrow Z)$

$S \in \; PROCESS_B = \{\, S \mid B::S \text{ is a well formed process } \} \cup \{E\}$

$NET \in NETWORK_B = \{NET \mid B::NET \text{ is a well formed network}\} \cup \{E\}$

$\tau \in \; CONF_B = (PROCESS_B \cup NETWORK_B) \times STATE_B$

The set of terminal configurations, representing a terminated process, is given by:

$TCONF_B = E \times STATE_B$

A process moves from one configuration to another on carrying out an action. The transition is labelled by the type of action: an internal action (denoted by '*') or a communication. An internal action is a local action of the process (such as assignment) which is not affected by any external process. A communication action in a process requires the participation of another process and so is explicitly recorded by a suitable label in the transition:

C?v represents the input of a value v over an external input channel C
C!e represents the output of expression e over an external output channel C
C.v represents the communication of a value v over an internal channel C

The set of labels is given below.

$v \in LABEL_B = (I - O) \times \{?\} \times Z \cup (O - I) \times \{!\} \times Z \cup (I \cap O) \times \{.\} \times Z \cup \{*\}$

The semantics is given by a transition relation $\rightarrow_B \subseteq CONF_B \times LABEL_B \times CONF_B$. A transition is represented as $B:: \tau \xrightarrow{v} \tau'$, where $\xrightarrow{v}$ is the transition labelled by v, $\tau,\tau' \in CONF_B$ and $v \in LABEL_B$. We will usually omit mention of B in a transition. We also assume the following identities:

E;S=S, seq E end=E, par E end=E

and

$$\langle \textbf{skip},\sigma\rangle \xrightarrow{*} \langle E,\sigma\rangle \tag{1}$$

The process **skip** starting in a state σ will terminate in the same state in one step. Further, this action is an internal action.

Let $\sigma[d/x]$ represent the state obtained by replacing the value of x by d in σ. The transition rules for assignment, simple communication, sequential composition and **if** follow.

$$\langle x:=e,\sigma\rangle \xrightarrow{*} \langle E,\sigma[\sigma(e)/x]\rangle \tag{2}$$

$$\langle C!e,\sigma\rangle \xrightarrow{C!\sigma(e)} \langle E,\sigma\rangle \tag{3}$$

$$\langle C?x,\sigma\rangle \xrightarrow{C?d} \langle E,\sigma[d/x]\rangle \qquad \forall\, d \in Z \tag{4}$$

$$\frac{\langle S_1,\sigma\rangle \xrightarrow{v} \langle S_1',\sigma'\rangle}{\langle S_1;S_2,\sigma\rangle \xrightarrow{v} \langle S_1',\sigma'\rangle} \tag{5}$$

$$\frac{\langle S,\sigma\rangle \xrightarrow{v} \langle S',\sigma'\rangle}{\langle \textbf{seq } S \textbf{ end},\sigma\rangle \xrightarrow{v} \langle \textbf{seq } S' \textbf{ end},\sigma'\rangle} \tag{6}$$

$$\frac{\sigma(b)=\text{true}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi},\sigma\rangle \xrightarrow{*} \langle S_1,\sigma\rangle} \tag{7}$$

$$\frac{\sigma(b)=\text{false}}{\langle \textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ fi},\sigma\rangle \xrightarrow{*} \langle S_2,\sigma\rangle} \tag{8}$$

In the following rules for **alt** and **while** , let c_i be a communication command of the form C?x or C!e.

$$\frac{\sigma(b_i)=\text{true},\ \langle c_i,\sigma\rangle \xrightarrow{v} \langle E,\sigma'\rangle}{\langle \textbf{alt} \underset{i=1}{\overset{n}{\Box}} b_i;c_i \rightarrow S_i \textbf{ end},\sigma\rangle \xrightarrow{v} \langle S_i,\sigma'\rangle} \tag{9}$$

$$\frac{\sigma(b)=\text{true}}{\langle \textbf{while } b \textbf{ do } S \textbf{ od},\sigma\rangle \xrightarrow{*} \langle S;\textbf{while } b \textbf{ do } S \textbf{ od},\sigma\rangle} \tag{10}$$

$$\frac{\sigma(b)=\text{false}}{\langle \textbf{while } b \textbf{ do } S \textbf{ od},\sigma\rangle \xrightarrow{*} \langle E,\sigma\rangle} \tag{11}$$

Consider a well formed network B::NET = (S_1//.../S_k//[S_k]//[S_n]) with B_i::S_i. The transitions of the network can be defined in terms of the transitions of its processes. We give an *interleaving* semantics for parallelism, i.e. we assume that only one action can occur at a time in a network of parallel processes, and that two independent actions in two different processes occur in some arbitrary but fixed order.

