November, 1984

MODULAR VERIFICATION OF COMMUNICATING SEQUENTIAL PROCESSES

E.A. Akkoyunlu[1] and R.M. Nemes[2]

DCS-TR-150

[1]Department of Computer & Information Science
Brooklyn College of the City University
of New York(CUNY)
Bedford Avenue & Avenue H
Brooklyn, NY 11210

[2]Department of Computer Science
Rutgers University
Hill Center
Busch Campus
New Brunswick, NJ 08903

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

November, 1984

MODULAR VERIFICATION OF COMMUNICATING SEQUENTIAL PROCESSES

E.A. Akkoyunlu[1] and R.M. Nemes[2]

DCS-TR-150

[1]Department of Computer & Information Science
Brooklyn College of the City University
of New York(CUNY)
Bedford Avenue & Avenue H
Brooklyn, NY 11210

[2]Department of Computer Science
Rutgers University
Hill Center
Busch Campus
New Brunswick, NJ 08903

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

ABSTRACT

The semantics of communication in a distributed computing environment without shared objects are investigated from the viewpoint of modularity and hierarchical system structure. *C*ommunicating *S*equential *P*rocesses (CSP), Hoare's language for parallel programming, is modified and expanded to support process modularity and hierarchical structure using a port construction. A formal axiomatic verification methodology for partial correctness is developed that extends the Hoare axiomatic proof methodology for sequential (nonparallel) programs to CSP–like programs, without resort to global invariants. Hierarchical structure and modularity are fully supported within the proof system. Processes are verified against an abstract entity, the interface, thereby achieving a formal notion of process specification and plug–compatibility. Alongside maintenance to a system is maintenance to its correctness proof, the two evolving side by side in an isomorphic fashion. The formalism is broadened further to include shared ports by the introduction of a universal assertion termed Kirchhoff's Law. Several examples are provided that demonstrate the methodology, including a modular proof of the generic single–entry, multiple–user CSP subroutine process.

Table of Contents

List of Figures

1. INTRODUCTION

With the advent of the inexpensive and versatile microprocessor, it has become advantageous to organize computations and systems as large collections of tasks – i.e. processes to be performed concurrently. Certainly it is useful investigating the characteristics of such truly distributed systems consisting of significant numbers of processing elements and to evolve design tools for their development.

The complexity of highly parallel systems increases very rapidly with size. Most notably, the number of interfaces between processes increases quadratically with the number of processes, so it behooves to organize systems in ways that minimize the complexity of the interfaces and that promote modularity. Early developments in this area include the seminal work of Dijkstra [4], who demonstrated how an operating system can be organized and developed in a hierarchical fashion so that it can be implemented and tested incrementally, and Parnas [14], who showed how sequential programs can be decomposed in ways that permit the intermodule interfaces to be defined on a functional rather than implementation basis. The newest, most sophisiticated programming languages now have features, called class, cluster, form, or package, that support data abstraction and encapsulation in order to separate the semantic properties of an abstract construct from its implementation.

What these methods all have in common is that they result in programs and systems in which a change to one module has minimal impact on the remainder of the system. Using this as a fundamental criterion of modularity, this paper explores certain language features and proof techniques that encourage modularity in the verification and specification of concurrent programs. In short it is a study of the semantics of communication with an eye toward modularity and specification.

To begin we choose the small language CSP, *C*ommunicating *S*equential *P*rocesses, defined by Hoare as a theoretical tool for the study of the semantics of parallelism and nondeterminacy. The reader not already familiar with the main features of CSP is referred to [8], [5], and [6]. CSP has several characteristics which make it attractive for the purposes here:

1. As an abstract model it reflects the underlying architecture: small processes with private store;

2. The notion of process is a key element in the language;

3. The emphasis is on simultaneity rather than mutual exclusion;

4. Interprocess communication protocol is simple and is based on the well-understood assignment operator;

5. Nondeterminacy is provided;

6. An elegant proof system for partial correctness of CSP programs has been developed (see [1]);

7. CSP is an excellent specification language.

2. A PROOF SYSTEM FOR CSP

The Apt system [1] is an elegant proof methodology for establishing the partial correctness of CSP programs. The technique, based on the work of Owicki [13], uses the notion of "joint cooperation between isolated proofs" of individual processes to establish program validity.

Apt's system proceeds as follows. In the separate proofs of each process, every statement is preceded by a pre-assertion and followed by a postassertion, referring only to variables of the process in which the statement appears. These assertions must satisfy the axioms and proof rules of purely sequential (non-parallel) CSP, the classic Hoare axiomatic paradigm described in [7]. The postassertion of an I/O command, however, is the exception and cannot be established solely within the local context. It is partly dependent on the pre-assertion of the matching I/O command found in the addressed process. Thus its validity must be established jointly by the two pre-assertions that "cooperate" to establish the two postassertions. As in Owicki's system, a global invariant and auxiliary variables are used throughout the proofs. A well-defined notion of critical section - termed "bracketed section" - delimits a purely sequential neighborhood of each I/O command outside of which the global invariant must hold. Brackets are inserted into the program text as part of the annotated proof.

The global invariant is a generally useful device. It can be used to specify loop invariants, for example, and in many instances an entire proof consists of an extensive global invariant with mostly trivial local assertions. Nevertheless Apt et al. claim that a canonical proof form does exist, in which each bracketed section consists of an I/O command and a local history-

variable update (usually to an auxiliary variable). In this case the global invariant is at its minimal.

3. MODULARITY AND CSP

CSP was designed primarily as a theoretical tool for the study of the semantics of concurrency and nondeterminacy. Although the process structure and communication mechanism do encourage some modularity, it is assumed that all modules and interfaces are known and fully specified at the outset. In this sense CSP functions best when an entire program is written all at once, preferably by one individual. Much of this is attributable to communicating processes addressing one another directly and explicitly, and having a very specific relationship: they must be "constituent processes" of a parallel command.

Of course the situation in any real environment is quite different from the one just described. Invariably specifications are incomplete and vague, any number of programmers participate in program development, there occur small but endless changes to the external environment, changes are continually being made to the program to improve performance, etc. To cope with these difficulties, support at the language level is necessary. Systems designed to function in dynamic environments must be structured in ways that enable them to remain stable and yet maintainable. Effects of any change must be localized as much as possible. On the verification side, *local changes should require only local reverification.*

Let us illustrate some of the problems with structuring programs in CSP as it is currently defined. Consider a pair of parallel processes as shown in figure 3-1.

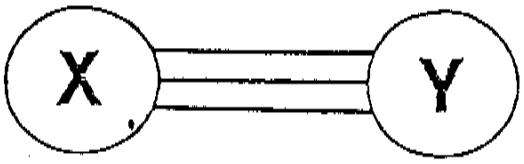

Figure 3-1: [X||Y]

At some point it may be desirable to divide Y into subprocesses as shown in figure 3-2. Such

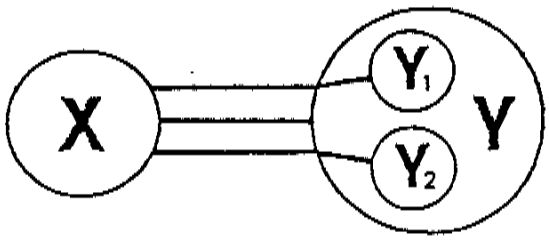

Figure 3-2: [X||Y::[Y_1||Y_2]]

a change should require no modification to X, provided that the interface remains intact, i.e., the external behavior of both versions of Y are identical. CSP clearly does not permit this (see [8]).

As a second example, consider that CSP is an excellent language for specifying the components of a highly distributed operating system and yet it is not possible to write a CSP system that executes CSP user programs. Dynamically created user processes cannot communicate with a static operating system.

The proof system suffers from similar limitations. "Joint cooperation between isolated proofs" has little if any tolerance to even minor changes. In the first example cited above, the entire program – including process X which was not changed in any way – would require reverification. Moreover, the proof system is incapable of validating anything but a complete system in which all modules have been finalized. Our notion of modularity strongly suggests that we have the capability of validating process X against a *specification* of process Y, even before Y has been coded.

