INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University Microfilms International
300 N. Zeeb Road
Ann Arbor, MI 48106

8319787

Nemes, Richard Michael

MODULAR VERIFICATION OF ASYNCHRONOUS SYSTEMS

City University of New York PH.D. 1983

University
Microfilms
International 300 N. Zeeb Road, Ann Arbor, MI 48106

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __√__.

1. Glossy photographs or pages ______
2. Colored illustrations, paper or print ______
3. Photographs with dark background ______
4. Illustrations are poor copy ______
5. Pages with black marks, not original copy __√__
6. Print shows through as there is text on both sides of page ______
7. Indistinct, broken or small print on several pages __√__
8. Print exceeds margin requirements ______
9. Tightly bound copy with print lost in spine ______
10. Computer printout pages with indistinct print ______
11. Page(s) __________ lacking when material received, and not available from school or author.
12. Page(s) __________ seem to be missing in numbering only as text follows.
13. Two pages numbered __________. Text follows.
14. Curling and wrinkled pages ______
15. Other ____

University
Microfilms
International

MODULAR VERIFICATION OF ASYNCHRONOUS SYSTEMS

BY

RICHARD MICHAEL NEMES

A DISSERTATION SUBMITTED TO THE GRADUATE FACULTY IN ENGINEERING IN PARTIAL FULFILLMENT OF TH. REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY, THE CITY UNIVERSITY OF NEW YORK.

1983

THIS MANUSCRIPT HAS BEEN READ AND ACCEPTED FOR THE GRADUATE FACULTY IN ENGINEERING IN SATISFACTION OF THE DISSERTATION REQUIREMENT FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

5/2/83
DATE

CHAIRMAN OF EXAMINING COMMITTEE

May 2, 1983
DATE

EXECUTIVE OFFICER

PROF. M. ANSHEL

PROF. F. BECKMAN

PROF. G. BLOOM

PROF. S. HABIB

DR. S. FORTUNE

SUPERVISORY COMMITTEE

THE CITY UNIVERSITY OF NEW YORK

ABSTRACT

MODULAR VERIFICATION OF ASYNCHRONOUS SYSTEMS

by

Richard Michael Nemes

Adviser: Professor E. A. Akkoyunlu

The semantics of communication are investigated from the viewpoint of modularity and hierarchical program development. Communicating Sequential Processes (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 is developed along the lines of Hoare's axiomatic proof system for sequential programs, extending his system to include CSP-like parallel programs without resort to global invariants. Hierarchial 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. Formal Port semantics are further broadened and extended to include a generalization of the simple Port, termed multi-Port, based on a universal assertion, Kirchoff's Law. As an application of the methodology, a modular proof of the generic single-entry, multiple-user CSP subroutine process is provided.

PREFACE

> Better is the end of a thing
>
> than its beginning
>
> \- Ecclesiastes

This document represents the culmination of a four year study into the nature of communication. My original intent was to produce a grand insight that would illuminate the immediate discipline (computer science), producing a heretofore unimagined formalism that would be termed nothing short of revolutionary. Far reaching ramifications were to have extended as far as the social sciences, and beyond; its universality was to have opened up entire new areas in the fields of psychology (human communication), philosophy (semantics), mathematics (logic), and linguistics. A popular literature was to have spontaneously arisen in the way that Einstein's relativity spawned popular presentations for lay audiences. My idealistic zeal led me to a grand insight that was more personal than expected. It led me to a more modest position. Borrowing from Ecclesiastes once again, I "discovered" that

> that which was will be,
>
> and that which was done is
>
> what will be done,
>
> and there is nothing new under the sun.

Unfortunately, a rediscovery of Ecclesiastes does not constitute a doctoral dissertation, and so I am forced to present secondary insights that, to my regret, may never illuminate the world. I present

these modest results for what they truly are: new ways of looking at things that have always been in existence. If we apply the Ecclesiastic principle once again, new ways of looking at old things are also not "new under the sun," and are at best rediscovered. No matter; what counts in the end is enthusiasm. Of the rest, "all is vanity."

The opening quote applies to this undertaking in two ways. First, I am gratified to be able to conclude this study and bring it to fruition. This treatise is the fruit of that endeavor. Second, the final section on multi-Ports in which CSP subroutines are proved is the most personally gratifying.

Leaving the realm of ethereal speculation, acknowledgments and thanks are due. Without the continuing encouragement, inspiration, guidance, direction, good sense, and personal involvement of my mentor and teacher, Professor E. A. Akkoyunlu, this work would never have seen the light of day. My wife Helen's constant support was an essential ingredient as well. The work of Dijkstra and Hoare forms the bedrock upon which rests this modest edifice.

TABLE OF CONTENTS

	PREFACE	iv
I	INTRODUCTION	1
PART 1		4
II	PREVIOUS RESULTS	5
	A. SUMMARY OF PRIOR RESULTS	5
	B. HOARE AXIOMATICS	10
	1. EXAMPLES	15
	2. WEAKEST PRECONDITIONS AND STRONGEST POSTCONDITIONS	17
	C. PARALLEL PROGRAMS AND THE OWICKI METHOD	19
	1. PARALLEL PROGRAMS	19
	2. THE OWICKI METHOD	22
	D. NONDETERMINACY AND CSP	25
	1. NONDETERMINACY	26
	2. CSP	29
	3. EXAMPLES OF CSP PROGRAMS	33
	E. APT'S METHOD FOR PROVING CSP PROGRAMS	35
	1. APT'S METHOD	35
	2. EXAMPLE	39
PART 2		40
III	MODULARITY AND CSP	41
IV	THE INTERFACE CONCEPT	46

V PORTS AS REPRESENTATION OF THE INTERFACE 49

A. GENERAL DESCRIPTION OF PORTS 49

B. THE PORT AS A PARALLEL-PROGRAMMING LANGUAGE FEATURE 51

1. SYNTAX 51

2. EXAMPLES OF PORT DECLARATION AND I/O COMMAND SYNTAX 54

3. SEMANTICS 55

4. EXAMPLES OF PORT-CSP PROGRAMS 59

C. FORMAL PORT SEMANTICS 66

1. GENERAL DESCRIPTION 66

2. THE INTERFACE DOMAIN 71

3. FORMAL DESCRIPTION 74

4. AXIOMS 77

5. REMARKS CONCERNING THE AXIOMS OF COMMUNICATION 81

6. EXAMPLES OF PORT-CSP PROOFS 85

D. MULTI-PORTS 99

1. GENERAL DESCRIPTION 99

2. FORMAL DESCRIPTION 102

3. SEMANTICS 104

4. EXAMPLES OF MULTI-PORT PROOFS 107

5. MANY-TO-ONE PORTS 111

6. EXAMPLES OF MANY-TO-ONE PORT PROOFS 115

VI CONCLUSION 121

VII APPENDIX - BNF DESCRIPTION OF CSP 124

VIII BIBLIOGRAPHY 127

I INTRODUCTION

This dissertation is, in essence, an investigation into the semantics of communication. Specifically, it examines the semantics of communication within a modified CSP environment (CSP, which stands for Communicating Sequential Processes, is a small language for parallel computations introduced by Hoare in 1978). Semantics are developed along the lines of Hoare's axiomatic verification methodology, forming an extension of his proof system to include CSP-like parallel programs. The novelty of this particular extension is that hierarchical structuring and modularity are fully supported within the language and the associated proof system. The modularity theme runs throughout and underlies the entire development.

The incipient idea behind it all comes from an extension of the electrical engineering notions "specification" and "plug-compatibility" (in software terms, "interface"). While primarily physical concepts, here we extend them to encompass the more abstract realm of parallel programs and formal first-order logic. The underlying inspiration is the physical analogy, and our choice of terminology (Port, Kirchoff's Law) strongly reflects it.

Apart from the satisfaction of seeing an aesthetically pleasing formalism with an associated methodology, there are less abstract (also less immediate) implications as well. With the advent of the inexpensive and versatile microprocessor, it has become advantageous to organize many computations as a collection of tasks (technical term: process) whose executions proceed concurrently. Consequently, a fuller understanding of such systems, from both the formal syntactic and semantic points of view, has become a contemporary issue. There are

now numerous investigations into the characteristics of such distributed systems that are composed of large numbers of processing elements, and many tools and methodologies are being created for their development. This work represents a small contribution to that effort.

The topic is developed like this. First, previous work in the area of program design methodologies is very briefly summarized. Following that, Hoare's formal axiomatic proof method is presented in detail, along with precise coverage of CSP and the first associated proof system, created by Apt et al. All this constitutes Part 1. The shortfalls of Apt's method are then discussed (Part 2), leading into the main portion of this work, which deals with Ports.

Before delving into a detailed discussion of Ports as a syntactic and semantic extension of CSP (the resultant language is termed Port-CSP), we investigate the general notions "modularity," "specifications," and "plug-compatibility" from the electrical-engineer-turned-programmer's point of view. Here the concept of the interface is introduced, with an eye toward the formalism that comes later.

Ports are discussed first from a syntactic angle, complete with BNF. Semantics come only later. The discussion is replete with examples of how Ports might actually be used; the outline of a distributed operating system is the zenith.

Formal Port semantics, which occupies the arena for the remainder, is the main purpose of this entire work. The formalism is firmly established along the lines of Hoare's axiomatic method. Examples of modular proofs are shown in sufficient detail that is

guaranteed to test the endurance of even the hardiest reader. Finally, multi-Ports, the full-blown generalization of simple-Ports discussed until that point, are examined. Semantics developed earlier are broadened and extended to cover these more general constructs. A modular proof of the generic single-entry, mulitple-user subroutine process (the concluding example) is our tour de force.

PART 1

II PREVIOUS RESULTS

A. SUMMARY OF PRIOR RESULTS

Since 1968 the design and verification of large software systems has been receiving considerable attention, spurred by the ever escalating percentages of total systems development and maintenance costs represented by software. The "software crisis," as the trend has come to be known, coupled by software failures in strategically essential systems, has significantly fueled the creation of new language features and design methodologies. Those developments relevant to the issues addressed by this dissertation are presented below.

The earliest techniques arose with efforts to achieve program clarity through restricted control structures (Dijkstra [68c]), top-down program design and stepwise refinement (Mills [71], Wirth [71]), and modularization (Parnas [72a]). In a landmark paper, Dijkstra [68a] demonstrated how an operating system can be organized and developed in a hierarchical fashion so that it can be implemented and tested incrementally, proceeding from level to level in the hierarchy. In a vertical structure such as this, the number of intermodule interfaces tends to be minimized.

Parnas, on the other hand, emphasizes the need for clear distinctions between function and implementation and insists that implementation details of a module be hidden from other modules (Parnas [71, 72a, 72b]). The only externally visible feature of a module, according to Parnas, should be its function. His technique is to choose module decompositions that permit the intermodule interfaces to be defined on a general functional level. This he advocates even at

the risk of introducing execution inefficiencies. The main feature of this method is that it results in systems in which a change to a particular module has minimal impact on the remainder of the system. This notion of module insularity is the essence of what is meant by the term modularity as used throughout this dissertation.

The methodologies described above have been successfully applied in the design of several experimental systems, including THE (Dijkstra [68a]), Venus (Liskov [73]), RC4000 (Brinch Hansen [73a]), and SBS (Akkoyunlu [72]).

New programming languages have been proposed that support data abstraction and encapsulation, including Simula 67 (Dahl [68]), CLU (Liskov [74]), and Alphard (Wulf [74]). Important differences between them notwithstanding, these languages all support user defined abstract "types" by providing a construct (termed class, cluster, or form) to encapsulate a data structure that represents an abstraction. Also provided by the abstract type are procedures that manipulate the encapsulated data structure. Semantic properties of the abstraction are kept independent of its representation and of any bugs in the user program.

As an example, consider the abstract type "push-down stack," represented by an array of elements or, perhaps, a linked list. The nature of the representation, whether it be in terms of an array or in terms of a linked list, is not made visible to the users of the stack. Their interaction with the stack is via procedures PUSH and POP, which are provided as part of the entire package known abtractly as STACK.

On the theoretical side, the major breakthrough was Hoare's development, inspired by Floyd's earlier work on flowcharts (Floyd

[67]), of an axiomatic system for program verification and semantic specification (Hoare [69]). His method associates an axiom, in the form of transformations on logical assertions, with each type of program statement and control structure. Rules of inference allow the establishment of all valid program proofs, subject to the condition of termination. Thus Hoare's methodology is termed a system for proving partial program correctness, the adjective partial indicating that termination is not proveable and that all logical assertions are valid modulo termination.

Hoare's insight was that the program text could be treated semantically as a formal mathematical object. He saw that program semantics can be described without reference to an "execution-time" domain, i.e., without reference to global run-time states, when the program text is viewed as a passive object with respect to the time domain, and as an active element in the space of logical predicates. This is analogous to time-frequency transformation methods (Laplace, Fourier) of analog systems. Hoare's method is primarily local, bottom-up in that full program proofs are built up from individual relations on pairs of predicates specified for each program statement.

Proof techniques involving abstract types include Hoare [72], Spitzen [75], Wegbreit [76], Zilles [75], and Wulf [76]. The language Pascal, whose semantics are specified axiomatically, fully incorporates these concepts (Hoare [73]). The following section describes the general Hoare axiomatic system in detail.