$$\frac{\begin{array}{l} B_i :: \langle S_i,\sigma_i\rangle \xrightarrow{v} \langle S_i',\sigma_i'\rangle \\ \text{where } v = * \text{ or C?d or C!d, and C} \notin \text{INT} \\ S_k=S_k',\ \sigma_k=\sigma_k' \text{ for } k\neq i \end{array}}{B :: \langle S_1//.../[S_n] , \sigma_1\times\cdots\times\sigma_n\rangle \xrightarrow{v} \langle S_1'//.../[S_n'] , \sigma_1'\times\cdots\times\sigma_n'\rangle} \qquad (12)$$

$$\frac{\begin{array}{l} B_i :: \langle S_i,\sigma_i\rangle \xrightarrow{C?d} \langle S_i',\sigma_i'\rangle \\ B_j :: \langle S_j,\sigma_j\rangle \xrightarrow{C!d} \langle S_j',\sigma_j'\rangle,\ i\neq j \\ S_k=S_k',\ \sigma_k=\sigma_k' \text{ for } k\neq i,j \end{array}}{B :: \langle S_1//.../[S_n] , \sigma_1\times\cdots\times\sigma_n\rangle \xrightarrow{C.d} \langle S_1'//.../[S_n'] , \sigma_1'\times\cdots\times\sigma_n'\rangle} \qquad (13)$$

This states that an internal communication C.d can occur only by simultaneous input and output by two processes S_i and S_j of the network over the internal channel C.

The network termination rule given below states that the network terminates when all its non-server processes have terminated. The local state of servers is irrelevant on termination.

$$\langle E//.../E//[S_{k+1}]//.../[S_n] , \sigma_1\times\cdots\times\sigma_n\rangle \xrightarrow{*} \langle E, \sigma_1\times\cdots\times\sigma_k\rangle \qquad (14)$$

Note that no priority is given to this transition and the servers can continue to execute as long as possible. However, if all the servers are deadlocked while all the non-server processes have terminated then this transition must be performed as it is the only possible action.

Finally, the transitions of a **par** process can be derived from the transitions of its network:

$$\frac{B :: \langle NET,\sigma\rangle \xrightarrow{v} \langle \textbf{par } NET' \textbf{ end},\sigma'\rangle \quad \text{where } v = * \text{ or C?d or C!d}}{B' :: \langle \textbf{par } NET \textbf{ end},\sigma\rangle \xrightarrow{v} \langle \textbf{par } NET' \textbf{ end},\sigma'\rangle} \qquad (15)$$

$$\frac{B :: \langle NET,\sigma\rangle \xrightarrow{C.d} \langle NET',\sigma'\rangle}{B' :: \langle \textbf{par } NET \textbf{ end},\sigma\rangle \xrightarrow{*} \langle \textbf{par } NET' \textbf{ end},\sigma'\rangle} \qquad (16)$$

The internal communications of the network cannot be observed outside the network (Rule 16) and must be considered as internal actions of the **par** process.

A configuration is called *terminal* if it has the form $\langle E,\sigma\rangle$. It is called *deadlocked* if there exist no τ',v such that $\tau \xrightarrow{v} \tau'$. A transition with $\tau \xrightarrow{*} \tau'$ with label * is called an *internal transition*. A configuration τ is called a *blocked configuration* if there is no τ' such that $\tau \xrightarrow{*} \tau'$.

The *choice set* of a configuration gives the set of actions possible from that configuration. Let $v \in LABEL_B$, $\tau \in CONF_B$. Let ch(C?d)=C?, ch(C!d)=C!, ch(C.d)=C?,C!, ch(*)=* . Then

$$choice(\tau)=\bigcup_{v}\{ch(v) \mid \tau \xrightarrow{v} \tau', \text{ for some } \tau'\}$$

It can easily be shown that a configuration τ is deadlocked *iff* choice(τ)=$\emptyset$ and it is blocked *iff* * $\notin$ choice(τ). Thus, the choice set adequately represents the deadlock behaviour of processes.

If we can make the weak assumption that a process (or a network) *must* carry out one of its possible transitions in finite time then it will continue to progress until it has reached either a deadlocked or a terminal configuration. When there is more than one transition possible, the choice between them is nondeterministic. We do not make any assumptions about fairness.

An execution over a basis B is a sequence of transitions $\Psi=\tau_1 -v_1\rightarrow \tau_2 -v_2\rightarrow \tau_2 \cdots$ which is either infinite or in which the last configuration is terminal or deadlocked.

$$\mathrm{Dom}(\Psi) = \{1..n\} \text{ if } \Psi = \tau_1 -v_1\rightarrow \tau_2 -v_2\rightarrow \tau_2 \cdots \tau_n$$
$$\{i \mid i>0\} \text{ if } \Psi \text{ is infinite}$$

An execution is said to *diverge* if it contains an infinite subsequence of consecutive internal transitions.