CSP proofs are even less tolerant to change than the language itself. Since the global invariant contains control information spanning an entire program, any change that might affect the invariant in the slightest way necessitates revalidation of the cooperation tests between

every pair of syntactically matching I/O commands, even in processes totally unrelated to, and logically removed from, the one that was changed. In a highly parallel system, revalidating every cooperation test may mean essentially re-establishing the entire proof.

4. THE INTERFACE CONCEPT

The notion of interface as it is developed here involves pairs of objects of the form (module.interface). In CSP the module is, of course, the process. Figure 4-1 illustrates the relationship of modules to a common interface.

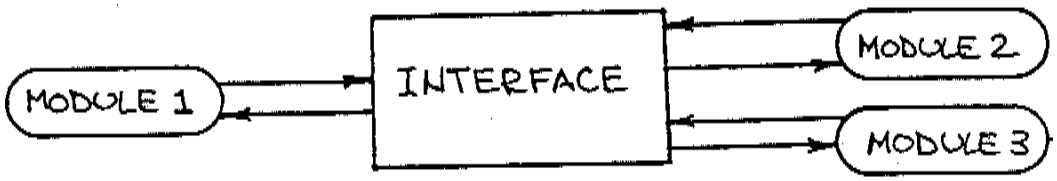

Figure 4-1: MODULE-INTERFACE RELATIONSHIP

The interface component captures the interaction of the module with its external environment, and it itself consists of two components (as is suggested by the pair of arrows attached to each module):

1. The external specification of the module viewed from without, i.e., the functional behavior of the module considered as a black box. From the point of view of systems theory this corresponds to the external description of a dynamical system. In figure 4-1 this is depicted as the arrow pointing away from the module toward the interface;

2. The specification of the external environment viewed from within the module, depicted in figure 4-1 by the arrow pointing away from the interface toward the module. This constitutes the conditions the module expects from any environment in which it expects to operate.

The interface, then, provides the necessary insulation to protect the module from external

changes that should not affect it, and allows modifications to the internal structure that ought not affect the external environment. In this setting the modules are uncoupled from one another and verification can proceed on a truly general level. Verifying a module against an established interface is equivalent to verifying the module against an entire class of programs, infinite in number.

5. PORTS AS REPRESENTATION OF THE INTERFACE

5.1. General Description

This section develops a representation of the abstract interface as previously described. We introduce the notion of a *port* and derive semantics that are consistent with modularity requirements and with the Hoare axiomatic verification methodology. In its most complete generality, the interface is represented by *multi-ports*, whose semantics derive in a natural way from those of the port. Ports and multi-ports are defined precisely, further on in the paper. First, however, we shall characterize the port concept heuristicly.

The port, as the term is used in this paper, differs considerably from constructs of the same name discussed in the literature. Hoare himself in [8] mentions the possibility of using ports, but what he suggests is simply a syntactic device without semantic implication. His ports are named channels between "constituent processes of a parallel command." Kieburtz [9], on the other hand, uses ports as a mechanism to uncouple communication and synchronization in CSP. There, ports are asynchronous data channels that provide data buffering and allow the outputting process to proceed independently of the inputting one. Mao's port [11], termed Communication Port, is a much weighter construct suitable for higher level communication. In our exposition we intend to retain the essence of the original communication semantics inherent in CSP.

Communication in our CSP-like language is based on the following: an I/O command names an abstract entity called a *port*. Thus, process Q issues the command A!y to output the value of local variable y through port A, and process P issues the command A?x to input from port A into local variable x. Ports are declared before the opening left bracket of a parallel command,

A.B.C: port: $[P_1 || P_2 || \ldots || P_n]$.

and have scope consisting of precisely that parallel command. Ports can be used for communication only between constitutent processes of the parallel command that follows the declaration, and futhermore only for communication that crosses a parallel boundary of the parallel command to which the declaration applies. Nested parallel commands must use ports declared locally for local communication that crosses only nested parallel boundaries. Thus, in the following situation communication between P and R (or S) - which we now allow - must proceed through port A or B, while communication between R and S may proceed only through port C:

A,B: port; [P||Q:: C: port; [R||S]]

As in standard CSP, a parallel command terminates when all the constituent processes have terminated. We shall forgo discussion of declaration syntax since we are primarily concerned here with semantic issues (see [12] for syntax details). In all the examples presented in this paper, every port is assumed to have global scope.

We call the collection of ports whose scope is the same parallel command the *port set* of that parallel command. In the example above the port set of the outer parallel command consists of A and B.

A *multi-port* is defined to be a port that is used by more than one process for input, or more than one process for output. The expression "by more than one process" means "by more than one process at the lowest level." Referring again to the example cited above, A is a multi-port if it is used by P, R, and S.

Ports have been introduced as the mechanism used for specifying and developing a system in a modular fashion. Our notion of modularity requires that a system be Hoare verifiable in a modular fashion as well, and that the proof be "maintainable." To this end we shall describe the semantics of the port.

First, consider a process P that inputs through port A into variable x, shown in figure 5-1. L and M are the pre- and postassertions (for proving partial correctness) associated with the I/O command A?x in a local proof of process P. As shown, the port supplies two semantic items to the process P, and P supplies one item to the port A. The port supplies the value to

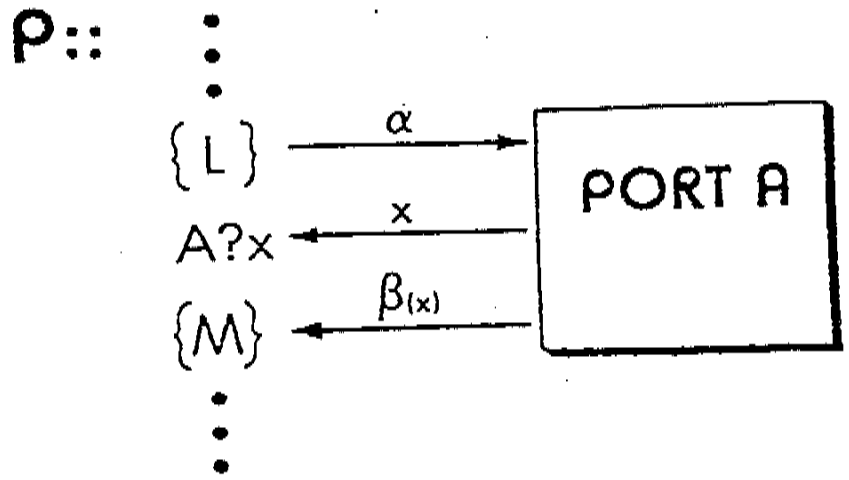

Figure 5-1: INPUT THROUGH PORT A

be input to x and an assertion β(x), guaranteed by the outputting process via the port, that aids in establishing the postcondition M. L and β together establish M. In turn process P must guarantee assertion α, which is used by the outputting process to establish its postcondition.

The case of the outputting process appears diagrammatically dual to the one above and is shown in figure 5-2. Process Q provides the port with the data value y and guarantees assertion β. α assists N in establishing the postcondition R.

Conceptually, β represents data and control information flowing from the output process to the input process, while α represents control information flowing in the opposite direction. The formal definition of *port* associates α and β with the port declaration in the annotated text.

Thus, we see that there are two disjoint semantic domains: the local process environment domain and the interface, or port set, domain. The interface domain contains a separate data base, described below, and interacts with the local process environment via the assertions α and β. At the time of interprocess communication, the port captures data flowing accross an

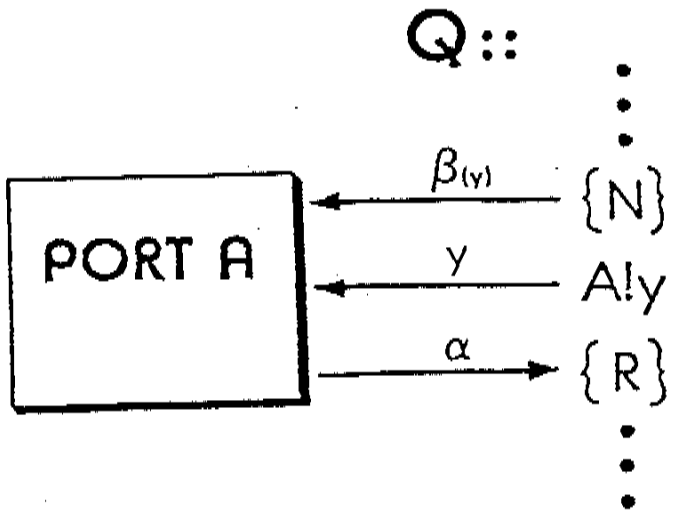

Figure 5-2: OUTPUT THROUGH PORT A

output process boundary and passes it to an input process: simultaneously, the port set data base is updated to reflect the effects of the transaction on the interface environment.