Systems that allow the concurrent (parallel) operation of several processes within a shared environment present problems of a considerably more complex nature. Again, Dijkstra paved the way by

identifying the notion of "cooperating sequential processes" (loosely termed asynchronous systems) and devising the semaphore as a mechanism for achieving proper synchronization and maintaining the integrity of shared data (Dijkstra [68b]).

While the semaphore is an operating system based construct that deals with the problem of synchronization, the monitor (Brinch Hansen [73b], Hoare [74]) is a more generalized language-based tool for creating concurrent programs. The language Concurrent Pascal (Brinch Hansen [75]) is an extension of Pascal in which monitors are supported. Modula (Wirth [77a,b]) has taken over the idea as well. Programs in Concurrent Pascal execute in an elaborate run-time environment, essentially under the auspices of an operating system. Modula, on the other hand, runs with a minimal kernel. There, communication and synchronization are provided by "interface" modules and "device" modules. Context switching is controlled by processes themselves. Modula is intended for real-time process control.

Jones and Liskov propose a language extension for data sharing which allows dynamic binding to shared objects through capabilities (Jones [76]). The emphasis is on access control; synchronization is not provided.

Recent developments in the area of concurrent language design include CSP, a small language for parallel computations (Hoare [78]) upon which this dissertation is largely based, Distributed Processes (Brinch Hansen [78]), and Communication Ports (Mao [80]). CSP is described in detail in a subsequent section.

Axiomatic proof techniques for parallel programs were launched with the work of Owicki and Gries (Owicki [76a,b]). They extend the

Hoare axiomatics for partial correctness to a small parallel-programming language based on shared objects, called "resources," and mutual exclusion. A crucial axiom provides for the use of auxiliary variables that are added to a parallel program as an aid in verification, and the technique makes use of a global invariant that describes the reasonable states of a resource. The resource invariant must be true when parallel execution begins and remains true outside of well-defined critical sections. Their method is used to prove such properties as mutual exclusion, freedom from deadlock (i.e., liveliness), and even termination in certain cases. In this sense their system is more powerful than the Hoare axiomatics and is in some sense "complete" for partial correctness. A description of the Owicki method is presented in a subsequent section.

Hoare [74], Howard [76a,b], and Lamport [77] also discuss proof techniques for parallel systems. Akkoyunlu [78] investigates the verification of operating systems.

Very recently an elegant proof system for establishing the partial correctness of CSP programs has been introduced by Apt (Apt [80]). The technique, which is based on the work of Owicki and Gries, is described in a subsequent section.

B. Hoare Axiomatics

In 1969, Hoare introduced an axiomatic system for proving the partial correctness of a class of Algol-like programs known as WHILE programs (or "GOTO-less" programs). The approach was partially based on the "intermediate assertion" method of Floyd [67] and on results obtained by Böhm and Jacopini (Böhm [66]) who showed that the class of WHILE programs is sufficiently rich to express the logic inherent in any flowchart. In this section we explore the Hoare system in detail, beginning with a formal description of WHILE programs.

A WHILE program consists of a statement S, where statement is defined inductively to include:

1) Assignment statements of the form x:=E, where x is a program variable and E is an expression. The null statement, which is defined as the assignment statement x:=x, is more conveniently written SKIP.
2) Sequencing: S_1; S_2; ... ;S_n is a statement if each S_i is a statement.
3) Alternative statements: IF B THEN S_1 ELSE S_2 ENDIF is a statement if B is a Boolean expression and S_1 and S_2 are statements.
4) Repetitive statements: WHILE B DO S ENDWHILE is a statement if B is a Boolean expression and S is a statement.

As a typical example of a WHILE program, consider the following program which computes the quotient of non-negative integers y and z, leaving the result in x:

```
x:=0;
t:=y;
WHILE t>z DO t:=t-z;
             x:=x+1
ENDWHILE
```

The Hoare proof system involves triples of the form {P}S{Q}, where P and Q are first-order logical formulas, called <u>assertions</u>, and S is a WHILE program. Program variables are among the variables that may be mentioned by assertions P and Q. The meaning of the construct {P}S{Q} is as follows: if P holds (i.e. is valid) prior to execution of S and execution of S successfully terminates, then Q holds following execution of S. The triple establishes only <u>partial correctness</u> of S with respect to P and Q since termination of S is not guaranteed (only the <u>WHILE</u> statement is subject to possible nontermination). P is commonly termed the <u>precondition</u> and Q the <u>postcondition</u>.

The Hoare system is a collection of axioms and proof rules that allow the establishment of valid triples. Given a WHILE program S, the validity of the triple {P}S{Q} is established inductively just as S itself is defined inductively from the four types of statements described earlier. Thus, we specify through axioms and rules what the valid triples are for each of the four types of statements. In this sense we are specifying the meaning, or <u>semantics</u>, of each type of program statement. We begin with the assignment statement.

AXIOM 1) <u>Assignment Axiom</u>

$$\{P(E)\}x:=E\{P(x)\}$$

P(E) stands for the assertion that results from substituting expression E for every free occurrence of x in P. This axiom is more precisely termed an axiom scheme since it encompasses all possible assignment statements.

The rules which follow, termed composition rules, are all of the form $\frac{\alpha}{\beta}$, where α and β are triples and/or logical formulas. α is termed the antecedent and β the consequent. It means the following: if validity of α can be established, then validity of β is deduced. The cancellation law makes this notation particularly convenient:

If $\frac{\alpha}{\beta}$ and $\frac{\beta}{\gamma}$ then $\frac{\alpha}{\gamma}$.

Note that for formulas α and β, $\frac{\alpha}{\beta}$ does not imply the validity of $\alpha \Rightarrow \beta$ (the converse, nevertheless, does hold). For example, $\frac{x=0}{x=1}$ is a valid composition rule in that it is equivalent to $\forall x[x=0] \Rightarrow \forall x[x=1]$. $\forall x[x=0 \Rightarrow x=1]$, however, is not valid.

RULE 1) Sequencing Rule

$$\frac{\{P\}S_1\{R\} \text{ and } \{R\}S_2\{Q\}}{\{P\}S_1;S_2\{Q\}}$$

RULE 2) Alternative Rule

$$\frac{\{P \wedge B\}S_1\{Q\} \text{ and } \{P \wedge \neg B\}S_2\{Q\}}{\{P\}\ \underline{\text{IF}}\ B\ \underline{\text{THEN}}\ S_1\ \underline{\text{ELSE}}\ S_2\ \underline{\text{ENDIF}}\ \{Q\}}$$

RULE 3) Repetitive Rule

$$\frac{\{P \wedge B\} S \{P\}}{\{P\}\ \underline{\text{WHILE}}\ B\ \underline{\text{DO}}\ S\ \underline{\text{ENDWHILE}}\ \{P \wedge \neg B\}}$$

Assertion P is termed a loop invariant since the validity of P is preserved under execution of the body of the loop.

The final rule permits the weakening of postconditions and the strengthening of preconditions of valid triples, two intuitively well-founded operations.

RULE 4) Rule of Inference

$$\frac{\{P\} S \{Q\} \text{ and } (R \Rightarrow P) \text{ and } (Q \Rightarrow U)}{\{R\} S \{U\}}$$

A concise method of demonstrating the validity of $\{P\}S\{Q\}$, where S is an entire program or program segment, is to annotate the program text with intermediate assertions so that a valid intermediate triple is formed with each individual program statement. The resultant text is termed the annotated program (also annotated proof or annotated text). In the annotated program, the precondition of each statement is the postcondition of the previous one. The annotated program, then, has the general appearance

$$\{P\} \equiv \{P_1\} S_1 \{P_2\}; S_2 \{P_3\}; \ \dots \ \{P_{n-1}\}; S_{n-1} \{P_n\} \equiv \{Q\} \quad ,$$

where $\{P_i\} S_i \{P_{i+1}\}$ is valid for each i. Note that the size of a proof is at least as long as the program itself.

The annotated text is conveniently derived from the program text as follows: starting with the postcondition Q, work back through the text, statement by statement, establishing valid preconditions at each step. Notice that Axiom 1 is easiest to apply when proceeding in this fashion. The only nonmechanical aspect of the entire process that usually requires creativity on the part of the person writing the proof is finding adequate loop invariants.

1. EXAMPLES

1. Consider the program consisting of the assignment statement x:=7. Then by Axiom 1, {7=7}x:=7{x=7} is valid, and since (7=7) ≡ TRUE, we conclude {TRUE}x:=7{x=7} from Rule 4. From Rule 4 we can also conclude {TRUE}x:=7{TRUE}.
2. {P}SKIP{P} is valid for any P, from Axiom 1.
3. {P}S{TRUE} is valid for any S and any P since (Q=>TRUE) is valid for any Q, and from Rule 4.
4. {FALSE}S{P} is valid for any S and any P since (FALSE=>P) is valid for any P, and from Rule 4.
5. Consider program S that computes $z=b^a$ for integers a and b:

```
        y:=a;
        x:=b;
        z:=1;
   WHILE y≠0 DO
        IF odd(y) THEN y:=y-1;z:=z*x ELSE SKIP ENDIF;
        x:=x*x;
        y:=y/2
   ENDWHILE
```

We will prove {TRUE}S{$z=b^a$}. The annotated text appears as follows:

{TRUE}

y:=a

{$b^y=b^a$};

x:=b

{$x^y=b^a$};

```
z:=1
{z·x^y=b^a};
WHILE y≠0 DO
{z·x^y=b^a} IF odd(y) THEN {z·x^y=b^a∧odd(y)}y:=y-1
                            {z·x^(y+1)=b^a∧even(y)};
                            z:=z*x{z·x^y=b^a∧even(y)}
                       ELSE {z·x^y=b^a∧even(y)}SKIP
                            {z·x^y=b^a∧even(y)}
                       ENDIF
{z·x^y=b^a∧even(y)};
x:=x*x
{z·x^(y/2)=b^a∧even(y)};
y:=y/2
{z·x^y=b^a}
ENDWHILE
{z=b^a}
```

The loop invariant P is given by $z \cdot x^y = b^a$. The loop exit condition, $(P \wedge \neg B)$ of Rule 3, yields the desired postcondition $z=b^a$. Note that Axiom 1 may be applied to assignment statements involving integer division only when integer quantities being divided are known to divide evenly prior to execution. This restriction is clearly unnecessary when division involves floating-point quantities.

2. WEAKEST PRECONDITIONS AND STRONGEST POSTCONDITIONS

We have thus far developed the notion of program semantics in terms of logical assertions. We can, however, take a more general point of view and arrive at an equivalent but more elegant construct by considering programs as transformations on assertions. Programs can be viewed as transformations that map preconditions into postconditions, and vice-versa. The problem with such transformations in general is that they are multi-valued: given a program S and a precondition P, Rule 4 states that there exist any number of postconditions Q such that {P}S{Q} is a valid triple. Similarly, given S and a postcondition Q, there are many preconditions P that form valid triples. We can, however, turn these multi-valued transformations into well-defined functions by mapping postconditions into _weakest preconditions_, and preconditions into _strongest postconditions_. Both are unique up to logical equivalence. This is developed as follows, beginning with weakest preconditions.

1) For each program S and assertion Q, _wlp(S,Q)_ (following Dijkstra [75, 76]'s notation) denotes the weakest liberal precondition of Q with respect to S. The adjective liberal signifies that termination of S is not guaranteed. Whenever termination of S is guaranteed we write simply _wp(S,Q)_. This is the case, for example, when S contains no _WHILE_ statements. {P}S{Q} is defined to be a valid triple if and only if (P=>wlp(S,Q)) is valid for all values of the free variables.
2) wp(x:=E,Q(x)) = Q(E)
3) $wlp(S_1;S_2,Q) = wlp(S_1,wlp(S_2,Q))$

4) $wlp(\underline{IF}\ B\ \underline{THEN}\ S_1\ \underline{ELSE}\ S_2\ \underline{ENDIF},Q) =$
$(B \Rightarrow wlp(S_1,Q)) \wedge ((\neg B) \Rightarrow wlp(S_2,Q))$

5) If $(P \wedge B) \Rightarrow wlp(S,P)$ is valid for all values of the free variables, then $P \Rightarrow wlp(\underline{WHILE}\ B\ \underline{DO}\ S\ \underline{ENDWHILE}, P \wedge \neg B)$ is valid for all values of the free variables.

Each of the above formulas has a dual, which is expressed in terms of strongest postconditions. The development follows that of de Bakker [80].

1') For each program S and assertion P, $\underline{slp(P,S)}$ denotes the strongest liberal postcondition of P with respect to S. The adjective liberal signifies that termination of S is not guaranteed. Whenever termination is assured we write simply $\underline{sp(P,S)}$. This will be the case when S contains no loops. {P}S{Q} is defined to be a valid triple if and only if $(slp(P,S) \Rightarrow Q)$ holds for all values of the free variables.

2') $sp(P(x), x:=E(x)) =$
$\exists y[P(y) \wedge x=E(y)] \wedge y \nmid x \wedge$ y is not free in P $\wedge$ y is not mentioned in E

3') $slp(P,S_1;S_2) = slp(slp(P,S_1)S_2)$

4') $slp(P,\underline{IF}\ B\ \underline{THEN}\ S_1\ \underline{ELSE}\ S_2\ \underline{ENDIF}) =$
$slp(P \wedge B,S_1) \vee slp(P \wedge \neg B,S_2)$

5') If $slp(P \wedge B,S) \Rightarrow P$ holds for all values of the free variables, then $slp(P,\underline{WHILE}\ B\ \underline{DO}\ S\ \underline{ENDWHILE}) \Rightarrow (P \wedge \neg B)$ holds for all values of the free variables.