Definition (Semantics): The semantics of a process B::S is given by

$[B::S]=\{\Psi \mid \Psi$ is an execution starting in a configuration $<S,\sigma>$, where $\sigma \in STATE_B\}$

5. Describing the behaviour

As can be seen from the example, the sequence of communications carried out by a process reflects an important aspect of its behaviour. Let the communication of the value d over a channel C be represented as a tuple C.d . A *finite* sequence of such communications is called a trace. We assume the following standard functions on finite sequences.

first(seq)	gives the first element of the sequence seq
<a>	gives a unit length sequence containing a
rest(seq)	gives the sequence obtained by removing the first element from seq

If tr is a trace and CS a set of ports, then $tr\uparrow CS$ represents the trace obtained by deleting from CS the communications which are not over the ports of CS. For example

$(<C.3>\hat{}<D.4>\hat{}<C.2>)\uparrow C? = <C.3>\hat{}<C.4>$

Consider an execution $\Psi=\tau_1 \cdots \tau_n \cdots$. At any point $k \in \mathrm{Dom}(\Psi)$ we can make an observation which contains

(1) The state of the process, i.e. the values of all its variables.

(2) The communication trace of the process, i.e. the sequence of communications carried out by the process so far, assuming some initial trace h_0.

(3) The set of enabled actions of the process, i.e. which of the process ports are enabled for communication and whether an internal action (designated by *) is enabled.

(4) Whether the process has terminated: for a network, we record additionally which of its processes have terminated.

Assuming that the initial trace is h_0, let $\mathrm{Obs}(\Psi,h_0,k)$ denote the observation at point k of the execution Ψ.

Since an observation contains many types of entities, we will use formulas of many-sorted first order logic to specify sets of observations (Enderton[6]). We assume that the program variables PVAR take values of some data type called DTYPE. The formulas also contain variables of the form h_{CS}, where CS is a set of ports, denoting the sequence of communications carried out by the process on the ports in CS. Such variables are called trace projections (Zwiers[23]). As before, the predicate en(P) indicates whether the associated process is willing to communicate through port P and the boolean variable u_S signifies whether the process S has terminated. The specification also includes *logical variables* used in assertions but not in programs. They differ from the auxiliary variables used in other proof systems (Owicki[16] , Apt[1] , Levin

and Gries[12]) in that no values can be assigned to them.

Let LVAR={a,b,c...} denote the logical variables taking values of DTYPE. Variables TVAR={t_1,t_2 $\cdots$} take traces (finite sequences of communications) as their value. IVAR={m,n...} have *standard* natural numbers NAT as values. We also assume that the standard functions 0, 1, + and * over NAT are available.

The assertions are formulas of many-sorted first order logic. Other functions and predicates can be defined in terms of the functions and predicates given above. Thus #seq denoting the length of the sequence seq is defined by the axioms:

$\#\lambda = 0$

$\#<a>\hat{}t = 1+\#t$

$seq_1 \underline{\propto} seq_2 \ \underline{\Delta}\ seq_1 \propto seq_2 \vee seq_1 = seq_2$

Similarly, seq[i] can be defined to denote the i'th element of seq and val*(tr) to denote the sequence of values in trace tr.

Examples

(1) $x=a \wedge y=x+1$

(2) $h_{C?,D!} = <C.2>\hat{}<D.2>\hat{}<D.3>$

We will usually drop parentheses from the set of ports and from the singleton lists, and write (2) as

(2') $h_{C?,D!} = C.2\hat{}D.2\hat{}D.3$

(3) $h_{D!} \underline{\propto} h_{C?}$

and we will abreviate val*(h sub {C?}) as C?. Thus (3) can also be rewritten as

(3') $D! \underline{\propto} C?$

(4) $D! \propto C? \Rightarrow en(D!)$

(5) $D!=C? \Rightarrow u_S$

We can associate a basis B=(I,O,V) with each formula P, denoted by B::P such that the input channels, output channels and program variables referred to in P are from I, O and V respectively. fv(P) designates the set of free variables of P.

Definitions

(1) *Variable substitution* : Let $q[\bar{e}/\bar{x}]$ represent the predicate obtained by simultaneously substituting all the free occurrences of the variables in $\bar{x}$ by the corresponding expressions from the list $\bar{e}$. It must be ensured by systematic renaming of bound variables that no free variable in e is bound.

(2) *Trace extension* : Let P contain trace projections $h_{S1}, \ldots, h_{Sn}$. Then P[h^C.v /h] represents the predicate obtained by replacing each h_{Si} for which C$\in$ Si by $h_{Si}\hat{}C.v$ in P.

(3) *Channel renaming* : Let P[$\overline{nchl}/\overline{chl}$] represent the predicate obtained by simultaneously renaming the channels in the list chl by the corresponding channels in the list nchl. Similarly, the renaming of channels of a process S[$\overline{nchl}/\overline{chl}$] can also be defined.

Let us denote by θ the assignment of values of appropriate type to the logical variables and by Θ the set of all such assignments. Given an observation δ and a logical assignment θ we can evaluate the truth value of a formula P. Let $<\delta,\theta>\perp P$ denote that P evaluates to true in the valuation $<\delta,\theta>$.

6. P-A logic

A P-A formula (presupposition-affirmation formula) has the form

$$B:: \left[A : p \right] \; S \; \left[q : C : L \right]$$

where the presupposition comprises of the invariants p the pre-condition and A the assumption. The affirmation contains the postcondition q, the commitment invariant C and the liveness invariant L. The basis B gives the set of input ports, output ports and variables used in the assertions or the program. The following syntactic restrictions must hold:

(i) $fv(A,C,L) \cap PVAR = NULL$

(i) variables of the form u_S and predicates of the form en(P) do not occur in (A,C,p,q)

In the subsequent discussion, we will use the meta-predicate en(*) to indicate that an internal action of the process is enabled.