5.2. Formal Description

A *simple* (non-multi) *port* is formally defined as follows.

1. An I/O command through port A takes the form A?x or A!y, where A is the *port name*. x and y must be of the same data type. A process issuing an I/O command naming A is delayed until another process issues a corresponding I/O command also naming A, assuming that A is not closed (explained below).

   Port A is declared to be either of type *static* or *dynamic*. A static port remains perpetually open to transactions and is not affected by the control state of the processes using it. A dynamic port, on the other hand, can become closed to subsequent transactions, depending upon the control state of the user processes. Thus an I/O command naming a static port never evaluates to *false* in a guard or *fails* in straight-line code, while naming a dynamic port A will lead to an evaluation of *false* in a guard or *fail* in straight-line code if any of the following conditions hold:

   a. All processes using A for input have terminated;

b. All processes using A for output have terminated;

c. All but one of the processes using A have terminated.

The first two cases are clear; the third case covers the situation in which only one of the processes refrencing A has not terminated and that process uses A for both input and output. Static ports are clearly cheaper to implement and hence we choose them whenever possible.

2. Port A has associated with it an abstract data object *a* (lower case letter whose uppercase is the port name) of the same data type as x and y. *a* is known as the *port variable*. Port variables lie in a name space that is disjoint from the individual process environments, i.e., processes in the scope of a port are not permitted to mention the port variable in program statements. Synchronization of A?x and A!y is interpreted as atomic execution of the sequence

$$a := y ; x := a \qquad . \qquad (1)$$

3. Port A has associated with it a *port action* S_A that consists of assignment statements to auxiliary variables. Auxiliary variables, like port variables, are confined to port space and may not be mentioned in program statements. An auxiliary variable is associated with a particular port set, i.e., with a particular parallel command, and is declared with the ports of the port set in the full annotated text. The port action executes simultaneously with transactions across the port. Interpretation of the synchronization of A?x and A!y is expanded from (1) to now mean atomic execution of the sequence

$$a := y ; S_A ; x := a \qquad . \qquad (2)$$

The right-hand sides of the assignment statements comprising S_A are expressions mentioning constants, auxiliary variables, and port variables of the port set containing A.

Auxiliary variables and port variables associated with a simple port are in no sense global. Each safely tracks the interprocess state of at most two processes. Consequently we must limit their scope as follows. For port variable *a*, process P *owns a* if either of the following hold:

a. P names port A in an I/O command;

b. P names port B in an I/O command and S_B mentions *a*.

For auxiliary variable *v*, process P *owns v* if P names port A in an I/O command and S_a mentions *v*.

We require that the following condition hold: *each port and auxiliary variable may be owned by no more than two processes* (for logical soundness of the proof system).

4. Port A has associated with it an assertion $\beta_A(s)$ that captures the view of the external environment as seen by the input process and guaranteed by the output process. $\beta_A(s)$ is confined to port space and therefore mentions as free variables only auxiliary variables and port variables of the port set containing A. In addition, the two processes using port A must own all variables of $\beta_A(s)$.

5. Port A has associated with it an assertion α_A that captures the view of the external environment as seen by the output process and guaranteed by the input process. α_A is constrained to mention the same kind of variables as found in $\beta_A(s)$, and the processes using A must own all variables of α_A.

In summary, then, the port set data base, or interface data base, consists of the associated port variables and auxiliary variables. The port actions of ports belonging to the port set constitute the operations that update the data base whenever a transaction occurs across the interface. The α's and β's provide the logical link between the state of the interface and the state of the local processes

5.3. Port Semantics

Port semantics specify precisely how α and β relate to the local process environment in terms of the Hoare axiomatic proof system. The aim here is to maintain modularity; a change to one process should require only reverification of that one process against an established interface, i.e., against a predefined port set. Individual process proofs interact only via the common interface and are buffered in a logical sense by it.

Local process variables and variables of the port set data base are the only variables allowed to appear free in proof assertions of a local process. The following axioms define the proof methodology for partial correctness.

1. Hoare's axiom's, composition rules, and rules of inference for sequential programs, both deterministic and nondeterministic, as defined by Dijkstra [6] in terms of wp and wlp (weakest precondition and weakest liberal precondition, respectively) and as described by deBakker [3] in terms of sp and slp (strongest postcondition and strongest liberal postcondition, respectively).

2. Axiom for Parallel Composition

$$\frac{\{L_i\}\,P_i\,\{M_i\} \quad \text{for } i = 1, \ldots, n}{\{\prod_{i=1}^{n} L_i\}\,[P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n]\,\{\prod_{i=1}^{n} M_i\}}$$

where each port and auxiliary variable appears free in the pre- and/or postcondition of only the two P_i that own it, and L_i and M_i mention no local process variables changed by P_j, $j \neq i$. This axiom allows the building up of a proof from proofs of individual processes.

3. Axioms of Communication

S_A is the port action declared for port A.

a. $wlp(A?x,M(x)) = \forall a\,[(\beta_A(a) \rightarrow wp(S_A(a),M(a))) \wedge \alpha_A]$

b. $wlp(A!y,R(a)) = \forall a\,[(\alpha_A \rightarrow wp(S_A(y),R(y))) \wedge \beta_A(y)]$

These two axioms form the heart of the methodology in that they provide for the establishment of triples of the form $\{L\}\,A?x\,\{M\}$ and $\{N\}\,A!y\,\{R\}$ wholly in terms of the local environment and the specified interface. They are a formal expression of the essence of our notion of modularity. The universal quantifiers are necessary for soundness and play the same role in all axioms in which they appear.

Figure 5-3 illustrates the relationships expressed by these two axioms. The prime notation indicates an assertion subsequent to execution of the port action. Notice how closely it suggests the two-port of electrical network theory.

4. Axiom of Substitution

$$\frac{\{L(j)\}\,s\,\{M\}\text{, } j \text{ is a free auxiliary variable, } z \text{ not in } s \text{ and not free in } M}{\{L(z)\}\,s\,\{M\}}$$

This axiom permits the elimination of auxiliary variables from preconditions. z is usually chosen to be a constant, most often the constant 0. It may be applied *no more than once per auxiliary variable* (for soundness).

In the next two axioms b_i is a boolean expression and G_i an I/O command, either of which may be nonexistent.

5. Axiom for Alternative Command

$$\frac{\prod_{i=1}^{n} [L \rightarrow [\sum_{k=1}^{n} b_k \wedge (b_j \rightarrow wlp(G_j;S_j,M))]]}{\{L\}\,[b_1;G_1 \rightarrow S_1 \,\square\, \ldots \,\square\, b_n;G_n \rightarrow S_n]\,\{M\}}$$

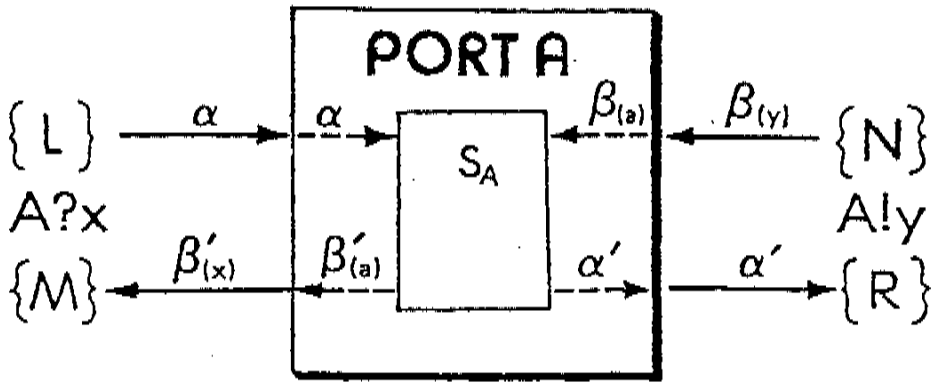

Figure 5-3: LOGICAL RELATIONSHIP BETWEEN LOCAL AND INTERFACE ENVIRONMENTS

6. Axiom for Repetitive Command

$$\frac{\prod_{i=1}^{n} [(L \wedge b_i) \Longrightarrow wlp(G_i ; S_i, L)]}{\{L\} * [b_1 ; G_1 \longrightarrow S_1 \,\square\, \ldots \,\square\, b_n ; G_n \longrightarrow S_n] \{L \wedge \prod_{i=1}^{n} (b_i \Longrightarrow G_i \text{ names DYNAMIC port})\}}$$

L is known as the loop invariant.