C. PARALLEL PROGRAMS AND THE OWICKI METHOD

1. PARALLEL PROGRAMS

Up to this point we have been dealing exclusively with what are termed sequential programs, that is, programs that do not exhibit characteristics of parallelism. Thus, WHILE programs, described earlier, constitute the class of sequential programs and the associated Hoare axiomatic system is a methodology for proving partial correctness of sequential programs. We turn now to a broader class of programs, namely, the class of parallel programs.

A parallel program, equivalently concurrent program, is a collection of programs that are executed simultaneously. In general, these simultaneous executions do not proceed in an isolated manner; rather, they involve the mutual interaction of the independently executing programs. Each independently executing member of the collection is more commonly termed a process. Processes whose parallel executions proceed in an asynchronous manner with respect to one another are termed asynchronous processes. We will be considering only asynchronous processes. Consequently, no assumptions are made about the relative speeds of the parallel processes with which we will be dealing. The results of the execution of asynchronous processes can depend on the unpredictable order in which statements of the various processes are executed. Hence, high-level parallel-programming languages contain synchronizing features designed to ensure predictability.

Parallel-programming systems and their associated high-level languages are conveniently classified into two distinct categories:

1) parallel processes which reference common variables, and 2) parallel processes for which there is no such sharing. The first category is customarily associated with multiprogrammed uniprocessor architectures while the second more accurately corresponds to a system of multiprocessors each with private storage. In the latter category each process is typically assigned its own processor, while in the former a single processor is time-shared among the processes. We begin by describing a typical parallel-programming language based on shared data, and proceed to define the Owicki proof system for that language.

The parallel-programming language used here is derived from the WHILE programming language described earlier. It includes: assignment statements, sequencing, alternative statements, repetitive statements, plus two statements that are intended specifically for parallel processing. The first is used to initiate parallel execution and takes the form

$$\underline{\text{RESOURCE}}\ r_1\ (\text{variable list}), \ldots, r_m\ (\text{variable list})$$
$$[S_1 \| \cdots \| S_n]$$

, where resource r_i is a set of logically related shared variables and the S_i are parallel processes, i.e., statements executed in parallel. In general, a parallel program may have any number of such statements.

The second statement provides the necessary synchronization for referencing shared data and takes the form $\underline{\text{WITH}}\ r\ \underline{\text{WHEN}}\ B\ \underline{\text{DO}}\ S\ \underline{\text{ENDWITH}}$, where r is a resource, B a Boolean expression, and S a statement mentioning variables of r. A process executing such a statement is delayed until B is TRUE and r is not being used by another process, at which time S is executed. While S is being executed no other process

can reference variables of r. No assumptions are made regarding the order in which competing processes are granted control of a resource. WITH statements may appear only within a parallel execution statement ($[S_1 || \ldots || S_n]$ construct shown above) and may not be nested when they are for the same resource. The purpose of the WITH statement, then, is to avoid interprocess interference when referencing shared data by providing access on a mutually exclusive basis; at most one process has access to shared variables at any point in time. The following two restrictions ensure that all shared variables are indeed protected by the resource mechanism:

1) Variables belonging to resource r may be referenced in a parallel execution statement only by WITH r statements;
2) Variables changed by a process may not be referenced by other processes unless they belong to a resource.

The following (from Owicki [76a]) is an example of a simple parallel program. Its purpose is to add 2 to variable x.

```
RESOURCE r(x)
[WITH r WHEN TRUE DO x:=x+1 ENDWITH ||
 WITH r WHEN TRUE DO x:=x+1 ENDWITH]
```

2) THE OWICKI METHOD

The Owicki proof system extends Hoare's axioms and rules for sequential programs to include proof rules for the parallel execution and WITH statements. These two additional rules require an assertion I(r), termed the invariant for resource r, that defines the proper states of the resource. The invariant I(r) must hold at all points of the parallel execution statement that are outside WITH r statements. The two additional rules are given as follows.

RULE 5) Parallel Execution Rule

$\{P_1\}S_1\{Q_1\}$ and $\{P_2\}S_2\{Q_2\}$ and ... and $\{P_n\}S_n\{Q_n\}$,

and no free variable in P_i or Q_i is changed by S_j, $i \neq j$,

and all variables in I(r) are in resource r

$\{P_1 \wedge \ldots \wedge P_n \wedge I(r)\}$ RESOURCE r $[S_1 \| \ldots \| S_n]$ $\{Q_1 \wedge \ldots \wedge Q_n \wedge I(r)\}$

RULE 6) WITH rule

$\{I(r) \wedge P \wedge B\}S\{I(r) \wedge Q\}$, where I(r) is the invariant from the containing parallel execution statement, and no free variable in P or Q is changed by another process

---

{P}WITH r WHEN B DO S ENDWITH{Q}

Rule 5 provides for the construction of a proof of a parallel program from proofs of individual processes. The antecedent specifies

that the individual proofs must be effectively disjoint; what makes the conjunctions of the preconditions and postconditions meaningful and binds them together is the invariant.

The addition of Rules 5 and 6 proves insufficient, however, for proving even the simplest of parallel programs. The repertoire of variables used by a typical program is generally not rich enough to allow the desired pre- and postconditions to be broken down into an invariant plus pre- and postconditions for each of the constituent processes. It is impossible, for example, to prove {x=0}S{x=2} for the program shown above unless we admit further variables into the program. This is done with the aid of an additional axiom. The additional variables are termed auxiliary variables.

AXIOM 2) Auxiliary Variable Axiom

$$\frac{\{P\}S\{Q\}\text{, and x not free in P or Q, and x appears in S only in assignment statements of the form x:=E (x may appear in E)}}{\{P\}S'\{Q\}\text{, where S' is obtained from S by deleting all assignment statements to x}}$$

With the aid of auxiliary variables y and z we can now prove the above program. y tracks the progress of the first process and z tracks the second. The invariant I(r) ≡ (x=y+z) tracks their joint progress. The annotated text follows.

```
{x=0}
y:=0;
z:=0
```

```
{y=0∧z=0∧I(r)};
RESOURCE r(x,y,z)
[{y=0} WITH r WHEN TRUE DO
              {y=0∧I(r)}
              x:=x+1;y:=1
              {y=1∧I(r)}
       ENDWITH
 {y=1}
||
 {z=0} WITH r WHEN TRUE DO
              {z=0∧I(r)}
              x:=x+1;z:=1
              {z=1∧I(r)}
       ENDWITH

 {z=1}
]
 {y=1∧z=1∧I(r)}
```

Since (y=1∧z=1∧I(r))=>x=2, the desired postcondition is established. y and z are eliminated by the auxiliary variable axiom and the proof is complete.

D. NONDETERMINACY AND CSP

In the previous section we presented a parallel-programming language based on shared objects and an associated axiomatic proof system. We now turn to a parallel-programming language that does not provide data sharing, CSP.

Communicating Sequential Processes, CSP for short, is based on Dijkstra's guarded commands (Dijkstra [75, 76]), which introduced the notion of nondeterminacy in sequential programming. Before describing CSP in detail we shall consider nondeterminacy, guarded commands, and related proof rules as they pertain to sequential programs.

1. NONDETERMINACY

The sequential-programming language that we now consider consists of assignment statements, sequencing, and two new statements constructed from guarded commands. A guarded command takes the form

$$\text{guard} \rightarrow \text{guarded list} \quad ,$$

where guard is a Boolean expression and guarded list a sequence of one or more statements separated by semicolons. A guarded command is not a statement; rather, it is a component from which statements may be composed. A guarded list can be selected for execution only if its guard is TRUE. When a guarded list is selected for execution, its statements are executed in the usual sequencing order. An example of a guarded command is

$$x \geq y \rightarrow m:=x \quad .$$

The first statement constructed from guarded commands is the generalized alternative statement. It takes the form

$$[G_1 \square G_2 \square \ldots \square G_n] \quad ,$$

where G_i is a guarded command. It operates as follows: if none of the guards are TRUE, the program fails, i.e., aborts; otherwise a guarded list with a TRUE guard is nondeterministically selected for execution. The order in which the G_i appear is immaterial. An example of such a statement is the program that assigns to m the value max(x,y):

$$[x \geq y \rightarrow m:=x \square y \geq x \rightarrow m:=y] \quad .$$

If x=y, it is indeterminate - as well as inconsequential - which guarded list will be executed.

The second statement constructed from guarded commands is the generalized repetitive statement. It takes the form

$$*[G_1 \square G_2 \square \ \dots \ \square G_n] \qquad ,$$

where G_i is a guarded command. This statement when executed will not terminate until all guards are FALSE. Until then, a guarded list with a TRUE guard is nondeterministically chosen for execution. Upon completion, another guarded list with a TRUE guard is nondeterministically chosen for execution, and so on. When the repetitive statement terminates, all guards are FALSE. An example of this statement is the program that computes the value gcd(x,y):

$$*[x>y \rightarrow x:=x-y \ \square \ y>x \rightarrow y:=y-x] \qquad .$$

Dijkstra has provided proof rules for these constructs in the form of weakest precondition semantics. For the alternative statement we have

$$wlp([g_1 \rightarrow S_1 \square \ \dots \ \square g_n \rightarrow S_n],Q) =$$

$$\sum_{i=1}^{n} g_i \wedge \left(\prod_{i=1}^{n} g_i \Rightarrow wlp(S_i,Q) \right) \qquad ,$$

where Σ and Π stand for logical disjunction and conjunction, respectively. The first term assures that at least one guard is TRUE and the second term assures that all eligible guarded lists lead to an acceptable postcondition.

Semantics for the repetitive statement rely on an invariant assertion, as in the case of WHILE DO. Here we have the rule

$$\frac{(P \wedge \sum_{i=1}^{n} g_i) \Rightarrow wlp([g_1 \rightarrow S_1 \square \ \dots \ \square g_n \rightarrow S_n], P)}{P \Rightarrow wlp(*[g_1 \rightarrow S_1 \square \dots \square g_n \rightarrow S_n], P \wedge \prod_{i=1}^{n} \neg g_i)} \quad .$$

2. CSP

CSP (Hoare [78]) is based on an underlying processor topology consisting of a number of similar, self-contained processors each with its own private store. There is no common store. Consequently, mutual exclusion (competition) is not a feature of the language; rather, simultaneity (cooperation) is the main theme. Interprocess communication, based on the well-understood assignment statement, takes the form of input and output commands between parallel processes.

Interprocess communication occurs when a process names another as destination for output, which in turn names the first as the input source. When matching I/O commands are synchronized, communication takes the form of an assignment from an output to an input variable. Automatic buffering is not provided; rather, an I/O command is delayed until the specified process reaches a matching I/O command. Such delays are invisible to either process.

A complete BNF description of CSP syntax is found in the Appendix. Types, declarations, and expressions are not treated however; they are Pascal-like and can be inferred from the examples. The basic syntactic category is <command>, which corresponds to what we have previously termed statement. The main features of the language are outlined below.

1) A command list, which is a sequence of commands separated by semicolons, corresponds to sequencing in WHILE programs.
2) A declaration within a command list introduces a fresh variable whose scope extends from the declaration to the end of the command list.
3) The command $[P_1 || \ldots || P_n]$ expresses parallel execution of

processes P_1, ... ,P_n. The P_i, which are termed "constituent processes of a parallel command," must be disjoint in the sense that P_i may mention no variables modified by P_j, $i \neq j$. The command terminates successfully when each of the constituent processes has terminated successfully.

4) A process is of the form P::S, where P is the process label and S the command list constituting the body of the process. A process with label subscripts that include one or more ranges represents an array of identical parallel processes. For example, P(j:1..n)::S represents $P(1)::S_1 || \ldots || P(n)::S_n$, where S_i stands for S with every occurrence of variable j replaced by the number i.

5) Constituent processes P_i and P_j, $i \neq j$, of a parallel command communicate via the input and output commands $P_j?x$ (in S_i, the body of process P_i) and $P_i!y$ (in S_j, the body of process P_j). Execution of the two I/O commands within their respective processes corresponds to execution of the assignment command x:=y. "?" always signifies input and "!" output. A process reaching an I/O command is delayed until the addressed process has reached a matching I/O command. An I/O command fails if the addressed process has terminated.

For matching I/O commands to successfully synchronize and complete execution, the implied assignment command must be between compatible data types. Thus, $P_j?x$ and $P_i!y$ will not synchronize with each other if, for example, y is of type real and x is of type integer.

Similarly, structured output expressions must match structured input targets. For example, $P_i!x$ and $P_j?(y,z)$ cannot

synchronize unless x is of the form (a,b). If it is, then the values of a and b are assigned to y and z respectively.

"Constructors" are used to differentiate between otherwise identical data types. Thus, for integer variables x and y, c(x) and d(y) are distinct structured types since constructors c and d are distinct. P_j?c(x) and P_id(y) will not synchronize; P_j?d(x) and P_i!d(y) will. A structured expression without components is known as a <u>signal</u>. P_j?d() and P_i!d() can synchronize, but without transfer of data from P_j to P_i.