A specification can be intuitively described to mean the following. Consider any execution Ψ of S starting such that {q} holds initially. A point in the execution Ψ is called *obedient* if the assumption {A} holds continuously from the start of the process to that point. The P-A formula states that the commitment {C} and {L ∨ en(*)} hold at each obedient point; the post-condition {r} holds in the terminal-configuration provided it is an obedient point, and the commitment {C} holds after an external output communication if the point before the communication is an obedient point. We can define two kinds of P-A formulas. A *total correctness* P-A formula will ensure that if all points in an execution are obedient then the execution is *finite*. A *divergence correctness* P-A formula will ensure that if all points of an execution are obedient then the execution is *divergence free*. A divergence correctness P-A formula will be represented as $[B:: \left[A : q \right] S \left[r : C : L \right]]$. The absence of surrounding parentheses will denote a total correctness P-A formula.

It will have been noted that we do not require L to be invariantly true: only (L ∨ en(*)) must be invariant. This means that L describes only the *blocked* configurations of the process. This is adequate as non-blocked configurations cannot be deadlocked. Such a specification leads to considerable simplification in the specification of a network of processes, as we need to describe only the blocked configurations of the network, disregarding its internal communications.

Formally, we can define the interpretation of a P-A formula as follows.

$\Psi \in [B::S], \theta \in THETA, \; h_0 \in TRACE,$

Let τ_k=conf(Ψ,k),and δ_k=Obs(Ψ,h_0,k)

$OBEDIENT(k) = \forall j \leq k : <\theta,\delta_j> \perp A$

Then $B:: \left[A : q \right] \; S \left[r : C : L \right]$ means:

$OBEDIENT(k) \Rightarrow <\theta,\delta_k> \perp (C \wedge (L \vee en(*)) \wedge (u \Rightarrow r)$

$OBEDIENT(k) \wedge \tau_k - C!d \rightarrow \tau_{k+1} \Rightarrow <\theta,\delta_{k+1}> \perp C$

If $<\theta,\delta_k> \perp q$ then, Total correctness : $(\forall j \in Dom(\Psi) : OBEDIENT(j)) \Rightarrow finite(\Psi)$

Divergence correctness: $(\forall j \in Dom(\Psi) : OBEDIENT(j)) \Rightarrow divergence\text{-}free(\Psi)$

Theorem : Any property P such that $A \wedge C \wedge (L \vee en(*)) \Rightarrow P$ will hold throughout the execution of the process in any enviroment satisfying A if the precondition holds initially.

The predicate en(C) indicates that the channel C is not blocked, i.e. a communication can occur on channel C. Note that differs from en(C?) which is true when the input port C is enabled.

Corrolary : For any closed network, if $C_{net} \wedge L_{net} \Rightarrow u_{net} \vee \exists C : en(C)$ then the network is deadlock-free.

7. Proof rules

There is a proof rule for each primitive process and for each constructor; each composite process can be considered to be synthesised from its components using a constructor. The specification of a process can then be proved from the specification of its components using the proof rule for the constructor.

In this section, we give the proof rules for establishing the correctness of a P-A formula. The proof rules for the sequential part of OCCAM-S are similar to the conventional rules in Hoare logic for total correctness (Gries[7]).

7.1. Network composition

Let $B::NET = (S_1//.../S_k//[S_{k+1}]//...//[S_n])$ be a well formed network with $B=(I,O,V)$, $B_i::S_i$. Since the channels we are considering are synchronous, the depatch and receipt of a message must occur simultaneously. Further, such a channel will be ready for communication when both its input port and output port are enabled. Thus, we can characterise a synchronous channel by the following invariances.

$$C? = C!$$

$$en(C) = en(C?) \wedge en(C!)$$

For the network as a whole, the following invariant must hold.

$$SMERGE = \forall C \in INT : C?=C!$$

Let $B_i :: \left[A_i : q_i \right] S_i \left[r_i : C_i : L_i \right]$ be the specification of the process S_i. This specification refers only to the ports and variables of the process, which cannot be accessed by any other process. Hence, execution of any other process in the network cannot "interfere" with the specification of the process (Owicki[15]).

Initially, the precondition of each process of the network must hold. If all the processes cooperate, the commitment of each process will hold invariantly. Hence,

$$q_{net} \underline{\Delta} (\forall 1 \le i \le n: q_i) \wedge SMERGE$$

$$C_{net} \underline{\Delta} (\forall 1 \le i \le n: C_i) \wedge SMERGE$$

The network terminates when all its non-server processes have terminated. We could say that the processes cooperate if their liveness invariants hold and if the post-conditions of the non-server processes hold on termination. Hence,

$$r_{net} \underline{\Delta} (\forall 1 \le i \le k: r_i) \wedge SMERGE$$

$$L_{net} \underline{\Delta} (frl\ 1 \le i \le n: L_i) \wedge SMERGE \wedge u_{net}=u_1 \wedge \cdots \wedge u_k$$

The network may contain external channels, and the network behaviour must be specified with respect to some assumption $B'::A_{net}$ about the behaviours of these channels. Note that the network assumption cannot mention internal ports of the network and must be only over the external ports of the network.