5.4. Remarks Concerning Axioms of Communication

The Axioms of Communication are used when "pulling" a postcondition back through a process. statement by statement, to derive the process precondition. This can be done once the port set has been fully defined. • But we can take another view of the situation as well. Suppose that at the outset we leave α and β unspecified and try to find the weakest set of relations that they must satisfy when the process proof assertions have already been specified.

Assume that we are given $\{L(a)\}\,A?x\,\{M(x)\}$ and $\{N(a)\}\,A!y\,\{R(a)\}$. Let $\vec{u} = (x,u_1,u_2, \ldots)$ indicate those variables in L and M that are neither port variables nor auxiliary variables, and similarly $\vec{v} = (y,v_1,v_2, \ldots)$ for N and R. Denote by $wl\beta$ and $wl\alpha$ the weakest liberal α and β, respectively.

Then

$$wl\beta_A(a)\ [A?x,L(a),M(x),N] = \begin{cases} D(a) \text{ if } N \Rightarrow D(y) \\ \mathit{false} \text{ otherwise} \end{cases}$$

where D(a) is given by

$\forall\vec{u}\forall w\,[L(w) \Rightarrow wp(S_A(a),M(a))]$, $w \neq a$, w not free in L ;

$$wl\alpha_A\ [A!y,N(a),R(a),L] = \begin{cases} E \text{ if } L \Rightarrow E \\ \mathit{false} \text{ otherwise} \end{cases}$$

where E is given by

$\forall\vec{v}\forall a\,[N(a) \Rightarrow wp(S_A(y),R(y))]$.

In the case that process P contains n input commands naming A.

$$\{L(a)_i\}\,A?x_i\,\{M(x_i)\} \qquad i = 1, \ldots, n \ ,$$

and process Q contains m output commands naming A,

$$\{N(a)_j\}\,A!y_j\,\{R(a)_j\} \qquad j = 1, \ldots, m \ ,$$

defining the weakest β and α is only a little more complex. Define $D(a)_i$ and E_j for each i and j as above. Let $D(a) = \prod_{i=1}^{n} D(a)_i$ and $E = \prod_{j=1}^{m} E_j$. Then

$$wl\beta_A = \begin{cases} D(a) \text{ if } \prod_{j=1}^{m} (N_j \Rightarrow D(y)_j) \\ \mathit{false} \text{ otherwise} \end{cases}$$

and

$$wl\alpha_A = \begin{cases} E \text{ if } \prod_{i=1}^{n} (L_i \Rightarrow E) \\ \mathit{false} \text{ otherwise} \end{cases} .$$

Working backwards as shown here is sometimes a useful technique for developing a proof when it is more clearly understood what transpires at the local level than at the interface level. Here the interface semantics are derived, in a sense, from the composite local environments.

We can take yet a third view of the situation and ask: what is the strongest postcondition given a specific precondition and given that the interface has been fully specified? The case of input proceeds as follows. If $L \not\rightarrow \alpha_A$, then

$$slp(L(a,x),A?x) = \mathit{false} \quad ;$$

otherwise from (2) of section 5.2 we get

$$slp(L(a,x),A?x) =$$
$$sp(\exists z_1\,[L(z_1,x)\wedge\beta_A(a)],\ S_A;x:=a),\ z_1\neq a,\ z_1 \text{ not free in } (L\wedge\beta_A(a))$$
$$= sp(\exists z_1\,[L(z_1,x)\wedge\beta_A(a)],\ x:=a;S_A)$$
$$= sp(\exists z_1 \exists z_2\,[L(z_1,z_2)\wedge\beta_A(a)\wedge(x=a)],\ S_A) \quad , \qquad (3)$$
$$z_1\neq a,\ z_2\neq x,\ z_1\neq z_2,\ z_1 \text{ and } z_2 \text{ not free in } (L\wedge\beta_A(a)) \quad .$$

When x and a are not free in L, which is normally the case, (3) reduces to the more intuitive expression

$$sp(L\wedge\beta_A(a)\wedge(x=a),\ S_A) \quad .$$

The output postcondition proceeds similarly. If $N \not\rightarrow \beta_A(y)$, then $slp(N(a),A!y) = \mathit{false}$; otherwise

$$slp(N(a),A!y) = sp((N(a)\wedge\alpha_A),\ a:=y;S_A)$$
$$= sp(\exists z\,[N(z)\wedge\alpha_A\wedge(a=y)],\ S_A),\ z\neq a,\ z \text{ not free in } (N\wedge\alpha_A) \quad . \qquad (4)$$

When a is not free in N, the usual case, (4) reduces to

$$sp(N\wedge\alpha_A\wedge(a=y),\ S_A) \quad .$$

5.5. Examples

1. Simple Example

Consider the program [P||Q] where P and Q are

```
P:: A!x;                    Q:: A?y;
    B?z                         B!y
```

We wish to verify $\{true\}P\{z=x\}$ against the port set {A,B} which have been declared

	A	B
Type	static	static
Action	*skip*	*skip*
α	*true*	*true*
β	*true*	b=a

The annotated version of P is

```
P::  {true}
     A!x
     {a=x};
     B?z
     {z=x}
```

The Axioms of Communication justify each assertion.

We still need to establish $\{true\}\,Q\,\{true\}$ in order to conclude $\{true\}\,[P||Q]\,\{z=x\}$ by the Axiom of Parallel Composition. The annotated version of Q is

```
Q::  {true}
     A?y
     {a=y};
     B!y
     {true}
```

Once again each assertion is justified by the Axioms of Communication.

2. Dijkstra's Exponentiation Problem

The following is a distributed version of Dijkstra's exponentiation problem [6] which computes $z = v^u$ for integers $u \geq 0$, $v > 1$, in an efficient manner (assuming that exponentiation is not a primitive operation). The program is given by [P||Q], where P and Q are

```
P:: z:=1;
    *[u≠0 → A!u; f:=true;
              *[f;B?yes() → v:=v*v □ f;C?u → f:=false];
    z:=v*z; u:=u-1]


Q:: *[A?x → *[even(x) → B!yes(); x:=x/2]; C!x]
```

The parallelism permits concurrent squaring of v with halving of x. We shall prove the validity of $\{v^u=k\}\,[P||Q]\,\{z=k \wedge u=0\}$. The ports are declared as shown; i is an auxiliary variable of type integer.

	A	B	C
Type	dynamic	static	static
Action	i := 0	i := i+1	*skip*
α	*true*	*true*	*true*
β	*true*	*true*	$2^{i}{}_{c=a}$

The annotated text for the proof of $\{v^u=k\}\,P\,\{z=k \wedge u=0\}$ appears as shown.