6) Guarded commands, the alternative command, and the repetitive command are as previously described, except that guards may contain I/O commands. In general, a guard may be a Boolean expression, an I/O command, or a combination of both (separated by ;). A guard evaluates to FALSE if the Boolean is FALSE or the I/O command has failed (i.e., the addressed process has terminated). A guard evaluates to TRUE if its Boolean is TRUE and the process addressed by the I/O command is ready to synchronize and execute the I/O. In this case, the I/O command is actually executed only if and when the associated command list is chosen for execution, which is done nondeterministically. A guard whose Boolean evaluates to TRUE but whose I/O command addresses a process not yet ready to execute a matching I/O command evaluates neither to TRUE nor FALSE. The associated command list is not eligible for execution, but in the case of a repetitive command it may become so at a later time. The use of I/O commands as a choice mechanism in connection with guarded commands constitutes the main feature of CSP.

As with processes, a guarded command with one or more subscripts represents a series of identical guarded commands. Here, however, the subscripts precede the command. Thus, (j:1..n)g->S is equivalent to

$$g_1 \text{->} S_1 \square \; \ldots \; \square g_n \text{->} S_n \quad ,$$

where g_i->S_i stands for g->S with every occurrence of variable j replaced by the number i. A declaration appearing in a guard introduces a fresh variable whose scope extends from the declaration to the end of the guarded command (i.e., to the end of the associated command list).

3. EXAMPLES OF CSP PROGRAMS

1. Integer Semaphore:

This example from Hoare [78] illustrates scheduling capabilities of CSP. Consider an array X(i:1..100) of parallel processes among which an integer semaphore S is shared. We implement S as a distinct process, with the semaphore operations P and V accessed by the I/O commands S!P() and S!V() respectively. Here P and V are constructors used to differentiate between the two signals. All processes of the array access the semaphore using the two commands as shown above. Recall that the V operation increments the semaphore and the P operation decrements it, the P operation being delayed if the semaphore is not positive.

The full system is given by [S||X(i:1..100)], where S is defined by

```
S::val:integer;val:=initial value of semaphore;
        *[(i:1..100)val>0;X(i)?P()->val:=val-1
        □(i:1..100)X(i)?V()->val:=val+1]     .
```

The semaphore process S terminates when all processes of array X have terminated.

2. Producer-Consumers:

This generic example illustrates a pattern that fits many small parallel systems and portions of larger ones. Consider a producer process P and 100 identical consumer processes C. Execution of the producer and each of the consumers proceeds in

parallel, so the entire system is given by

[P||C(i:1..100)::CONSUMER]. P, which produces data to be consumed

by the consumer, is given by

```
P::x:real;*[produce(x)->[(i:1..100)C(i)!x->SKIP]]
```

.

Data produced by P is handed to any available consumer, the

identity of which is not noted.

Each consumer is given by

```
CONSUMER = x:real;*[P?x->consume(x)]
```

.

A delay occurs when P produces data and all consumers are busy consuming, or when a free consumer must wait for P to produce more data. The required synchronization is managed automatically by the repetitive command. Termination occurs when P no longer produces and each consumer completes consumption of its last data item.

E. APT'S METHOD FOR PROVING CSP PROGRAMS

1. APT'S METHOD

The Apt (Apt [80]) axiomatic proof system for partial correctness of CSP programs, based heavily on the work of Owicki, employs the original idea of "joint cooperation between isolated proofs" of individual parallel processes. Individual process proofs are strictly disjoint, mentioning only variables local to a process, and conform to the sequential proof rules mentioned earlier. Postconditions of I/O commands, however, are the exception. They cannot be established solely within the context of a local process since they are dependent to some extent on the preconditions of matching I/O commands in the addressed process. Consequently, their validity can only be established jointly by the two preconditions "cooperating" in the establishment of the two postconditions. As in Owicki's system, auxiliary variables are employed and individual process proofs are combined using a parallel execution rule.

Since syntactically matching I/O commands may fail to match semantically, i.e., synchronization may be logically precluded, Apt includes an assertion termed the global invariant, that specifies precisely which pairs of matching commands can synchronize. A well-defined notion of critical section, termed bracketed section, delimits a purely sequential neighborhood of each I/O command within which the global invariant need not hold, but outside of which it must. The brackets, which appear as < >, are inserted into the program text forming the annotated proof.

The global invariant is a generally useful device that can be used for other than specifying which I/O commands can synchronize. It

can establish loop exit conditions, for example, and in many cases an entire proof may consist of an extensive global invariant and relatively trivial pre- and postconditions in the individual proofs. Nevertheless, Apt asserts the existence of a canonical proof form in which the bracketed sections consist of an I/O command and a local-history variable update (usually to an auxiliary variable). In the canonical form, the global invariant is minimal, mentioning only I/O variables and local-history variables.

Apt's formal proof system is presented, followed by an example. I/O commands will be denoted generically by the letters w and x, the global invariant by I, P_i, i=1, ... ,n, represents a process, and b_j stands for a Boolean expression. Well-known axioms and rules for sequential programs are omitted.

1) Parallel Execution:

$$\frac{\{p_i\}P_i\{q_i\} \text{ cooperate and no free variable in I is changed outside a bracketed section}}{\{\prod_{i=1}^{n} p_i \wedge I\}[P_1 || \; \cdots \; || P_n]\{\prod_{i=1}^{n} q_i \wedge I\}}$$

2) Bracketed sections take the form

$<S_1;w;S_2>$ or $<w\text{->}S_1>$,

where S_1 and S_2 contain no I/O commands. Bracketed sections <U> and <V> from distinct processes <u>match</u> if U and V contain syntactically matching I/O commands.

3) Alternative and Repetitive Commands Involving Communication:

a. $$\frac{\{p \wedge b_j\}w_j\{r_j\},\ \{r_j\}S_j\{q\},\ j=1,\ \dots\ ,m}{\{p\}[b_1;w_1\text{->}S_1 \;\square\; \cdots \;\square\; b_m;w_m\text{->}S_m]\{q\}}$$

b. $$\frac{\{p \wedge b_j\} w_j \{r_j\},\ \{r_j\} S_j \{p\},\ j=1,\ \ldots\ ,m}{\{p\} * [b_1; w_1 \text{->} S_1 \square\ \ldots\ \square b_m; w_m \text{->} S_m] \{p\}}$$

4) Proofs $\{p_i\}P_i\{q_i\}$ cooperate if

a. assertions used in the proof of $\{p_i\}P_i\{q_i\}$ mention no free variable changed by P_j, $i \neq j$;

b. $\{u_1 \wedge v_1 \wedge I\} U || V \{u_2 \wedge v_2 \wedge I\}$ holds for all matching bracketed sections <U> (in P_i) and <V> (in P_j), where $\{u_1\}U\{u_2\}$ and $\{v_1\}V\{v_2\}$ are taken from the proofs of P_i and P_j, respectively.

5) Communication Constructs:

a. $$\frac{\{p\} S_1; S_3 \{p_1\}, \{p_1\} w || x \{p_2\}, \{p_2\} S_2; S_4 \{q\}}{\{p\} (S_1; w; S_2) || (S_3; x; S_4) \{q\}}$$

, where w and x match and S_1, S_2, S_3, and S_4 contain no I/O commands.

b. $$\frac{\{p\} (w; S) || S_1 \{q\}}{\{p\} (w \text{->} S) || S_1 \{q\}}$$

6) Communication:

$$\{TRUE\} P_i ? v || P_j ! y \{v=y\}$$

7) Preservation:

$\{p\}S\{p\}$ if no free variable of p is changed by S.

8) I/O:

$\{p\}w\{q\}$

9) Auxiliary Variables:

Each member v of a set of variables A is an auxiliary variable with

respect to program S if v appears in S only in assignments of the form y:=E, where y is any variable in A. If q contains no free auxiliary variables with respect to S, and S' is obtained from S by deleting all assignments to auxiliary variables, then

$$\frac{\{p\}S\{q\}}{\{p\}S'\{q\}}$$

10) Elimination of Auxiliary Variables from Preconditions:

$$\frac{\{p(v)\}S\{q\},\ v \text{ is an auxiliary variable},\ z \text{ not in } S \text{ and not free in } q}{\{p(z)\}S\{q\}}$$

2. EXAMPLE

Consider the following two parallel processes, where u and v are integers, $u \geq 0$, $v>1$:

```
P:: Q!u;f:=TRUE;*[f;Q?yes()->v:=v*v□f;Q?u->f:=FALSE]
Q:: P?x;*[even(x)->P!yes();x:=x/2];P!x
```

We prove that for constant k, [P||Q] leaves invariant the relation $v^u=k$, i.e., $\{v^u=k\}$[P||Q]$\{v^u=k\}$. The annotated proof follows.

```
P:: {v^u=k}
    <Q!u>{TRUE};f:=TRUE{TRUE};
    *[f;<Q?yes()->v:=v*v>□f;<Q?u>->f:=FALSE]{TRUE}

Q:: {(¬av_1)∧(¬av_2)}
    <P?x;av_1:=TRUE>{LI};
    *[{LI}even(x)->{LI}<P!yes();x:=x/2>{LI}]{LI};
    <P!x;av_2:=TRUE>
    {av_1∧av_2}
```

LI, the loop invariant for process Q, is given by $(av_1 \wedge (\neg av_2))$. av_1 and av_2 are auxiliary variables; the global invariant is $(av_1 \Rightarrow v^x=k) \wedge (av_2 \Rightarrow x=u)$.

PART 2

III MODULARITY AND CSP

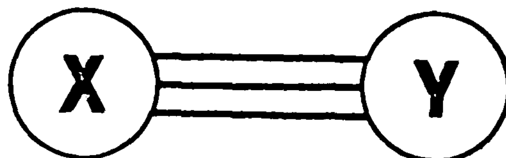
CSP was conceived primarily as a tool for the study of parallelism and nondeterminacy in a nonshared environment, and to this extent the language succeeds. Though the process structure and communication mechanism do encourage some degree of modularity, they do not admit the measure expected in a production environment. It is assumed, for example, that all modules and interfaces are known at the outset, an unrealistic approach when dealing with large, highly-parallel distributed systems composed of significant numbers of processing elements. CSP functions best when an entire program is written all at once, preferably by one individual, and further modification is avoided.

Production environments in which CSP-type languages are adopted are generally quite demanding. Problems that arise are due mainly to size, but an asynchronous system need not be large for problems generally attributed to size to become manifest. Typical problems include incomplete, vague, and nonspecific specifications; small but endless changes to the external environment; internal modifications for improved performance; and large numbers of individuals participating in system development and maintenance.

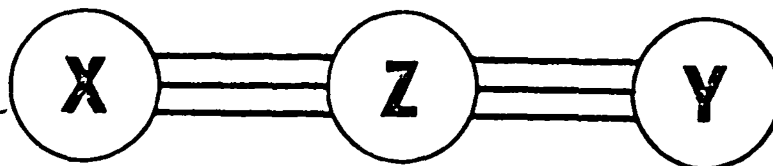
In order to cope with these difficulties, support at the language level is necessary. Experience with Modula has borne this out. A system intended for a dynamic environment must be structured in a way that enables it to remain stable and yet maintainable. Since such a system will change in unanticipated ways, the effect of a change must be localized. As much as possible, process A must be kept immune from changes to process B. On the verification side, local changes should

require only local reverification, and verification should proceed hierarchically in accordance with the hierarchical structure of a system. With this in mind, we now investigate some drawbacks of CSP and its associated proof system as they are currently defined.

First, consider that communicating processes must name each other explicitly. This makes it virtually impossible to create a library of well-known utility processes to be accessed by user processes in the same way that a Fortran subroutine library is used. Similarly, it complicates the situtation of separately developed and compiled communicating processes that are combined into a complete system. In such a case changing even the name of a process can prove difficult. Consider, for example a pair of communicating processes X and Y:



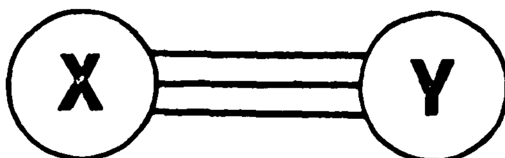
It may be convenient at some point to interpose an intermediate process Z:



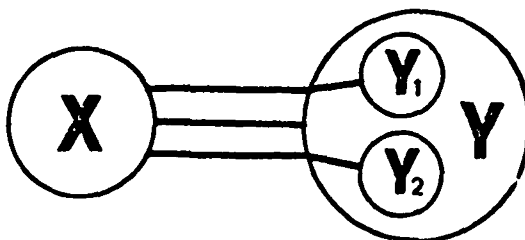
In a network, for which CSP seems particularly attractive, this can

occur when X is moved to a different node site. Z acts as the ghost of X at the original site, and it is desirable to hide these details from both X and Y.

Second, and more important, is the prohibition in CSP of communicating with a nested process, thus making it impossible to obtain a hierarchically structured system. Consider, once again, a pair of communicating processes X and Y:



By definition, X and Y must be constituent processes of a parallel command. At some point it may be desirable to divide Y into parallel subprocesses Y_1 and Y_2:



Such a change should be totally transparent to X, requiring no modifications, provided that the interface remains intact, i.e., the external behavior of both versions of Y are identical. CSP does not

permit this since X must name its correspondents explicitly, as mentioned earlier, and communication may occur only between "constituent processes of a parallel command." Thus, X may communicate with Y, but not with Y_1 or Y_2.