The environment of a process consists of the external channels of the network and the other processes of the network. The behaviour of the other processes in the network can be characterised by:

$$C_{rest}(i) \underline{\Delta} (\forall 1 \le j \le n, j \ne i : C_j) \wedge SMERGE$$

To establish that the processes of the network do cooperate, we must show that the assumption of each process in the network is upheld by its environment, i.e. for each process S_i of the network,

$$C_{rest}(i) \wedge A_{net} \Rightarrow A_i$$

$$q_{net} \wedge A_{net} \Rightarrow A_i$$

A channel connecting two servers of a network is called a *server-channel.* It is sometimes possible for two servers to communicate infinitely on a server-channel, preventing more useful communication with the other processes of the network. To ensure that such *infinite chatter* does not occur we must show that at any point in the execution of the network the number of communications over a server channel is bounded by some total function of the number of communications over non-server ports. Let SERV denote the set of server ports, i.e. ports connected to a server channel, and NSERC denote the set of non-server ports, i.e. the external ports as well the ports connected to non-server channels.

$$\text{nochatter} \underline{\Delta} \forall C \in SERV: \#C \leq f(\sum_{D \in NSERV} : \#D)$$

Thus, the network composition rule can be stated as follows:

$$\frac{\begin{array}{l} B_i :: \left[A_i : q_i\right]\ S_i\left[r_i : C_i : L_i\right] \\ \text{all the processes cooperate} \\ C_{net} \Rightarrow \text{nochatter} \end{array}}{B :: \left[A_{net} : q_{net}\right] \text{NET} \left[r_{net} : C_{net} : L_{net}\right]} \qquad \text{R12}$$

Using this proof rule we can derive weak total-correctness specification of a network of processes. The following corrolary allows us to prove that there is no deadlock in a closed network so that strong total correctness of a closed network can be establised.

Corollary (to theorem): For any closed network, if $C_{net} \wedge L_{net} \Rightarrow u_{net} \vee \exists C : en(C)$ then the network is deadlock-free.

7.2. Network abstraction

An important requirement for a proof system for distributed programs is that it must allow a network of processes to be described as a single abstract process, disregarding the details of its internal communications. The predicates in the external specification of the network must therefore not mention the internal ports of the network.

For a well formed network B::NET such that B=(I,O,V), we define an *external* basis B'::(I–O,O–I,V). The set of internal channels of the network INT is $I \cap O$ and the set of its internal ports CINT is $(I \cap O) \times \{?,!\}$. The internal channels of the network are created with the network and are initially empty.

$$\frac{\begin{array}{l} B :: \left[A : (q \wedge h_{CINT} = \lambda)\right]\ \text{NET}\left[r : C : L_{int}\right] \\ A \wedge C \wedge L_{int} \Rightarrow (L_{ext} \vee \exists C \in INT: en(C)) \\ B' :: (A,q,r,C,L_{ext}) \end{array}}{B' :: \left[A : q\right]\ \textbf{par NET end}\ \left[r : C : L_{ext}\right]} \qquad \text{R13}$$

The second clause in this rule can be explained as follows. The external liveness invariant L_{ext} of the network must describe those blocked states of the process which are unaffected by any possible internal communication because, after network abstraction, such communications must be considered as internal actions.

7.3. Other proof rules

In what follows, the basis of all P-A formulas is assumed to be (I,O,V) with $CH = I \times \{?\} \cup O \times !$. We will not explicitly mention the basis in the formula; u will designate the termination flag of the process.

Definition

$$\left[\, A : q \,\right]\ S\left[\, r : C : \langle L \rangle \,\right] \ \underline{\Delta}\ \left[\, A : q \,\right]\ S\left[\, r : C : L' \,\right]$$

where $L' \ \underline{\Delta}\ L \vee \exists \bar{v} : r \wedge u_S \wedge \forall C \in CH : \neg en(C)$

The P-A formula with <L> describes all the blocked configurations of the process except the terminal configuration. Such a formula is much more convenient to use for sequential composition as can be seen from the sequential composition and while rules below.

$$\frac{q \wedge A \Rightarrow C}{\left[\, A : q \,\right]\ \textbf{skip}\left[\, q : C : \langle L \rangle \,\right]} \qquad \text{R1}$$

$$\frac{q \wedge A \Rightarrow C}{\left[\, A : q[e/x] \,\right]\ x := e\left[\, q : C : \langle L \rangle \,\right]} \qquad \text{R2}$$

The proof of a communication statement is given in two parts. In the first part we verify the correctness of the postcondition r and the commitment C and in the second we check the correctness of the liveness invariant. The conclusion of the first part of the check is written by enclosing the communication in <>. Formally, this is just an intermediate syntactic stage in applying the proof rule.