$P::\quad \{v^u=k\}$

$z := 1$

$\{zv^u=k\};$

$*[\{zv^u=k\}\ u\neq 0 \longrightarrow \{zv^u=k\}\ A!u\ \{zv^u=k \wedge a=u \wedge i=0\};\ f := true\ \{zv^u=k \wedge a=u \wedge i=0 \wedge f\};$

$*[\{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a)\}\ f\ \{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^{i+1} u=a)\};\ B?yes() \longrightarrow$

$\{zv^{a2^{1-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a)\}\ v := v*v\ \{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a)\}$

$\Box\ \{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a)\}\ f\ \{zv^{a2^{-i}}=k\};\ C?u \longrightarrow$

$\{zv^{a2^{-i}}=k \wedge 2^i u=a\}\ f := false\ \{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a)\}]$

$\{zv^{a2^{-i}}=k \wedge (\neg f \Longrightarrow 2^i u=a) \wedge \neg f\};$

$z := v*z\ \{zv^{u-1}=k\};\ u := u-1\ \{zv^u=k\}]$

$\{zv^u=k \wedge u=0\}$

The annotated proof of $\{true\}\,Q\,\{true\}$ is as shown.

$Q::\ \{true\}$

$*[\{true\}\ A?x \longrightarrow \{i=0 \wedge a=x\}\ *[\{2^i x=a\}\ even(x) \longrightarrow \{2^i x=a\}\ B!yes()\ \{2^{i-1} x=a\};$

$x := x/2\ \{2^i x=a\}]\ \{2^i x=a\};$

$C!x\ \{true\}]$

$\{true\}$

An application of the Axiom for Parallel Composition yields the desired result.

Notice that the proof is largely insensitive to changes in the modules. A different version of Q, for example Q1 shown below, would require only verification of $\{true\}\,Q1\,\{true\}$ against the port set to establish the validity of $\{v^u=k\}\ [P||Q1]\ \{z=k \wedge u=0\}$. P would not require revalidation.

$Q1::\ *[A?x \longrightarrow n := 0;$

$*[even(x/2^n) \longrightarrow n := n+1];\ x := x/2^n;$

$*[n>0 \longrightarrow B!yes();\ n := n-1];\ C!x]$

See [12] for larger examples.

6. MULTI-PORTS

6.1. General Description

A multi-port is defined to be a port that is used by more than one process for input, or more than one process for output. Hence any port that is used by three or more process is necessarily a multi-port. The converse is not true, however, since we allow a process to use a port for both input and output (though none of the processes in any of the examples shown here will do so, since it reflects awkward style). Transactions across a multi-port always proceed serially, one at a time. The pairing of commands for synchronization is done nondeterministically, but fairly; no process will be passed over in favor of others indefinitely. Moreover each process of a synchronized pair does not know the identity of the other (for unified semantics). As with simple ports, multi-ports are either of type static or dynamic.

Consider, for example, a producer process P and n identical consumer proccesses $C_1, \ldots, C_n$ operating in parallel. The full system is described by the parallel command $[P||C_1|| \ldots ||C_n]$, where P is

```
P:: *[produce(x) → A!x]
```

each C_i is given by

```
Ci:: *[A?x → consume(x)]
```

and A is the multi-port through which communication and synchronization proceeds. Multi-ports used in a many-to-one configuration as in this example resemble the port as described and implemented by Silberschatz [16]; the semantics defined below are not, however, limited in the most general case to the many-to-one configuration.

6.2. Formal Description

A general (i.e. many-to-many) multi-port is defined as follows.

1. Multi-port variables and those auxiliary variables that are referenced by multi-port A are of the vector-type

$$\vec{i} = (i_{P_{1_{in}}}, i_{P_{1_{out}}}, i_{P_{2_{in}}}, i_{P_{2_{out}}}, \ldots, i_{P_{n_{in}}}, i_{P_{n_{out}}}) ,$$

where component $i_{P_{j_{in}}}$ is associated with process P_j that uses A for input, and component $i_{P_{k_{out}}}$ is associated with process P_k that uses A for output. A process P_j that uses A for both input and output is represented by two components: $i_{P_{j_{in}}}$ and $i_{P_{j_{out}}}$. For the sake of brevity we will write $i_{j_{in}}$ when referring to $i_{P_{j_{in}}}$. When the context allows, we will drop the *in* (or *out*) and write simply i_j.

2. Associated with multi-port A is a *multi-port variable* $\vec{a}$. Synchronization of A?x in process P_j and A!y in process P_k is interpreted as atomic execution of the sequence

$$a_{k_{out}} := y; a_{j_{in}} := a_{k_{out}}; x := a_{j_{in}} \quad . \tag{5}$$

3. Associated with multi-port A is a distinguished auxiliary variable $\vec{t}_A$ that maintains the *transaction count* through A. $\vec{t}_A$ may not be changed by any port action in the port set.

4. Associated with multi-port A is a *multi-port action* S_A that consists of assignment statements to vector type auxiliary variables. S_A is defined in a generic manner: free variables referenced are vector variables and not vector components (for unified semantics; a typical S_A might be $\vec{v} := \vec{v}+1$). Synchronization of A?x in P_j and A!y in P_k is reinterpreted from (5) to now mean atomic execution of the sequence

$$a_{k_{out}} := y; t_{A_{k_{out}}} := t_{A_{k_{out}}} + 1; S_{A_{k_{out}}}; a_{j_{in}} := a_{k_{out}}; t_{A_{j_{in}}} := t_{A_{j_{in}}} + 1; S_{A_{j_{in}}}; x := a_{j_{in}} \quad ,$$

where the subscript k_{out} in $S_{A_{k_{out}}}$ indicates that all references to variables refer to component k_{out}; similarly for j_{in} in $S_{A_{j_{in}}}$.

5. *Kirchhoff's Law* is an invariant relation on $\vec{t}_A$ and $\vec{a}$:

   a. Initially, $t_{A_{j_{in}}} = t_{A_{j_{out}}} = 0$ for all j. This is used when applying the Axiom of Substitution in that *only the constant* $\vec{0}$ may be substituted for $\vec{t}_A$ in a process precondition.

b. $[(\vec{t}_A \geq \vec{0}) \wedge (\sum_j t_{A_{j_{in}}} = \sum_j t_{A_{j_{out}}}) \wedge (\vec{t}_A \neq \vec{0}) \Rightarrow \sum_{(i \neq j)} [(a_{i_{in}} = a_{j_{out}}) \wedge (t_{A_{i_{in}}} > 0) \wedge (t_{A_{j_{out}}} > 0)])]$,

abbreviated KL_A, is such that $\{true\} S \{KL_A\}$ is valid for any program statement S in the scope of A.

Intuitively, Kirchhoff's Law states that every transaction involves both a distinct input process and a distinct output process, and once a transaction through A has occurred, there must exist two distinct processes, one input and one output, that were involved in the most recent transaction.

6. Associated with A are assertions $\beta_A(\vec{s})$ and α_A. As in the case of S_A, they are defined in a generic manner: references to vector components do not appear (this restriction is relaxed somewhat in the less general case of many-to-one multi-ports). A typical $\beta_A(\vec{s})$ might be $\vec{s}=\vec{b}$.

6.3. General Multi-Port Semantics

Multi-port semantics are expressed as a generalized form of simple-port semantics described earlier. The axioms that require revision appear as follows.

2′. General Multi-Port Axiom for Parallel Composition

$$\frac{\{L_i\}\, P_i\, \{M_i\} \quad \text{for } i = 1, \ldots, n}{\{\prod_{i=1}^{n} L_i\}\ \text{declare multi-port}_1, \ldots, \text{multi-port}_m : [P_1 \,||\, P_2 \,||\, \ldots \,||\, P_n]\ \{\prod_{i=1}^{n} M_i \wedge \prod_{i=1}^{m} KL_{\text{multi-port}_i}\}}$$

where each port or auxiliary variable of scalar type appears free in the pre- and/or postcondition of only the two processes that own it, and each port or auxiliary vector variable component v_i appears free only in the pre- and/or postcondition of P_i. (Note that there may be simple ports as well declared in front of the parallel command above; we have just not shown them.)