Consequently, we arrive at the irony that though CSP is an excellent language for specifying most of the components of a highly-distributed operating system, it is not possible to write a complete CSP operating system capable of running CSP user-programs since the system must explicitly name the users with which it communicates. Nor would a hierarchy within such a system be possible since all communicating processes - users and system - must occur on the same level.

The Apt proof system is even less tolerant of change than the language itself. "Joint cooperation between isolated proofs" has little if any tolerance to even minor program modifications. An entire system must be reverified if even a single process is changed in the slightest way. Modifications that affect the global invariant in any way require revalidation of cooperation tests between every pair of syntactically matching I/O commands, even in processes totally unrelated to and logically removed from the ones changed. This is because the global invariant contains control information spanning the entire system. Consequently, top-down hierarchical verification in CSP is not feasible.

In addition, the proof system is capable of validating only systems for which all modules have been finalized, thereby making it impossible to verify a module against a <u>specification</u> of the environment in which it will be embedded. Accordingly, unit

verification is not feasible within the Apt system. A more general verification strategy is required, one in which a process can be validated against the entire class of modules that meet the specifications of its external environment.

The remainder of this dissertation is aimed at amending these difficulties within CSP and exhibiting a proof strategy that is capable of providing truly modular verification. Before we proceed with this, however, we must explain in better detail the abstract approach that underlies the development of the modular verification technique.

IV THE INTERFACE CONCEPT

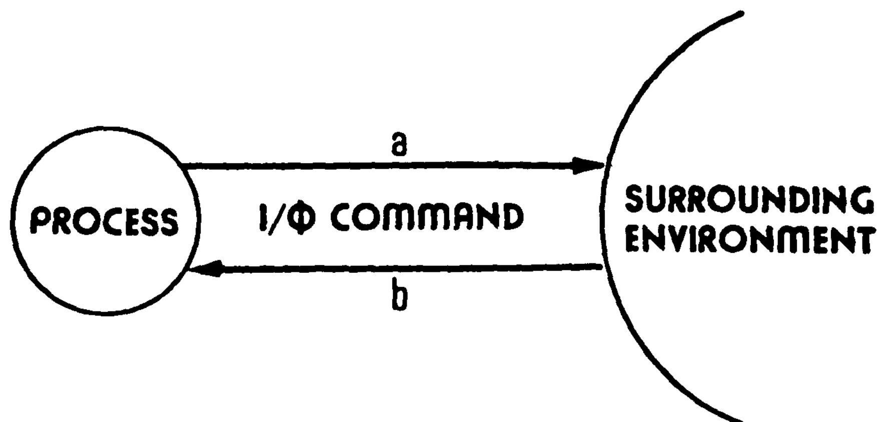
The concept of interface, as developed here, is centered on interprocess communication. We expect the interface to capture the interaction of a process with its external environment, the interaction being expressed in terms of the semantics of communication. The focus is on pairs of objects of the form (module,interface), viewed as an entity. The first component, module, represents one or more processes, while the second component, the interface, is a description of how the module interacts with its surroundings. Since the interface is meant to mesh with the Hoare axiomatic system at the local process level, the interface must contain logical assertions; for the most part it consists almost exclusevely of assertions.

The interface itself consists of two subcomponents, dual to one another:

1. The external specification of the module as viewed from without, i.e., the functional behavior of the module considered as a black box. In systems theory this corresponds to the external description of a dynamical system.
2. The specification of the external environment as viewed from within the module. (Note: this does not correspond to the internal description of a dynamical system, as might be perhaps expected).

The relationship embodied in this notion is expressed pictorially as shown below.

Arrow b represents assumptions made by the process (module) about the environment. If communication is to be coordinated and meaningful, the



environment must satisfy those assumptions. Dually, arrow a represents assumptions made by the environment about the process. Here the process is seen as a black box. Again, if communication is to succeed semantically, the process must satisfy those specifications. The duality of the relationship is that each domain's assumptions are the other domain's obligations. When the I/O command is input with respect to the process and output with respect to the environment, we call the assertion corresponding to arrow a $\underline{\alpha}$, and to arrow b $\underline{\beta}$, and vice versa.

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 should have no affect on the external environment. In effect, a module will be verified against the interface rather than against a specific version of the remaining system. This uncouples the verification of modules from another and provides true unit verification. Verifying a module against an interface, then, is equivalent to verifying it against an entire class

of programs, infinite in number. While this capability already exists within the realm of purely sequential programming (Alphard - Wulf [76]- is a good example), it has not, up till now, been made available to parallel systems.

Another benefit of the interface approach is that it allows closed and complete proofs which are not possible otherwise. To illustrate, {TRUE}[X::Y!2||Y::X?a]{a=2} is Apt verifiable, whereas [U::V!2||V::U?a;PRINT!a] is not. {TRUE}[U||V]{a=2} is not necessarily meaningful since {a=2} is not necessarily "observable" (if a is local to V, it is not defined outside the parallel command). On the other hand, if the interface is included then we do indeed have a closed system -- the interface is an abstract representative of the physical printer -- and {printed value is 2} is proveable within the framework. While this issue is more philosophical than technical, it does, nevertheless, point out the strong departure from the Apt position.

V PORTS AS REPRESENTATION OF THE INTERFACE

A. GENERAL DESCRIPTION OF PORTS

This chapter develops a representation of the abstract interface as described earlier. We establish the notion of a Port and derive formal semantics that are consistent with modularity requirements and with the Hoare axiomatic verification methodology for partial correctness. 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. First, however, we shall characterize the Port concept heuristically.

The modified CSP language that will be defined is termed Port-CSP. Interprocess communication in Port-CSP 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. (They replace what in standard CSP are P!y and Q?x, respectively). Furthermore, communicating processes need not be "constituent processes of a parallel command."

A Port may be used for communication by more than two processes. Such a port is termed a multi-Port, and is defined as a Port used by more than one process for input, or more than one process for output. A Port that is not a multi-Port is described as simple.

Ports are not referenced indiscriminantly throughout the program text. Rather, like data objects, they possess scope and require proper declaration. This is elucidated further in the section on syntax.

The Port, as the term is used here, differs considerably from

constructs of the same name discussed in the literature. Hoare [78] himself 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 [79], 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 [80]'s Port, termed Communication Port, is a much weightier construct suitable for higher level communications. In this exposition we retain the essence of the original communication semantics inherent in CSP.

B. THE PORT AS A PARALLEL-PROGRAMMING LANGUAGE FEATURE

1. SYNTAX

This section develops the notion of Port as a programming language feature from the syntactical point of view. The BNF description of standard CSP is expanded and modified to include Port declarations and Port-directed I/O. Braces {} have been introduced into BNF to denote none or more repetitions of the enclosed text.

First, consider the input and output commands. As stated previously all interprocess communication takes place through Ports, so these commands are redefined as follows:

<input command>::=<port name>?<target variable>

<output command>::=<port name>!<expression>

<port name>::=<identifier>|<identifier>(<subscripts>) ,

where <target variable>, <expression>, and <subscripts> are defined as in the Appendix.

Just as CSP allows arrays of processes to be declared, so we allow arrays of Ports to be declared. Hence Port names may be subscripted. The subscripts must be run-time constants, however. This is in keeping with the original spirit of CSP in which process subscripts are also run-time constant. At any rate, Port names subscripted with run-time variables can always be simulated with constant subscripts and the alternative command. To illustrate let i be a program variable and consider the input command A(i)?x. If the command does not appear in a guard, then it is semantically equivalent to the alternative command

[i=j_1 -> A(j_1)?x

□ i=j_2 -> A(j_2)?x

. . .

. . .

. . .

□ i=j_n -> A(j_n)?x

□ i≠j_1∧i≠j_2∧ ... ∧i≠j_n->FAIL],

where j_1, ... ,j_n are constants and Ports A(j_1), ... ,A(j_n) have been properly declared. In the case A(i)?x is found in a guard, b;A(i)?x->S, the entire guarded command is semantically equivalent to the collection of guarded commands

(b∧i=j_1);A(j_1)?x -> S

□ (b∧i=j_2);A(j_2)?x -> S

. . .

. . .

. . .

□ (b∧i=j_n);A(j_n)?x -> S ,

where the j_k's are constants as above. Limiting the subscripts to execution-time constants makes the proof rules - to be defined further on - more concise and certainly easier to apply.

Since the scope of a Port will be defined semantically to correspond to a parallel command, it makes sense to associate Port declarations with parallel commands directly. The parallel command is redefined thus:

<parallel command>::={<port declaration>;}[<process>{||<process>}].

A Port declaration is prefixed to the parallel command over which the Port ranges.

The syntax of Port declarations is based on a Pascal-like notation that derives from the examples used by Hoare in describing CSP. As an example of a declaration in this notation, program variable lineimage, an array of 125 elements of type character, is declared

lineimage:(1..125)character;

The only notable difference between this syntax and Pascal is the absence of keywords such as VAR, TYPE, and ARRAY.

A Port declaration specifies the following items: the name (i.e. ID) of the Port, subscript ranges in the case that an array of Ports is declared, the type of Port (static or dynamic), and the type of data, called port-variable type, that passes through the Port during transactions. More than one port may be declared by a single declaration, but all must be of the same Port type and have the same port-variable type. The Port declaration is defined

<port declaration>::=<port id>{,<port id>}:PORT(<port type>)

<port id>::=<identifier>|<identifier>(<label subscript>{,<label subscript>})

<port type>::=STATIC,<port-variable type>|DYNAMIC,<port-variable type>

<port-variable type>::=<type>|<constructor>(<type list>)

<type list>::=<empty>|<type>{,<type>} .

<label subscript> is defined as for process labels (see Appendix) and <type> is Pascal-like as discussed previously.

2. EXAMPLES OF PORT DECLARATION AND I/O COMMAND SYNTAX

1. A:PORT(STATIC,real);

   Declare Port A to be of type STATIC with port-variable type real.

2. A(i:1..100),B:PORT(DYNAMIC,integer);

   Declare a Port B and an array A of 100 Ports, all of type DYNAMIC, all of which transfer data of type integer. This is equivalent to A(1),A(2), ... ,A(100),B:PORT(DYNAMIC,integer);

3. B,C:PORT(DYNAMIC,cons(integer,boolean));

   Declare B and C to be DYNAMIC Ports that transfer data of the form cons(integer,boolean).

4. x:real;A?(x)

   Input through Port A into variable x, where A is declared as in 1 above.

5. i:integer;i:=10;A(7)!i

   Output the value of i through Port A(7), where A(7) is declared as in 2 above.

6. i:integer;b:boolean;C?cons(i,b)

   Input through Port C into cons(i,b), where C is declared as in 3 above. cons(i,b) is a <structured target> as defined by the BNF in the Appendix.

3. SEMANTICS

As stated previously, a Port declaration is always prefixed to a parallel command, and a parallel command delimits the scope of Ports whose declarations are prefixed to it. We call the collection of Ports whose scope is the same parallel command the Port set of the parallel command. We now introduce semantic restrictions that force Port declarations to be made in a manner that is consistent with hierarchical structuring of processes.

All interprocess communicaion occurs through properly declared Ports. Communicating processes need not be "constituent process of a parallel comand," but they must be contained, directly or indirectly, in constituent processes of a parallel command. Furthermore a Port can be used only for communication that crosses a parallel boundary of the parallel command to which the declaration is prefixed. Nested parallel commands must use Ports declared locally for local communication that crosses only nested parallel boundaries.

To illustrate, consider

```
A,B:PORT(STATIC,integer);
[P||Q::C:PORT(STATIC,integer);[R||S]]     .
```

The Port set of the outer parallel command consists of A and B; the Port set of [R||S] consists solely of C. Communciation between P and R (or S) must proceed through Port A or B, while communication between R and S may proceed only through Port C.

A second restriction asserts that a Port name must be unique - i.e. unambiguous - within its scope. In short there are never Port

name conflicts. Had this restriction not been introduced, we would have been forced to define the syntax of the input and output commands in a manner that is reminiscent of monitor calls (see Hoare [74]). Port sets would require names and an input command would take the form

<port set name>.<port name>?<target variable>

in order to resolve port name conflicts. Better to introduce a minor restriction than to muddle syntax.

Transactions across Ports are characterized as follows. An I/O command through Port A takes the form A?x or A!y, where x and y are of type <port-variable type> as given in the Port declaration (see previous section on syntax). As in standard CSP a process issuing an I/O command naming A is delayed (blocked) until another process issues a corresponding (matching) I/O command also naming A. The effect of synchronization of matching commands is the assignment of the value of the output expression to the input target variable. Transactions across a Port proceed serially, one at a time, so that a process issuing an I/O command is delayed until a current transaction across the Port has completed. (This is really an implementation concern since we assume, from the language point of view, that transactions are atomic actions). Furthermore when several I/O commands naming a Port are executed simultaneously, they are nondeterministically paried for synchronization. The scheduling of Port service is fair, however: no process will be passed over in favor of others indefinitely. Each process of a synchronizing pair does not know the identity of the other.

Ports are declared to be either of type <u>STATIC</u> or <u>DYNAMIC</u>. A

STATIC Port remains perpetually open to transactions and is not affected by the control state of processes using it. A DYNAMIC Port, on the other hand, can become closed to subsequent transactions, depending upon the control state of user processes, as described shortly. I/O in standard CSP corresponds to DYNAMIC Port I/O.