Let $h' = h \hat{} D.e$

$$\frac{\begin{array}{l} q \wedge A \Rightarrow C \wedge C[h'/h] \\ q \wedge A \wedge A[h'/h] \Rightarrow r[h'/h] \end{array}}{\left[\, A : q \,\right]\ \langle D!e \rangle\left[\, r : C \,\right]} \qquad \text{R3}$$

Let $h' = h \hat{} D.v$, where v is a fresh logical variable.

$$\frac{\begin{array}{l} q \wedge A \Rightarrow C \\ \forall_v : (q \wedge A \wedge A[h'/h] \Rightarrow C[h'/h] \wedge r[h'/h, v/x] \end{array}}{\left[\, A : q \,\right]\ \langle D?x \rangle\left[\, r : C \,\right]} \qquad \text{R4}$$

Let $port(C?x) \ \underline{\Delta}\ C?$ and $port(C!e) \ \underline{\Delta}\ C!$

$$\frac{\begin{array}{c} q \wedge \bar{A} \wedge en(port(comm)) \wedge \forall_{c \in CH,\, c \neq port(comm)} : \neg\, en(c) \Rightarrow L[false/u] \\ \left[\, A : q \,\right]\ \langle comm \rangle\left[\, r : C \,\right] \end{array}}{\left[\, A : q \,\right]\ comm\left[\, r : C : \langle L \rangle \,\right]} \qquad \text{R5}$$

$$\frac{\begin{array}{l} \left[\, A : q \,\right]\ S_1\left[\, r1 : C : \langle L \rangle \,\right] \\ \left[\, A : r1 \,\right]\ S_2\left[\, r : C : \langle L \rangle \,\right] \end{array}}{\left[\, A : q \,\right]\ S_1 ; S_2\left[\, r : C : \langle L \rangle \,\right]} \qquad \text{R6}$$

$$\frac{\left[\,A : q\,\right]\ S\ \left[\,r : C :\!<L>\,\right]}{\left[\,A : q\,\right]\ \textbf{seq}\ S\ \textbf{end}\ \left[\,r : C :\!<L>\,\right]} \qquad \text{R7}$$

$$\frac{\left[\,A : q \wedge b_i\,\right]\ S_i\ \left[\,r : C :\!<L>\,\right] \quad q \wedge A \wedge \neg BB \wedge \forall c \in CH: \neg en(c) \Rightarrow L[false/u]}{\left[\,A : q\,\right]\ \textbf{if}\ \Box_{i=1}^{i=n} b_i \Rightarrow S_i\ \textbf{fi}\ \left[\,r : C :\!<L>\,\right]} \qquad \text{R8}$$

$$q \wedge b \Rightarrow bf > 0, \quad \text{where } bf \in IEXPR$$

$$\frac{\left[\,A : q \wedge b \wedge bf = v\,\right]\ S\ \left[\,q \wedge bf < v : C :\!<L>\,\right]}{\left[\,A : q\,\right]\ \textbf{while b do S od}\ \left[\,q \wedge \neg b : C :\!<L>\,\right]} \qquad \text{R9}$$

$$\frac{\left[\,A :\ q \wedge b_j\,\right]\ <c_j>\ \left[\,r_j : C\,\right] \quad \left[\,A : r_j\,\right]\ S_j\ \left[\,r : C :\!<L>\,\right] \quad \forall T \subseteq [1..n] \quad q \wedge A \underset{j \in T}{\wedge} b_j \wedge en(port(c_j)) \underset{k \in T}{\wedge} b_k \wedge \neg en(port(c_k)) \Rightarrow L[false/u]}{\{A\} : \{q\}\ \textbf{alt}\ \Box_{i=1}^{i=n} b_i;\ c_i \rightarrow S_i\ \textbf{end}\ \{r\} : \{C\}\ \{L\}} \qquad \text{R10}$$

$$\frac{\left[\,A : q\,\right]\ S\ \left[\,r : C : L\,\right]}{\left[\,A : q\,\right]\ S\ \left[\,r : C :\!<L \wedge \neg u>\,\right]} \qquad \text{R11}$$

$$A \Rightarrow A1\ ;\ q \Rightarrow q1\ ;\ r1 \Rightarrow r\ ;\ C1 \Rightarrow C\ ;\ L1 \Rightarrow L$$

$$\frac{\left[\,A1 : q1\,\right]\ S\ \left[\,r1 : C1 :\!<L1>\,\right]}{\left[\,A : q\,\right]\ \left[\,r : C :\!<L>\,\right]} \qquad \text{R15}$$

7.4. Proof rules for divergence correctness

With the exception of the rule for the **while** contructor, the other proof rules for total correctness can be used to combine the divergence correctness specifications of components into the divergence correctness specifications of composite processes. But in order to prove that loop execution is divergence-free we need to show that there cannot be an infinite sequence of loop iterations involving no communication.

$$\frac{q \wedge b \Rightarrow bf > 0 \quad \left[\,A : q \wedge b \wedge (h_{CH} = t) \wedge bf = v\,\right]\ S\ \left[\,q \wedge (h_{CH} = t \Rightarrow bf < v : C :\!<L>\,\right]}{\left[\,A : q\,\right]\ \textbf{while b do S od}\ \left[\,q \wedge \neg b : C :\!<L>\,\right]} \qquad \text{R9d}$$

The new rule is not necessary if all the internal channels of the network can be considered as slave channels. But it can be used to establish the correctness of a server implemented as a network of processes which do not terminate.