3′. Axioms of Communication for General Multi-Ports

Let A be a multi-port, let U be the set of multi-ports contained in the port set containing A, and let V be the set of vector type interface variables associated with U. $S_A(\vec{s})$ is the multi-port action defined for A. For A?x in process P_j and A!y in process P_k

a. $wlp(A?x,M(x)) =$

$$(\prod_i \forall a_{j_{in}}\ \forall v_r \forall r \neq j_{in}\ \forall \vec{v} \in V\ [(\beta_A(a_{j_{in}})_{i_{out}} \wedge [\!]KL_p)_{p \in U} \Longrightarrow wp(t_{A_{j_{in}}} := t_{A_{j_{in}}} + 1; S_A(\vec{a})_{j_{in}}, M(a_{j_{in}}))])\wedge(\alpha_A)_{j_{in}}$$

b. $wlp(A!y,R(a_{k_{out}})) =$

$$(\prod_j \forall a_{k_{out}}\ \forall v_r \forall r \neq k_{out}\ \forall \vec{v} \in V\ [((\alpha_A)_{j_{in}} \wedge [\!]KL_p)_{p \in U} \Longrightarrow wp(t_{A_{k_{out}}} := t_{A_{k_{out}}} + 1; S_A(y)_{k_{out}}, R(y))])\wedge\beta_A(y)_{k_{out}}$$

where "j_{in}" indicates that all references to vector variables in generic form (e.g. $\vec{r}_A$) refer to components corresponding to $P_{j_{in}}$; similarly for "k_{out}."

It is worth noting that these axioms collapse to the simple case presented in section 5.3 when A is simple (non-multi), i.e., when the vector variables of V contain only two components. In that case KL_A asserts, essentially, that the two components track each other, acting together as a single variable.

6.4. General Multi-Port Examples

1. Simple Example

Consider the three process P, Q, and R operating in parallel and communicating via the *dynamic* multi-port A (x is an integer variable):

```
R:: *[x>0 → A!count(); x:=x-1]

P:: m:=0; *[A?count() → m:=m+1]

Q:: n:=0; *[A?count() → n:=n+1]
```

P, Q, and R fit the producer-consumers paradigm mentioned above. x number of signals are transmitted from R to P and Q. Upon termination x number of signals must have been received by P and Q, but the distribution of how many were received by each is left undetermined. We will show that $\{x\geq 0 \wedge x=c\}\ [P||Q||R]\ \{x=0 \wedge m+n=c\}$ is valid. Multi-port A is defined such that $\alpha_A \equiv \beta_A \equiv true$ and $S_A \equiv skip$. The annotated text appears as shown.

$R::\ \{x\geq 0 \wedge x=c \wedge t_{A_{R_{out}}}=0\}$

$*[\{IL_R\}\ x>0 \rightarrow$

$\{(c=t_{A_{R_{out}}}+x) \wedge (x>0)\}\ A!count()\ \{(c=t_{A_{R_{out}}}+x-1) \wedge (x>0)\};\ x:=x-1\ \{IL_R\}]$

$\{IL_R \wedge \neg(x>0)\}$,

where the loop invariant IL_R is given by $(c=t_{A_{R_{out}}}) \wedge (x \geq 0)$.

$$P::\ \{t_{A_{P_{in}}}=0\}$$

$$m:=0$$

$$\{(m=0) \wedge (t_{A_{P_{in}}}=0)\};$$

$$*[\{IL_P\}A?count() \longrightarrow \{m+1=t_{A_{P_{in}}}\}m:=m+1\{IL_P\}]\ \{IL_P\}\ ,$$

where the loop invariant is given by $(m=t_{A_{P_{in}}})$.

The proof of $\{t_{A_{Q_{in}}}=0\}Q\{n=t_{A_{Q_{in}}}\}$ is similar and will be omitted. The final result now follows from the Axioms for Parallel Composition (General Multi-Port) and Substitution. Notice that if there was an additional process in the system outputting through A, the result would not follow, for then KL_A would not imply

$$t_{A_{R_{out}}} = t_{A_{P_{in}}} + t_{A_{Q_{in}}} \quad . \tag{6}$$

Processes may be added without need for reverification as long as they preserve (6).

2. Larger Example ("Who received the last item?")

Consider P, Q, and R operating in parallel and sharing *dynamic* port A:

$$R::\ *[even(x) \longrightarrow x:=x/2;\ A!x]$$

$$P::\ *[A?z \longrightarrow skip]$$

$$Q::\ *[A?y \longrightarrow skip]$$

We will show that $\{even(x)\}\ [P||Q||R]\ \{odd(x) \wedge (odd(y) \vee odd(z))\}$ is valid. Port A is defined as in Example 1. The annotated text appears below.

$$R::\ \{even(x) \wedge t_{A_{R_{out}}}=0\}$$

$$*[\{IL_R\}even(x) \longrightarrow \{true\}x:=x/2\{true\};\ A!x\{IL_R\}]$$

$$\{IL_R \wedge odd(x)\}\ .$$

where the loop invariant IL_R is $(odd(x) \rightarrow t_{A_{R_{out}}} > 0) \wedge (t_{A_{R_{out}}} > 0 \rightarrow x = a_{R_{out}})$.

$P:: \{t_{A_{P_{in}}} = 0\}$

$*[\{IL_P\}\, A?z \rightarrow skip\{IL_P\}]$

$\{IL_P\}$,

where the loop invariant IL_P is $(t_{A_{P_{in}}} > 0 \rightarrow z = a_{P_{in}})$.

The proof of $\{t_{A_{Q_{in}}} = 0\}\, Q\, \{t_{A_{Q_{in}}} > 0 \rightarrow y = a_{Q_{in}}\}$ is similar and will be omitted.

Now $(IL_R \wedge odd(x) \wedge KL_A) \rightarrow [odd(x) \wedge (x = a_{R_{out}}) \wedge (t_{A_{P_{in}}} > 0 \vee t_{A_{Q_{in}}} > 0) \wedge (a_{R_{out}} = a_{P_{in}} \vee a_{R_{out}} = a_{Q_{in}}) \wedge$

$(t_{A_{P_{in}}} = 0 \rightarrow a_{R_{out}} = a_{Q_{in}}) \wedge (t_{A_{Q_{in}}} = 0 \rightarrow a_{R_{out}} = a_{P_{in}})]$.

and thus the result follows from the Axioms for Parallel Composition (General Multi-Port) and Substitution. The addition of other processes that input through port A invalidates the proof since the implication above would not hold; KL_A lacks sufficient strength in that case.

6.5. Many-To-One Ports

Most applications of multi-ports assume a many-to-one configuration. The semantics of the many-to-one port are much stronger than for the general case since the identity of one of the communicating partners is always known. We can relax some of the restrictions on the sharing of interface variables and still maintain the requirement, for soundness, that no such variable spans more than a pair of processes. We expand the general semantics developed thus far and conclude with two examples in the next section.

Let $P = \{P_1, P_2, \ldots, P_n\}$ be a collection of lowest level processes and let U be the set of ports used for communication only between process Q (also lowest level) and some subset of P in a many-to-one configuration as shown in figure 6-1. Since we assume, further, that P is maximal and that the P_i's do not use ports in U to communicate among themselves, we deduce

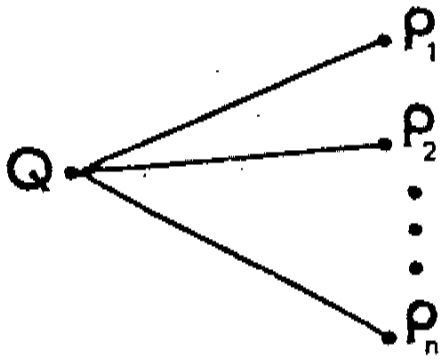

Figure 6-1: MANY-TO-ONE PORT COMMUNICATION

that ports in U are not used by Q and elements of P for both input and output within a single process. For uniformity of treatment we shall suppose that all ports of U are defined as multi-ports, even if used by only one P_i.

Let V be the set of interface variables referenced by ports in U. Then we require that the following conditions be met:

1. Only processes in P can reference variables of V.

2. As in the case of general multi-ports, elements of V are vector type and port actions are generically defined (i.e., no references to components).

3. α's and β's may mention as free variables any variable of V, subject to the following constraints:

   For $A \in U$,

   a. If A is used by Q for input, then free variables mentioned by α_A must be vector components and $\beta_A(\vec{a})$ is generically defined;

   b. If A is used by Q for output, then free variables mentioned by $\beta_A(\vec{a})$ must be vector components and α_A is generically defined.