An I/O command naming a STATIC Port can never FAIL, nor can it evaluate to FALSE in a guard. A DYNAMIC Port A will lead to an evaluation of FALSE - or fail if not in a guard - if any of the following conditions hold:

1) All processes using A for input have terminated;

2) All processes using A for output have terminated;

3) All but one of the processes using A have terminated.

Cases 1 and 2 are clear: transactions through A are no longer possible. Case 3 covers the situation in which only one of the processes referencing A has not yet terminated and that process uses A for both input and output. Again, transactions are no longer possible.

STATIC ports have been introduced for two reasons: 1) the full power of a dynamic port is in many instances unnecessary, and 2) STATIC ports allow strong postassertions upon termination of loops. To illustrate the second point consider the repetitive command *[b;A?x->S]. If A is DYNAMIC one can not conclude (¬b) upon loop termination, since the loop will terminate if A becomes closed and causes the input command in the guard to evaluate to FALSE. If A is STATIC, however, one does conclude (¬b) as a postcondition. This will be formalized later in the section on Formal Semantics. Note that if A

is STATIC, *[A?x->S] never terminates. STATIC Ports are clearly cheaper to implement and hence we choose them whenever possible.

As in standard CSP, a parallel command terminates when all the constituent processes have terminated. Not all programs are designed to termiate, however. The kinds of programs written in Port-CSP generally fall into one of two relative distinct categories, termed "parallel computations" and "parallel systems." Parallel computations are parallel programs designed to perform a particular computation and then terminate. They are parallel versions of non-real-time programs. A concurrent program that generates all prime numbers less than 100000 is an example of this sort.

A parallel system, on the other hand, is a collection of processes whose termination is either uninteresting -- one is not interested in the postcondition -- or not guaranteed by design. In either case, the program is usually thought to "run forever." These are generally real-time programs. A distributed operating system, or subsystem thereof, is an example of this sort of program. The power of Ports finds better expression in parallel systems than in parallel computations, so the examples below are from among the former.

4. EXAMPLES OF PORT-CSP PROGRAMS

1. Integer Semaphore:

The integer semaphore in standard CSP, shown previously, is implemented using two Ports. The full program is given by

```
P,V:PORT(DYNAMIC,());[S||X(i:1..100)],
```

where S is

```
S::val:integer;val:=initial value of semaphore;
    *[val>0;P?()->val:=val-1
    □V?()->val:=val+1]     .
```

For each i, process X(i) issues the command P!() to perform a P operation on semaphore S, and issues the command V!() to perform a V operation on S. The semaphore process S terminates only when all processes of array X have terminated. If Port V is declared STATIC, S will not terminate; if only Port P is declared STATIC, S may or may not terminate, depending on the final value of val.

2. Array of Semaphores:

We define an array of 50 semaphores, each element of the array constituting an individual process. The program is given by

```
P(i:1..50),V(i:1..50):PORT(DYNAMIC,());[S(i:1..50)||X(i:1..100)],
```

where S(i) is

```
S(i)::val:integer;val:=initial value of semaphore(i);
    *[val>0;P(i)?()->val:=val-1
    □V(i)?()->val:=val+1].
```

For each i and j, process X(i) issues the command P(j)!() to perform a P operation on semaphore S(j), and issues the command V(j)!() to

perform a V operation on S(j).

3. Producer-Consumers:

Consider a producer process P and 100 identical consumer processes C. Once again we use this example as a contrast to the one in standard CSP shown previously. The program is described by

```
A:PORT(DYNAMIC,real);[P||C(i:1..100)]     ,
```

where P and C are given by

```
P::x:real;*[produce(x)->A!x]
C::x:real;*[A?x->consume(x)]     .
```

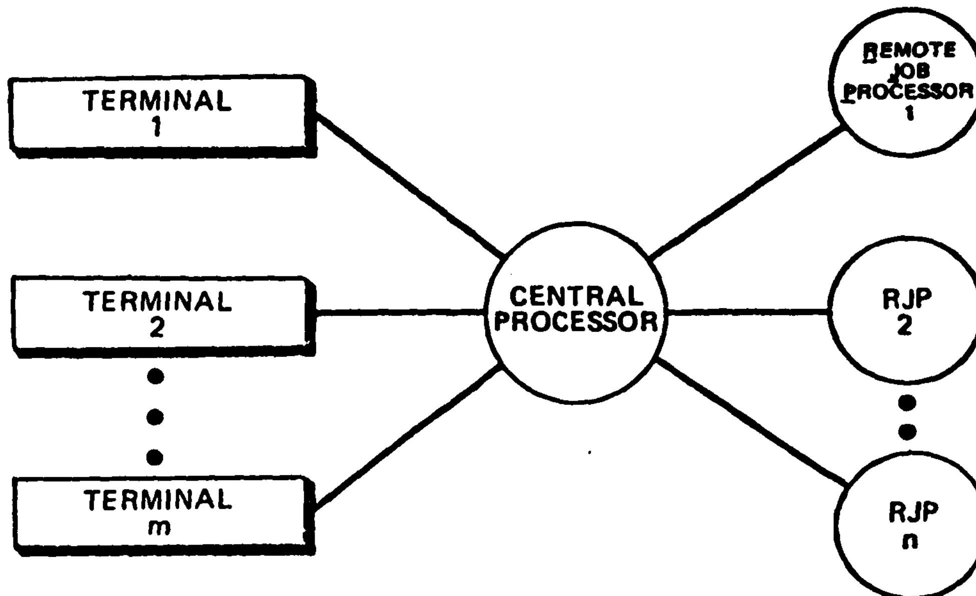
If producer P is assumed to produce indefinitely, then Port A can be declared STATIC. Most parallel systems - as opposed to parallel computations - fall into a producer-consumer paradigm. This is exemplified by the next example.

4. Distributed Operating System:

This example demonstrates the power of hierarchical structure that can be achieved with Ports. Consider the network of processors and terminals shown below. M identical terminals are connected to a central processor, to which are attached n identical remote job processors (RJP's). The terminals independently generate jobs that are routed by the central processor to a free RJP. Any RJP can run any job. The RJP runs the job, computes results, and then ships the results back to the central processor. The central processor in turn

routes the results to the originating terminal. The central processor is expected to provide some level of buffering between terminals and RJP's.

Each terminal consists of a keyboard and a display unit that operate as independent devices for input and output respectively.



Terminal i's operation is modeled by the parallel command

TERMINAL(i)::[*[char:character;DISPLAY(i)?char -> display char on screen]||

*[char:character;get char from keyboard -> KEYBOARD(i)!char]] ,

where Ports DISPLAY(i) and KEYBOARD(i) are used by terminal i for I/O. The displaying process is a consumer; the keyboard process is a producer.

We shall describe the system in top-down fashion. The full

system is given by the following parallel command, which reflects the network topology exactly.

```
DISPLAY(i:1..m),KEYBOARD(i:1..m):PORT(STATIC,character);
JOBRUN:PORT(STATIC,(integer,(1..maxinput)character));
JOBFIN:PORT(STATIC,(integer,(1..maxoutput)character));
[TERMINAL(i:1..m)||CENTRAL PROCESSOR||RJP(i:1..n)]
```

Ports DISPLAY and KEYBOARD have already been described. Port JOBRUN is the interface between the central processor and the RJP's in the direction of jobs flowing toward the RJP's. maxinput is a constant specifying the maximum number of characters that constitute a job. JOBFIN is the interface between the central processor and the RJP's for job results flowing back toward the terminals. maxoutput specifies the maximum size of job results. The integer component of the port variable type associated with JOBRUN and JOBFIN is the terminal number of the corresponding job and job results, respectively.

The RJP(i) are identical for all i, and are given by

```
RJP::term:integer;job:(1..maxinput)character;
      results(1..maxoutput)character;
*[JOBRUN?(term,job)->execute(job,results);JOBFIN!(term,results)]     .
```

The central processor is more complex than both the terminals and the RJP's in that it has a two-sided interface. It is written

```
CENTRAL PROCESSOR::
```

```
JOBNEW:PORT(STATIC,(integer,(1..maxinput)character));
DRIVER(i:1..m):PORT(STATIC,((1..maxoutput)character));
[TERMHANDLER||RJPHANDLER]     .
```

TERMHANDLER interfaces to the terminals and RJPHANDLER interfaces to the RJP'S. JOBNEW is used to send jobs from TERMHANDLER to RJPHANDLER; DRIVER(i) is used for job results moving in the opposite direction.

TERMHANDLER is broken down into display handlers and keyboard handlers:

```
TERMHANDLER::[TERMOUT(i:1..m)||TERMIN(i:1..m)]     .
```

TERMIN(i) handles the keyboard of terminal i:

```
TERMIN(i)::job(1..maxinput)character;p:integer;p:=1;char:character;
*[KEYBOARD(i)?char -> job(p):=char;p:=p+1;
                      [char=EOJ -> JOBNEW!(i,job);p:=1
                      []char≠EOJ -> SKIP]]     .
```

Characters are input from the keyboard and stored in array job until an end-of-job (EOJ) appears, at which time the job is output to RJPHANDLER for processing. Input from the keyboard will still be accepted even though the terminal has a job outstanding, the results of which have not been returned to the display. In fact several jobs may be initiated from a terminal before any results have returned.

TERMOUT(i) handles the display of terminal i:

```
TERMOUT(i)::results:(1..maxoutput)character;
*[DRIVER(i)?results -> proceed:boolean;proceed:=TRUE;
                       p:integer;p:=1;
        *[proceed -> DISPLAY(i)!results(p);
                     [results(p)=EOT -> proceed:=FALSE
                     []results(p)≠EOT -> p:=p+1]
        ]
 ]
```

Whenever job results are received from RJPHANDLER, they are output to the display, character by character, until an end-of-transmission (EOT) appears.

RJPHANDLER is the process that buffers jobs and results in the central processor:

RJPHANDLER::[RJPIN(i:1..k)||RJPOUT(i:1..l)] ,

where RJPIN(i) and RJPOUT(i) are identical for each i. (This is not to say that RJPIN is identical to RJPOUT). k and l are constants that depend on the level of buffering desired. Large l means that many incoming jobs can be buffered when all RJP's are busy; large k means that many job results can be buffered when a terminal issues jobs that are processed and returned faster than the display can handle. RJPIN and RJPOUT are given by

```
RJPIN::term:integer;results:(1..maxoutput)character;
        *[JOBFIN?(term,results) -> [(j:1..m)term=j ->
                                              DRIVER(j)!results]
        ]
```

```
RJPOUT::term:integer;job:(1..maxinput)character;
        *[JOBNEW?(term,job) -> JOBRUN!(term,job)]     .
```

RJPOUT is a simple buffer process; RJPIN is a buffer process for the entire collection of terminals.

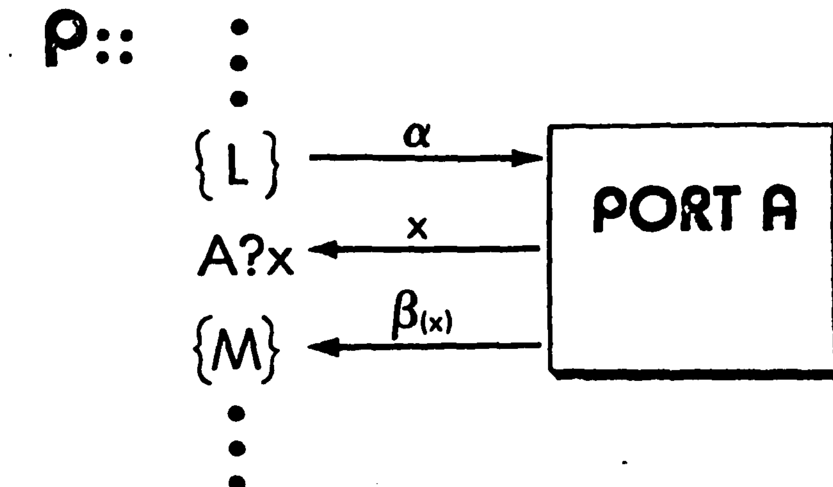
Although many details have been omitted -- how "execute(job,results)" is implemented in Port-CSP -- the overall dynamics of the system have been successfully captured.

C. FORMAL PORT SEMANTICS

1. GENERAL DESCRIPTION

Ports have been introduced as the mechanism used for specifying and developing systems in a modular fashion. In the distributed operating system shown above, we saw how additional buffer processes could be added to the system without changing a single program statement. Our notion of modularity, however, requires that any system be Hoare-verifiable in a modular fashion as well. To this end we shall describe the formal semantics of the Port. We begin by describing simple Ports and how they interact with the environment. Multi-Port semantics are generalizations of the simple case; therefore the following discussion pertains to them as well.

First, consider a process P that inputs through simple Port A into variable x, as shown below.