8. Asynchronous Communication

Processes can communicate asynchronously with each other if messages sent between them are bufferred. A buffer of finite size puts a limit to the number of messages which can be sent before the first of them is received, but there is no such limit with an infinite buffer. In both cases, the basic problem is to determine how to relate the sending of a message in one process with its receipt by the other process.

We shall assume that each channel has an infinite buffer. Thus, the sender can always send a message without delay, and the receiver need only wait if there is no message in the channel. As before, each channel will connect a pair of processes and messages will be received in the order in which they were sent.

We shall use the same syntax as for synchronous communication with the restriction that output guards are not used in an alt constructor as they are non-blocking. In asynchronous communication, the sending and receipt of a message are distinct events and they will be recorded separately in the communication history as the triples C?d or C!e and a trace could have the form C!2^D!3^C?2. The liveness invariant now need only specify whether input ports are enabled.

An asynchronous channel C can be characterised by the following invariances:

$$C? \underline{\propto} C!$$

$$en(C) \underline{\Delta} C? \propto C! \wedge en(C?)$$

Thus for a network of processes communicating on asynchronous channels the network invariant SMERGE is replaced by

$$ASMERGE \underline{\Delta} \forall_{C \in INT} : C? \underline{\propto} C!$$

The network composition rule and network abstraction rules for asynchronous networks are same as the previous rules R12 and R13 using the new definitions of ASMERGE and en(C).

The proof rules for all the sequential programming constructors remain unchanged, and verification is done in two parts: part 1 uses the proof rules R3 and R, and part 2 is given below.
Let $h' = h\hat{}D.e$

$$\frac{q \wedge A \Rightarrow C \wedge C[h'/h] \qquad q \wedge A \wedge A[h'/h] \Rightarrow r[h'/h]}{\left[A : q \right] \; <D!e> \; \left[r : C \right]} \qquad \text{R3}$$

Let $h' = h\hat{}D.v$, where v is a fresh logical variable.

$$\frac{q \wedge A \Rightarrow C \qquad \forall v : (q \wedge A \wedge A[h'/h] \Rightarrow C[h'/h] \wedge r[h'/h, v/x]}{\left[A : q \right] \; <D?x> \; \left[r : C \right]} \qquad \text{R4}$$

Let $port(C?x) \underline{\Delta} C?$.

$$\frac{q \wedge A \wedge en(D?) \wedge \forall c \in CH, c \neq D? : \neg en(c) \Rightarrow L[false/u] \qquad \left[A : q \right] \; <D?x> \; \left[r : C \right]}{\left[A : q \right] \; comm \; \left[r : C :<L> \right]} \qquad \text{R5a}$$

$$\frac{\left[A : q \right] \; <D!e> \; \left[r : C \right]}{\left[A : q \right] \; comm \; \left[r : C :<L> \right]} \qquad \text{R5b}$$

The other proof rules are the same as for synchronous communication.

9. Conclusions

The last few years have been notable for the development of a variety of different proof systems for what is basically one class of distributed programming languages. Building on the pioneering work of Owicki and Gries [17] which showed that Hoare-style proofs of individual processes in a shared variable parallel program are valid provided the processes can be shown to be 'interference-free', the first of these proof systems came from Apt, Francez and de Roever [1] and from Levin and Gries [12] : the former used local and global invariant assertions and a proof of 'cooperation' between the proofs of individual processes to establish the properties of a program, while the latter extended the work on interference freedom to distributed programs using global auxiliary variables and a proof of 'satisfaction'. Both were concerned with what Lamport[11] has called 'safety properties', which is partial correctness and deadlock freedom, but if the termination of each process could be proved locally the Levin and Gries proof system would also handle total correctness. A different style of proof system was developed by Soundararajan [22] using reasoning over the traces, or communication sequences, of different processes: unlike proofs using satisfaction or cooperation between the proof outlines of different processes, the parallel composition rule in this system requires the communication sequences of different processes to pass a test of 'compatibility'. Each proof system threw some light on the methods that could be used for proving properties of distributed programs, but none of them could be said to lead to a method of modular program construction as any proof required fairly involved global reasoning (as Barringer[4] illustrates well).

Other proof systems have used reasoning over program traces: eg, the work of Misra and Chandy [14] and subsequent developments of this work by Zwiers and de Roever[23] Though these proof systems laid down a possible framework for developing programs and proofs compositionally, they shared with their predecessors a motivation which is primarily *verificational* in the sense that they provided proof rules for specific language constructs.

Given the extent of detail inherent in even simple programming languge constructs, it is difficult to construct a verification method based on proof rules for language constructs which is also sufficiently abstract to allow the effective specification of entire distributed programs. And little acquaintance with these proof systems is needed to realise that the verificational systems are cumbersome to use, even when some rigour is omitted in the interests of convenience.