4. Process proof assertion references to (free) variables of V must be to components, subject to the following:

a. Q can mention any component.

b. P_i can mention only components subscripted i.

(Hence components subscripted i are *owned* by processes Q and P_i).

The revised axioms appear as follows.

2''. Many-To-One Port Axiom for Parallel Composition

$$\frac{\{L_i\}\ P_i\ \{M_i\} \quad \text{for } i = 1, \ldots, n}{\{\prod_{i=1}^{n} L_i\}\ \text{declare many-to-one port}_1, \ldots, \text{many-to-one port}_m; [P_1 \,||\, \ldots \,||\, P_n]\ \{\prod_{i=1}^{n} M_i \wedge \prod_{i=1}^{m} KL_{\text{many-to-one port}_i}\}}$$

where condition 4 above holds.

3''. Axioms of Communication for Many-To-One Ports

For A?x in process Q, A!y in process P_i

a. $wlp(A?x,M(x)) =$

$$[\prod_{i=1}^{n} \forall a_i ((\beta_A(\vec{a})_i \wedge \prod_{P \in U} KL_P) \Rightarrow wp(t_{A_i} := t_{A_i}+1; S_A(\vec{a})_i; a_Q := a_i; t_{A_Q} := t_{A_Q}+1; S_A(\vec{a})_Q, M(a_Q)))] \wedge \alpha_A$$

b. $wlp(A!y,R(a_i)) =$

$$(\forall a_i \forall v_j \forall j \neq i \forall \vec{v} \in V\ [(\alpha_A \wedge \prod_{P \in U} KL_P) \Rightarrow wp(t_{A_i} := t_{A_i}+1; S_A(y)_i, R(y))]) \wedge \beta_A(y)_i$$

For A!y in process Q, A?x in process P_i

c. $wlp(A!y,R(a_Q)) =$

$$[\prod_{i=1}^{n} \forall a_i ((\alpha_{A_i} \wedge \prod_{P \in U} KL_P) \Rightarrow wp(a_i := y; t_{A_i} := t_{A_i}+1; S_A(y)_i; t_{A_Q} := t_{A_Q}+1; S_A(y)_Q, R(y)))] \wedge \beta_A(y)$$

d. $wlp(A?x,M(x)) =$

$$(\forall a_i \forall v_j \forall j \neq i \forall \vec{v} \in V\ [(\beta_A(a_i) \wedge \prod_{P \in U} KL_P) \Rightarrow wp(t_{A_i} := t_{A_i}+1; S_A(\vec{a})_i, M(a_i))]) \wedge (\alpha_A)_i$$

where subscript i indicates vector components corresponding to process P_i; subscript Q indicates vector components corresponding to process Q.

6.6. Many-To-One Port Examples

1. Simple Example

Consider the system [P||Q(i:1..n)], where P and Q(i) are given by

```
P:: m:=100; *[m>0 ⟶ A!x; B?y; m:=m-1]

Q(i):: *[A?w ⟶ B!w]
```

(each Q(i) has its own local variable w)
We will prove the validity of $\{true\}\ [P||Q(i:1..n)]\ \{x=y\}$. A and B are defined as shown:

	A	B
Type	dynamic	static
Action	*skip*	*skip*
α	*true*	*true*
β	*true*	$\vec{t}_A - \vec{t}_B + 1 > 0$

Subscript j denotes vector component Q(j). The proof of $\{\prod_{j=1}^{n} t_{A_j} = t_{B_j}\}\ P\ \{\sum_{j=1}^{n} (x=a_j \wedge y=b_j \wedge t_{A_j} > 0)\}$ is shown in the annotated text below.

$$P::\ \{\prod_{j=1}^{n} t_{A_j} = t_{B_j}\}$$

$$m:=100$$

$$\{\prod_{j=1}^{n} t_{A_j} = t_{B_j} \wedge m=100\};$$

$$*[\{IL_P\}\, m>0 \longrightarrow \{\prod_{j=1}^{n} t_{A_j} = t_{B_j}\}\, A!x\, \{\sum_{j=1}^{n} [t_{A_j} = t_{B_j} + 1 \wedge x=a_j \wedge \prod_{k=1}^{n} (k \neq j \Longrightarrow t_{A_k} = t_{B_k})]\}];$$

$$B?y\, \{\prod_{j=1}^{n} (t_{A_j} = t_{B_j}) \wedge (\sum_{j=1}^{n} (x=a_j \wedge y=b_j \wedge t_{A_j} > 0))\};\ m:=m-1\ \{IL_P\}]$$

$\{IL_P \wedge m \leq 0\}$,

where the loop invariant IL_P is $m \geq 0 \wedge \prod_{j=1}^{n}(t_{A_j} = t_{B_j}) \wedge (m=0 \rightarrow \sum_{j=1}^{n}(x=a_j \wedge y=b_j \wedge t_{A_j} > 0))$.

$(IL_P \wedge m \leq 0)$ implies the desired postcondition.

The proof of $\{t_{A_i} = t_{B_i} = 0\}\, Q(i)\, \{t_{A_i} > 0 \rightarrow a_i = b_i\}$ follows.

$Q(i)::\ \{t_{A_i} = t_{B_i} = 0\}$

$*[\{IL_Q\}\, A?w \rightarrow \{t_{A_i} = t_{B_i} + 1 > 0 \wedge w = a_i\}\, B!w\, \{IL_Q\}]$

$\{IL_Q\}$,

where IL_Q is $(t_{A_i} = t_{B_i} \geq 0) \wedge (t_{A_i} > 0 \rightarrow a_i = b_i)$.

The result follows from the Axioms for Parallel Composition (Many-To-One Port) and Substitution.

2. The Subroutine Process

We prove the single entry subroutine process, which is similar to that described in [8]. A subroutine process is patterned as

```
SUB:: initialize; *[A?(input parameters) → ... ;B!y(output parameters)] ,
```

where "..." computes the output parameters from the input parameters. The user calls SUB with the pair of commands

```
A!(arguments); ... ;B?(results)
```

where "..." is executed concurrently with the subroutine. There are several calling processes but only one copy of SUB; hence A and B are many-to-one ports. A is dynamic and B is static since SUB should not terminate until all its users have terminated.

As a representative subroutine let SUB be the program that computes the sine function for USER(i), i = 1, ... ,n:

```
SINE: *[A?θ → s := sin(θ); B!s]
```

The proof verifies that if all users adhere to the calling sequence protocol, then y = sin x holds immediately following B?y, modulo termination of course. The annotated texts follow.

	A	B
Type	dynamic	static
Action	*skip*	*skip*
α	*true*	$\vec{t}_A = \vec{t}_B + 1$
β	*true*	$\sum_{j=1}^{n} [t_{A_j} = t_{B_j} + 1 \wedge \vec{b} = \sin a_j \wedge \prod_{k=1}^{n} (k \neq j \Longrightarrow t_{A_k} = t_{B_k})]$

SINE:: $\{IL_s\}^*[\{IL_s\}A?\theta \longrightarrow \{\sum_{j=1}^{n} [t_{A_j} = t_{B_j} + 1 \wedge a_j = \theta \wedge$

$\prod_{k=1}^{n} [(k \neq j \Longrightarrow t_{A_k} = t_{B_k}) \wedge (t_{A_k} > 0 \Longrightarrow b_k = \sin a_k)]]\}; s := \sin(\theta)$

$\{\sum_{j=1}^{n} [t_{A_j} = t_{B_j} + 1 \wedge s = \sin a_j \wedge \prod_{k=1}^{n} [(k \neq j \Longrightarrow t_{A_k} = t_{B_k}) \wedge (t_{A_k} > 0 \Longrightarrow b_k = \sin a_k)]]\};$

$B!s\{IL_s\}]$

$\{IL_s\}$,

where IL_s is $\prod_{j=1}^{n} [(t_{A_j} = t_{B_j}) \wedge (t_{A_j} > 0 \Longrightarrow b_j = \sin a_j)]$.