L and M are Hoare-like pre- and postassertions associated with the I/O command A?x in a local proof of process P. If L holds prior to

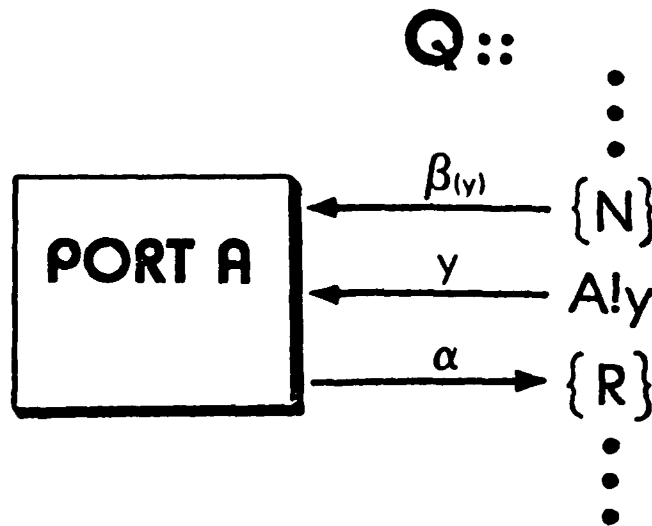
execution of the command, and the command completes execution successfully, then M is guaranteed to hold upon completion. As shown in the diagram above, the Port supplies two semantic items to the process P, and P supplies one item to the Port A.

The Port supplies P with the value to be input into variable x, and an assertion $\beta(x)$. $\beta(x)$ aids L in establishing the postcondition M. In the simplest case we have that $(L \wedge \beta(x))$ implies M. Consider, for instance, a typical simple $\beta(x)$: $x=5$; we can write $\{TRUE\}A?x\{x=5\}$ if we take $L \equiv TRUE$ and $M = \beta(x)$. Note that this triple holds against the Port A, and not just against any one process in particular.

Process P assumes $\beta(x)$ to be true at this point of execution because Port A guarantees that it will hold whenever I/O through A occurs. The Port is able to make this guarantee by requiring that the output process guarantee $\beta(x)$ just prior to execution of its output command. Thus the Port is a logical conduit through which pass assertions that proceed from a guarantor process toward a utilizer process.

In turn process P must guarantee an assertion α that is used by the outputting process to establish its postcondition. Formally, we require that L imply α. There is no flow of data in this direction and hence we write α without a data argument.

The case of the outputting process appears diagramatically as the dual of the one above.



Process Q provides the Port with data value y and guarantees the assertion $\beta(y)$. α assists N in establishing the postcondition R. In simple terms again, $(N \wedge \alpha)$ implies R, and N implies $\beta(y)$.

The Port, then, allows the semantic environments - as well as the execution environments - of the individual processes to interact without their referring to one another explicitly. The Port itself acts as a semantic buffer. If we remove this semantic buffer for a moment, we arrive at a formula closely resembling the Apt communication axiom. This is developed as follows.

Assume x is not a free variable in L. We have thus far characterized I/O by the following four relations:

$$(L \wedge \beta(x)) \Rightarrow M \qquad (1)$$

$$(N \wedge \alpha) \Rightarrow R \qquad (2)$$

$$L \Rightarrow \alpha \qquad (3)$$

$$N \Rightarrow \beta(y) \qquad (4)$$

Since y does not appear free in L or $\beta(x)$ (processes are disjoint in CSP as well as in Port-CSP), from (1) we get

$$(L \wedge \beta(y) \wedge (x=y)) \Rightarrow M \quad . \tag{5}$$

(5) and (4) yield the relation

$$(L \wedge N \wedge (x=y)) \Rightarrow M \quad . \tag{6}$$

Similarly, from (2) and (3) we get

$$(L \wedge N) \Rightarrow R \quad . \tag{7}$$

Combining (6) and (7) to form

$$(L \wedge N \wedge (x=y)) \Rightarrow (M \wedge R) \quad , \tag{8}$$

we see that since (8) holds for all valid postconditions M and R, $(L \wedge N \wedge (x=y))$ is an expression for the strongest combined postcondition.

From Apt's axiom of communication, only formulas of the form $\{r\}P?x \| Q!y\{(x=y) \wedge r\}$ can be derived. Taking $r = (L \wedge N)$, the connection is clear.

Conceptually, $\beta(x)$ represents data and control information flowing from the output process to the input process, while α represents control information only, flowing in the opposite direction. Both α and $\beta(x)$ are associated with the Port rather than with individual processes. A Port is characterized by, among other things, its associated α and β. Individual processes use the α's and β's in their local proofs as previously described. A process that contains more than one input command (or output command) through a particular Post A always uses the same α and β whenever dealing with A.

To generalize, we have established thus far two disjoint semantic domains: the local process environment domain and the interface, or Port set, domain. The interface domain consists of the Port set

endowed with a separate data base (described below). The two domains interact mainly via assertions α and β. At the time of interprocess communication, the Port captures data flowing across an 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.

2. THE INTERFACE DOMAIN

As stated in the previous section, the interface domain consists of a collection of Ports, the Port set, and an entirely separate data base, separate in that it is totally disjoint from the set of variables used by the individual processes. It is separate also in that there exist separate program statements (assignment statements) lying outside the realm of the processes whose sole purpose is to maintain the state of this data base.

At the least, the interface data base consists of variables, called Port variables, that remember the data value of the last transaction through each Port. Each time a transaction occurs through a Port, the associated Port variable is updated with the value of the data passing through the Port.

Usually the data base possesses other variables in addition to the Port variables just mentioned. These are known as auxiliary variables and play a role that is analogous to the role of auxiliary variables in the Owicki and Apt systems. Auxiliary variables are freely chosen since they have no affect whatsoever on the state of execution from the point of view of the processes. Their sole function is to record the state of events from an interprocess point of view.

The separate program statements that update auxiliary variables during a transaction are known as the "Port action" and consist only of assignment statements to auxiliary variables. Modularity dictates that the interface maintain its own independent and self-contained view of the ongoing computation; hence the Port action may refer only to variables of the interface data base. Mentioning process variables is strictly forbidden. Now whenever a transaction occurs through some

Port of the Port set, in addition to the Port variable update mentioned previously, the associated Port action is executed.

Port transactions, developed thus far, involve several separate operations: capturing the output data, updating the Port variable, passing the output data to the input process, and executing the Port action. We always consider the entire transaction as an uninterruptible, atomic operation. Once begun, a Port transaction completes without external interference. This is analogous to Apt's bracketed section, which is considered as an atomic program segment even though it spans two processes and contains concurrently executing statements. Although concurrent transactions through several Ports of the Port set may be logically possible, our interpretation always takes the view that they proceed serially, the order of execution being left undetermined. The formal description of the interface semantics given in the next section places a prohibition on the sharing of auxiliary variables among such Ports over which concurrent transactions are logically possible. Without such a restriction, the value of auxiliary variables might be left undetermined (i.e., ambiguous).

The Apt bracketed section together with the global invariant has counterparts in the Port. The typical bracketed section and global invariant GI is described generically by

```
Q::        .                              P::        .
           .                                         .
           .                                         .
           <P?x;i:=i+1>                              <Q!y;j:=j+1>
           .                                         .
           .                                         .
           .                                         .
```

where i and j are auxiliary variables and GI implies $i=j$. i and j are meant to track one another and the global invariant asserts that outside of the bracketed section they always agree. In the Port we have eliminated GI and j, and allowed both P and Q access to i. $i:=i+1$ would be the Port action. Other Ports of the Port set have access to i through α and β.

As stated at the end of the previous section, the local process environment domain and the interface domain interact primarily via the α's and β's. Modularity requirements prohibit the interface from "knowing" anything in particular about process variables, but the converse is not true. Processes need access to interface variables in establishing proof asserions, though they may not alter these variables that belong to another domain. This is a counterpart to Apt's global invariant, which is here nonexistent. α and β, which are logical formulas in the variables of the interface, allow the introduction of interface variables into the local process proof in places where they are not updated by the Port action. β captures the view of the interface as seen by the input process and guaranteed by the output process, while α captures the view of the interface as seen by the output process and guaranteed by the input process. The formal description of Port semantics in the next section establishes what has been hitherto described informally.

3. FORMAL DESCRIPTION

In this section and the following one we develop the formal semantics of the simple (non-multi) Port as described heuristically in previous sections. The aim here is to specify the nature of assertions α and β, to indicate how they relate to the local process environment in terms of the Hoare axiomatic proof methodology, and to maintain modularity throughout. Modularity means that a change to one process should require only reverification of that one process against an established interface. First we describe those elements that formally constitute the Port; the following section provides axioms that constitute the proof methodology.

A simple Port is formally described as follows.

1. An I/O command through Port A takes the form A?x or A!y (input and output, respectively), where A is the Port name. Port name corresponds to <Port name> in the BNF description of Port-CSP given earlier.
2. Associated with Port A is an abstract data object "a" (lower case letter whose upper case is the Port name) of data type <port-variable type> as given in the declaration of A (see BNF description of Port-CSP for a syntactic description of this nonterminal). a is known as the Port variable. Port variables lie in a name space that is disjoint from the individual process environments; therefore, processes in the scope of A are not permitted to mention a in program statements (conflicts can be resolved by renaming variables or Ports). Synchronization of A?x and A!y is interpreted as atomic execution of the sequence

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

3. Associated with Port A is a <u>Port action</u> S_A. S_A consists of assignment statements to auxiliary variables. Auxiliary variables, like Port variables, are confined to the interface domain and therefore may not be mentioned in program statements of processes in the scope of A. An auxiliary variable is associated with a parallel command and hence is declared in the annotated program text (i.e., program + proof) where the Port set is declared. 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

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

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

4. Auxiliary variables and Port variables associated with a simple Port are in no sense global. Each safely tracks the interprocess state of at most <u>two</u> processes. Consequently, we limit their scope as follows.

For Port variable a, process P <u>owns</u> a if either of the following holds:

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 <u>owns</u> v if P names Port A in an I/O command and S_A mentions v. We require that the following condition hold: each interface variable may be owned by no more than two processes (this constraint is referenced in the axioms). Thus a pair of processes associates naturally with a subset of

Ports and with the subset of auxiliary and port variables jointly owned by them.

5. Associated with Port A is an <u>assertion $\beta_A(a)$</u>. $\beta_A(a)$ is confined to Port space, and therefore mentions as free variables only auxiliary and Port variables of the Port set containing A. Moreover, the two processes using Port A -- A is simple, so there are only two -- must own all variables of $\beta_A(a)$.

6. Associated with Port A is an <u>assertion α_A</u>. α_A is constrained to mention the same kind of variables as found in $\beta_A(a)$. The two processes using A must own all free variables of α_A.

4. AXIOMS

We extend the Hoare axiomatic proof scheme for proof of partial correctness of sequential programs to include concurrent programs written in Port-CSP. We assume here that all Ports are simple; multi-Port axioms are taken up in a subsequent section.

The axioms specify precisely how the proof of a local process interacts with a previously established interface, i.e., with a predefined Port set complete with ancillary items: port variables, auxiliary variables, Port actions, and α's and β's. The ancillary items are required only for the purposes of verification and need not be specified (or may be eliminated or ignored) when verification is of no interest. Once partial correctness has been established, they may be ignored until changes to the program require reverification.

Verification of a local process is always accomplished against an estabished interface. This is in strict contrast to the Apt system in which all process are verified simultaneously. Verifying a process against an interface as is done here has the effect of validating the process against an entire class of processes, infinite in number. The axiom for parallel composition allows the individual process proofs to be combined when the individual processes themselves are combined into a parallel system. A system-wide pre- and postcondition is then established.

Local process variables and variables of the Port set data base (port variables and auxiliary variables) are the only variables allowed to appear free in proof assertions of a local process. Furthermore, proof assertions may not mention process variables changed by other processes. The axioms defining the proof methodology for partial

correctness follow.

1. Hoare's axioms, composition rules, and rules of inference for sequential programs, both deterministic and nondeterministic, as defined by Dijkstra [76] in terms of wp and wlp (weakest precondition and weakest liberal precondition, respectively) and as described by de Bakker [80] in terms of sp and slp (strongest postcondition and strongest liberal postcondition, respectively). These have been described earlier.

2. Axiom for Parallel Composition

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

where each Port and auxiliary variable appears free in the pre- and /or postcondition of only those 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 provides for the construction of a proof of a parallel program from proofs of the individual processes.

3. Axioms of Communication

$S_A(a)$ is the Port action declared for Port A.

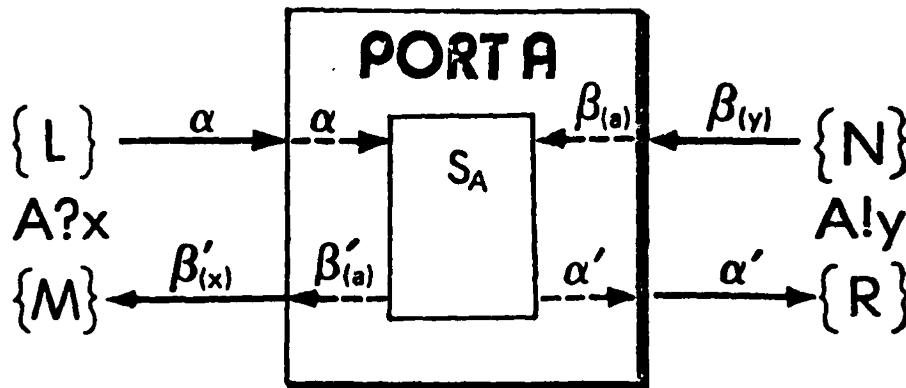
(i) $wlp(A?x, M(x)) = \forall a[(\beta_A(a) \Rightarrow wp(S_A(a), M(a))) \wedge \alpha_A]$

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

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 figure below 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. <u>Axiom of Substitution</u>

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

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 <u>no more than once per auxiliary variable</u>.