Using a rather different approach, Hoare [9] has described a total correctness proof system for communicating processes based on traces which is primarily *specificational*, in that it is relatively independent of the constructs of languages and therefore inherently more abstract than verificational proof systems. Specificational systems do have many attractive properties, such as conciseness and compositionality, but when used in a program development system they would need to be supported by some verificational system to guarantee that the programs that are finally produced do in fact conform to the specifications. And it is not clear that the temporal logic based proof systems (Pnueli[20] , Barringer, Kuiper and Pnueli[3]), which in some sense do bridge the gap between verificational and specificational proof systems, offer any significant advantages in terms of simplicity when used for program development.

One way to bring together the requirements for providing abstract specifications and verification rules is for the proof system to make effective use of features for network abstraction and network composition in suggesting a top-down program development method. As we have shown in Section 2, it is possible to specify a distributed program at a level which permits its total correctness to be established without at the same time requiring details of its implementation. These specifications can then be used to define the operations of individual processes, and the proof rules of Section 7 used to verify that the processes are correct. The language used can be rich enough to make use of network abstraction, servers and asynchronous communication and no further check is necessary to establish either "interference freedom" or "compatibility".

This was the aim of this work, and we shall leave the reader to judge how much of it has been achieved. It is clear, nevertheless, that considerably more work is necessary in using this method for more substantial problems so that its effectiveness and its limitations can be studied.

References

1. K.R. Apt, N. Francez, and W.-P. de Roever, "A Proof System for Communicating Sequential Processes," *ACM Trans. Progr. Lang. & Syst.* 2(3) pp. 359-384 (1980).
2. K.R. Apt, "A Static Analysis of CSP Programs," in *Proc. Logics of Programs, Pittsburgh*, ed. E. Clarke, D. Kozen,Springer-Verlag LNCS 164, Heidelberg (1984).
3. H. Barringer, R. Kuiper, and A. Pnueli, "Now You May Compose Temporal Logic Specifications," in *Proc. 16th ACM STOCS*, , Washington (May 1984).
4. H. Barringer, *A Survey of Verification Techniques for Parallel Programs*, Springer-Verlag LNCS 191, Heidelberg (1985).
5. Z. Chaochen and C.A.R. Hoare, *Partial Correctness of Communicating Processes and Protocols*, Oxford University Computing Laboratory (1981).
6. H.B. Enderton, *A Mathematical Introduction to Logic*, Academic Press (1972).
7. D. Gries, *The Science of Programming*, Springer-Verlag, New York (1981).
8. C.A.R. Hoare, "Communicating Sequential Processes," *Comm. ACM* 21(8) pp. 666-677 (1978).
9. C.A.R. Hoare, "A Calculus of Total Correctness for Communicating Processes," *Sc. Comp. Progr* 1(1,2) pp. 49-72 (1981).
10. J. Hooman and W.-P. de Roever, *The Quest Goes On: A Survey of Proof Systems for Partial Correctness of CSP*, Dept. of Maths. and Comp. Sc., Eindhoven University of Technology, Eindhoven (1986).
11. L. Lamport, "Proving the Correctness of Multiprocess Programs," *IEEE Trans. SE* SE-3(2) pp. 125-143 (1977).
12. G.M. Levin and D. Gries, "A Proof Technique for Communicating Sequential Processes," *Acta. Inf.* 15 pp. 281-302 (1981).
13. INMOS Ltd., *The occam Programming Manual*, Prentice-Hall International U.K. (1983).
14. J. Misra and K.M. Chandy, "Proofs of Networks of Processes," *IEEE Trans. SE* 7(4) pp. 417-426 (1981).
15. S.S. Owicki, *Axiomatic Proof Techniques for Parallel Programs*, Cornell University, Ithaca, N.Y. (1975). Ph.D. Thesis
16. S.S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," *Comm. ACM* 19(5) pp. 279-285 (May 1976).
17. S.S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs I," *Acta Inf.* 6 pp. 319-340 (1976).
18. G. Plotkin, *A Structural Approach to Operational Semantics*, Aarhus University (1981).
19. G. Plotkin, "An Operational Semantics for CSP," pp. 199-225 in *Formal Description of Programming Concepts*, ed. D. Bjorner,North-Holland (1983).
20. A. Pnueli, "The Temporal Logic of Programs," pp. 46-57 in *Proc. 18th Annual Symp. FOCS*, , Providence, R.I. (October 1977).
21. W.-P. de Roever, "The Quest for Compositionality, Part I," in *Proc. IFIP Working Conference on The Role of Abstract Models in Computer Science*, ed. E.J. Neuhold, (1985).
22. N. Soundararajan, "Correctness Proofs of CSP Programs," *Th. Comp. Sc.* 24(2) pp. 131-141 (1983).
23. J. Zwiers, W.-P. de Roever, and P.E.M de Boas, "Compositionality and Concurrent Networks: Soundness and Completeness of a Proof System," in *Proc. ICALP 85*, Springer-Verlag LNCS, Heidelberg (1985).