USER(i)::

$\{t_{A_i} = t_{B_i} = 0\}$

.
.
.

$\{t_{A_i} = t_{B_i}\}$

$A!x$

$\{x = a_i \wedge t_{A_i} = t_{B_i} + 1\}$

.
.
.

$\{x = a_i \wedge t_{A_i} = t_{B_i} + 1\}$

B?y

$\{y=\sin x \wedge t_{A_i} = t_{B_i}\}$

.
.
.

$\{t_{A_i} = t_{B_i}\}$

If user u violates the calling protocol, then $(t_{A_u} = t_{B_u})$ will not hold as a process postcondition, and the Axioms for Parallel Composition and Substitution will yield $\{true\}$ [SINE||USER(i:1..n)] $\{false\}$. If all users adhere to the protocol, then $\{true\}$ [SINE||USER(i:1..n)] $\{true\}$ holds.

7. ANOTHER APPROACH

We mention a slightly varied approach that some may find preferable, but is, to an extent, more restrictive. In practice, however, these restrictions are generally inconsequential since all reasonable programs follow the pattern anyhow.

First, we limit the usage of both simple and multi-ports in a natural way by disallowing a lowest level process from using a particular port for *both input and output*. Second, associate simple ports with processes, again in a natural way, as follows. For each pair of communicating, lowest level processes, associate an "interface" consisting of a collection of ports to be used by no other processes. For each such interface, associate a set of interface variables, which are referenced only by that particular interface and its two associated processes. What we have, then, is that given a process P, the complete interface to its external environment is given by a collection of sets of ports, each set of the collection being its interface to one other process. This is illustrated in figure 7-1.

The benefits gained by this approach are the following:

1. While logically equivalent to the original scheme, this is, perhaps, a better organized and more structured arrangement;

2. Simplification of the closure dynamics of *dynamic* ports;

3. Elimination of subscripts *in/out* when dealing with multi-ports;

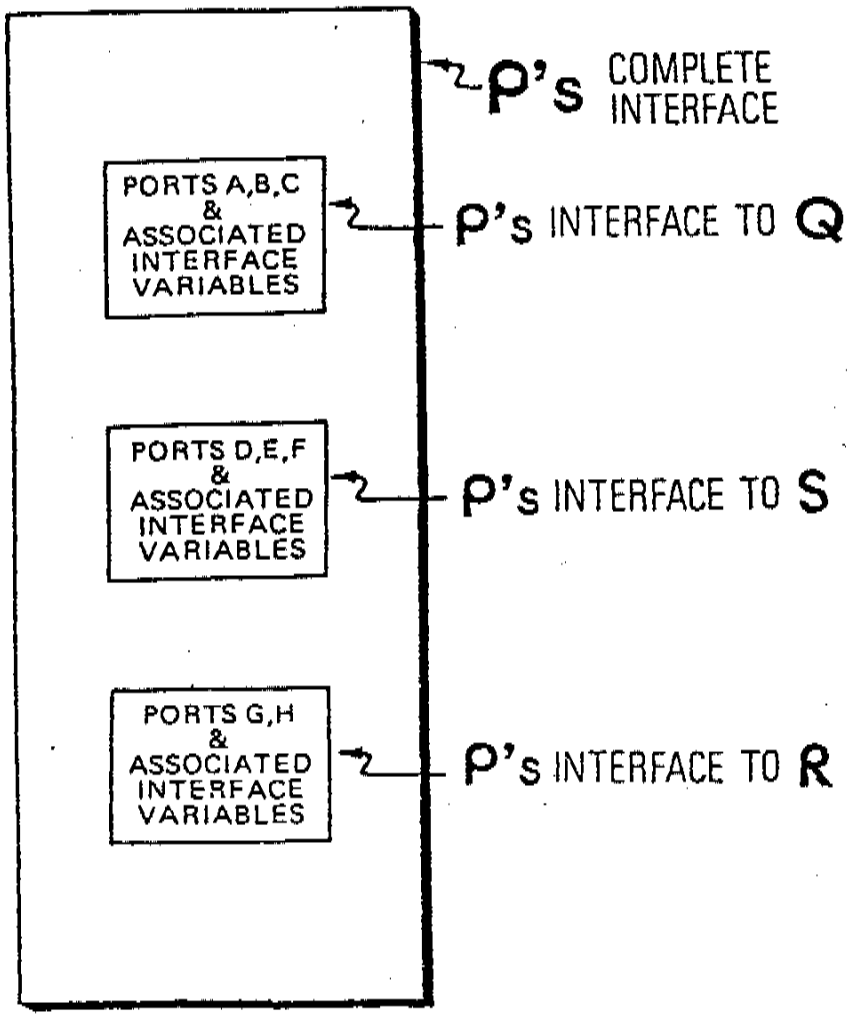

Figure 7-1: ASSOCIATING PORTS WITH PAIRS OF PROCESSES

4. Elimination of the notion of *ownership* of interface variables since interface variables are now automatically owned by only two processes, with a resulting simplification of the Axioms for Parallel Composition;

5. The generalization from simple-port semantics to multi-port semantics is more natural.

8. CONCLUSION

We have shown that it is possible to do for parallel asynchronous systems what has been done for analog systems in general: formally specify components and adequate interfaces to provide plug-compatible modularity. This was accomplished through the notion of *port* with carefully chosen formal semantics that mesh neatly with sequential Hoare axiomatics. A proof can be constructed along the same lines as the system itself and maintained along with the system in an isomorphic manner. Although this particular methodology functions in the CSP setting, the techniques shown here can be applied as well to shared-data environments (e.g. monitors) in which synchronization is not handled by communication primitives. Although we have not discussed the question of soundness and completeness, these issues should be addressed.

9. ACKNOWLEDGEMENT

We wish to thank Tony Hoare for an encouraging review of this work.

References

1. Apt, K., Francez, N. and de Roever, W. P. "A Proof System for Communicating Sequential Processes". *ACM Trans. on Programming Lang. and Systems 2*, 3 (July 1980), 359–385.

2. Bernstein, A. J. "Output Guards and Nondeterminism in 'Communicating Sequential Processes'". *ACM Trans. on Programming Lang. and Systems 2*, 2 (April 1980), 234–238.

3. de Bakker, J.. *Mathematical Theory of Program Correctness.* Prentice–Hall International, London, 1980.

4. Dijkstra, E. W. "The Structure of the T.H.E. Multiprogramming System". *Comm. ACM 11*, 5 (May 1968), 341–346.

5. Dijkstra, E. W. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs". *Comm. ACM 18*, 8 (August 1975), 453–457.

6. Dijkstra, E. W.. *A Discipline of Programming.* Prentice–Hall, Englewood Cliffs, N.J., 1976.

7. Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". *Comm. ACM 12*, 10 (October 1969), 576–580,583.

8. Hoare, C. A. R. "Communicating Sequential Processes". *Comm. ACM 21*, 8 (August 1978), 666–677.

9. Kieburtz, R. B. and Silberschatz, A. "Comments on 'Communicating Sequential Processes'". *ACM Trans. on Programming Lang. and Systems 1*, 2 (October 1979), 218–225.

10. Manna, Z.. *Mathematical Theory of Computation.* McGraw–Hill, New York, 1974.

11. Mao, T. and Yeh, R. "Communication Port: A Language Concept for Concurrent Programming". *IEEE Transactions on Software Engineering 6*, 2 (March 1980), 194–204.

12. Nemes, R. M. *Modular Verification of Asynchronous Systems.* Ph.D. Th., The City University of New York (CUNY), 1983.

13. Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach". *Comm. ACM 19*, 5 (May 1976), 279–289.

14. Parnas, D. L. "On the Criteria to be Used in Decomposing Systems into Modules". *Comm. ACM 15*, 12 (December 1972), 1053–1058.

15. Silberschatz, A. "Communication and Synchronization in Distributed Systems". *IEEE Transactions on Software Engineering 5*, 6 (November 1979), 542–546.

16. Silberschatz, A. "Port–Directed Communication". *Computer Journal 24*, 2 (February 1981), 78–83.