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

5. <u>Axiom for Alternative Command</u>

$$\frac{\prod_{i=1}^{n} [L \Rightarrow [\sum_{k=1}^{n} b_k \wedge (b_i \Rightarrow wlp(G_i;S_i,M))]]}{\{L\}[b_1;G_1 \rightarrow S_1 \Box \ldots \Box b_n;G_n \rightarrow S_n]\{M\}}$$

6. <u>Axiom for Repetitive Command</u>

$$\frac{\prod_{i=1}^{n} [(L \wedge b_i) \Rightarrow wlp(G_i;S_i,L)]}{\{L\}*[b_1;G_1 \rightarrow S_1 \Box \ldots \Box b_n;G_n \rightarrow S_n]\{L \wedge \prod_{i=1}^{n} (b_i \Rightarrow G_i \text{ names } \underline{\text{DYNAMIC}} \text{ Port }\}}$$

L is known as the loop invariant.

5. REMARKS CONCERNING THE AXIOMS OF COMMUNICATION

The axioms of communication relate three groups of formulas to one another: preconditions, postconditions, and α's and β's. The communication axioms specify preconditions as a function of postconditions and α's and β's. But this is only one out of 3 possible ways of relating the three groups. Each can be expressed as a function of the remaining two (if we appeal to an intuitive notion in logic analagous to the implicit function theorem of advanced calculus). Thus, we can also derive weakest α's and β's as a function of preconditions and postconditions, as well as strongest postconditions as a function of preconditions and α's and β's. We begin by developing the former.

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 β and α, respectively.

Then

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

where D(a) is given by

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

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

where E is given by

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

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

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

and process Q contains m output commands naming A,

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

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

$$D(a) = \prod_{i=1}^{n} D_i(a) \quad \text{and} \quad E = \prod_{j=1}^{m} E_j \quad .$$

$$\text{Then } wl\beta_A(a) = \begin{cases} D(a) \text{ if } \prod_{j=1}^{m} (N_j => D(y_j)) = \text{TRUE} \\ \text{FALSE otherwise} \end{cases}$$

and

$$wl\alpha_A = \begin{cases} E \text{ if } \prod_{i=1}^{n} (L_i => E) = \text{TRUE} \\ \text{FALSE otherwise} \end{cases} \quad .$$

Deriving $wl\beta$ and $wl\alpha$ 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. In this case the interface semantics are derived, in a sense, from the composite local environments.

Now let us find the third way of relating the three groups of formulas: strongest postconditions as a function of preconditions and α's and β's. The case of the input command proceeds as follows. If $(L=>\alpha_A) \neq$ TRUE, then slp(L(a,x),A?x) = FALSE; otherwise from (2) (see section 3) we get

$$slp(L(a,x),A?x) =$$

$$sp(\exists z_1[L(z_1,x)\wedge\beta_A(a)],S_A;x:=a),\ z_1 \neq a,\ z_1 \text{ not free in } (L\wedge\beta_A(a))$$

$$= sp(\exists z_1[L(z_1,x)\wedge\beta_A(a)],\ x:=a;S_A)$$

$$= sp(\exists z_1\exists z_2[L(z_1,z_2)\wedge\beta_A(a)\wedge(x=a)],S_A) \quad , \tag{3}$$

$z_1 \neq a$, $z_2 \neq x$, $z_1 \neq z_2$, z_1 and z_2 not free in $(L \wedge \beta_A(a))$.

When x and a are not free in L, (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 $(N \Rightarrow \beta_A(y)) \neq$ TRUE, then $slp(N(a), A!y)$ = FALSE; otherwise

$slp(N(a), A!y) = sp((N(a) \wedge \alpha_A), a{:=}y; S_A)$

$= sp(\exists z[N(z) \wedge \alpha_A \wedge (a=y)], S_A)$, (4)

$z \neq a$, z not free in $(N \wedge \alpha_A)$.

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

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

Our final remarks concern the role of the universal quantifier appearing in Axioms 3(i) and (ii). Without them the axioms appear to be reasonably intuitive; the quantifiers may seem superfluous at first sight. But consider the following program segment where $\beta_A(a) \equiv$ TRUE and $S_A(a)$ = SKIP:

$$A?x; A?z \quad .$$

Since $\beta_A(a)$ tells us nothing about the value of a, we know nothing about the values of x and z and, therefore, we should not be able to conclude $\{x=z\}$ as a postcondition. Indeed we cannot when we use Axiom 3(i) with the quantifier included:

$\{\forall a[\text{TRUE} \Rightarrow a=x]\} A?z \{z=x\}$.

But $\forall a[\text{TRUE} \Rightarrow a=x] = \forall a[a=x]$ = FALSE, so we conclude {FALSE}A?z{z=x}, which in turn results in {FALSE}A?x;A?z{z=x} when we apply the axiom a second time.

Without the quantifier, however, we get, upon applying the axiom

once, {a=x}A?z{z=x}. The second application, using {a=x} as the postcondition, results in {a=a}, or equivalently {TRUE}, as the precondition. Thus, {TRUE}A?x;A?z{z=x} can be established, a triple that is intuitively invalid. This justifies the need for the quantifier.

From this example we see that the role of the quantifier is to limit the scope of the Port variable a. The power of the Port variable is effectively nullified in an unsafe program region by becoming a bound variable. This technique of using a quantifier to remove the power of a free variable was used in arriving at the expressions for $wl\beta$, $wl\alpha$, and the strongest liberal postconditions given earlier. It will be used again throughout the development of multi-Port semantics.

6. EXAMPLES OF PORT-CSP PROOFS

The examples presented in this section are taken from the realm of parallel computations rather than parallel systems. As previously stated, parallel systems do not generally possess interesting postconditions, and consequently their proofs tend to be uninteresting as well. The section on multi-Ports, however, does include examples from the domain of parallel systems.

1. Simple Example:

Consider the following program, which plainly performs not too meaningful a computation, but does illustrate the basic proof technique. The program is given by

A,B:PORT(STATIC,integer);[P||Q]

where P and Q are

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

We wish to verify that upon termination x=z, i.e., that the triple {TRUE}[P||Q]{x=z} is valid (we omit the Port declaration for the sake of brevity). First, we specify the ancillary Port components:

	A	B
ACTION	SKIP	SKIP
α	TRUE	TRUE
β	TRUE	b=a

The only nontrivial component is $\beta_B(b)$, which ensures for the input process P that the value of the arriving data, b, is identical to the value of the data that last passed through Port A. The proof proceeds in three steps:

a) establishing {TRUE}P{z=x} against the Port set;

b) establishing {TRUE}Q{TRUE} against the Port set;

c) using the axiom for parallel composition to conclude that {TRUE}[P||Q]{x=z} is valid from steps a) and b).

step a) The annotated version of P is

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

We proceed from the bottom of the annotated text and work backward, proving each triple.

<u>proof of {a=x}B?z{z=x}</u>: To establish this we must show that the precondition, a=x, implies the weakest liberal precondition as given by Axiom 3(i);

$wlp(B?z, z=x) = \forall b[(\beta_B(b) \Rightarrow wp(S_B(b), b=x)) \wedge \alpha_B]$

$= \forall b[((b=a) \Rightarrow wp(SKIP, b=x)) \wedge TRUE]$

$= \forall b[(b=a) \Rightarrow b=x]$. (1)

Clearly (a=x)=>(1), and the triple is established.

<u>proof of {TRUE}A!x{a=x}</u>: The weakest liberal precondition of A!x{a=x} is given by Axiom 3(ii):

$wlp(A!x, a=x) = \forall a[(TRUE \Rightarrow wp(SKIP, x=x)) \wedge \beta_A(x)]$

$= \forall a[(TRUE \Rightarrow wp(SKIP, TRUE)) \wedge TRUE]$

$= \forall a[TRUE \Rightarrow TRUE]$

= TRUE .

Thus, {TRUE}P{z=x} is valid against the Port set.

step b) The annotated version of Q is

Q:: {TRUE}
A?y
{a=y};
B!y
{TRUE} .

Once again we proceed from the bottom.

proof of {a=y}B!y{TRUE}: Axiom 3(ii) gives

$wlp(B!y,TRUE) = \forall b[(\alpha_B=>wp(SKIP,TRUE))\wedge\beta_B(y)]$

$= \forall b[(TRUE=>TRUE)\wedge(y=a)]$

$= \forall b[y=a]$

= y=a, which is equivalent to the precondition.

proof of {TRUE}A?y{a=y}: Axiom 3(i) gives

$wlp(A?y,a=y) = \forall a[(\beta_A(a)=>wp(SKIP,a=a))\wedge\alpha_A]$

$= \forall a[(TRUE=>wp(SKIP,TRUE)\wedge TRUE]$

$= \forall a[TRUE=>TRUE]$

= TRUE .

Thus, {TRUE}Q{TRUE} is valid against the Port set.

step c) The axiom for parallel composition, Axiom 2, is applied to establish the final result, as follows. Axiom 2 tells us

$$\frac{\{TRUE\}P\{z=x\} \text{ and } \{TRUE\}Q\{TRUE\}}{\{TRUE\}[P||Q]\{z=x\}} \quad , \qquad (2)$$

provided there are no conflicts of ownership of Port variables. But there are only two process, P and Q, so there can be no problems of ownership. Steps a) and b) established the antecedent formulas (those above the line) of (2); thus the consequent, the formula below the line, is established. Q.E.D.

Using a different version of Q that maintains the same interface, for example the one shown below, would require reperforming only step b); steps a) and c) would still be valid.

```
Q::     A?y;
        w:=y;
        B!w
```

The annotated text of this Q, which is very similar to the proof of the former one, is given by

```
        {TRUE}
        A?y
        {a=y};
        w:=y
        {a=w};
        B!w
        {TRUE}    .
```

Thus, local changes to Q require only reverification of the local process Q.

2. Dijkstra's Exponentiation Problem:

The following is a distributed version of Dijkstra's exponentiation problem (Dijkstra [76]) 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

```
A:PORT(DYNAMIC,integer);
B:PORT(STATIC,yes());
C:PORT(STATIC,integer);
[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]
```

Without Q and the inner loop of P, this program is the usual sequential program for computing exponentiation. P's inner loop speeds up the computation. The parallelism permits concurrent squaring of v with halving of x. We shall prove the validity of

$$\{v^u=k\}[P||Q]\{z=k\wedge u=0\}. \qquad (3)$$

As in example 1, we omit the Port declarations for brevity.

The proof proceeds in three steps:

a) establishing $\{v^u=k\}P\{z=k\wedge u=0\}$ against the Port set;

b) establishing {TRUE}Q{TRUE} against the Port set;

c) using the axiom for parallel composition to conclude that (3) is valid from steps a) and b).

The Ports are defined as follows; i is an auxiliary variable of type integer:

	A	B	C
ACTION	i:=0	i:=i+1	SKIP
α	TRUE	TRUE	TRUE
β	TRUE	TRUE	$2^i c=a$

step a) The annotated version of P is

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

$z:=1$

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

$*[\{zv^u=k\}u\neq 0\text{->}\{zv^u=k\}A!u\{zv^u=k\wedge a=u\wedge i=0\};f:=\text{TRUE}\{zv^u=k\wedge a=u\wedge i=0\wedge f\};$

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

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

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

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

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

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

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

<u>inner loop</u>: This loop, which is designed to preserve the relation $zv^u=k$, maintains the loop invariant $(zv^{a2^{-i}}=k)\wedge(\neg f\Rightarrow 2^i u=a)$. From the

annotated text, we see that $zv^u=k$ holds upon loop entry. Since B and C are STATIC Ports, the consequent of Axiom 6, the axiom for the repetitive command, permits us to conclude $\neg f$ upon loop termination, which together with the loop invariant implies $zv^u=k$ upon exit. We shall demonstrate the validity of triples involving the two I/O commands; all other statements are sequential and are handled in the usual manner.

proof of $\{zv^{a2^{-i}}=k\wedge(\neg f \Rightarrow 2^{i+1}u=a)\}$B?yes()$\{zv^{a2^{1-i}}=k\wedge(\neg f \Rightarrow 2^{i}u=a)\}$:

From Axiom 3(i) we have

$$wlp(B?yes(),(zv^{a2^{1-i}}=k\wedge(\neg f \Rightarrow 2^{i}u=a))) =$$

$$[(TRUE \Rightarrow wp(i:=i+1,(zv^{a2^{1-i}}=k\wedge(\neg f \Rightarrow 2^{i}u=a))))\wedge TRUE] \tag{4}$$

$$= wp(i:i+1,(zv^{a2^{1-i}}=k\wedge(\neg f \Rightarrow 2^{i}u=a)))$$

$$= zv^{a2^{-i}}=k \wedge (\neg f \Rightarrow 2^{i+1}u=a) \quad .$$

yes() is a signal involving no transfer of data; hence, the universal quantifier of Axiom 3(i) is vacuous and therefore does not appear in (4).

proof of $\{zv^{a2^{-i}}=k\}$C?u$\{zv^{a2^{-i}}=k\wedge 2^{i}u=a\}$:

From Axiom 3(i) we write

$$wlp(C?u,(zv^{a2^{-i}}=k\wedge 2^{i}u=a)) =$$

$$\forall c[(2^{i}c=a \Rightarrow wp(SKIP,zv^{a2^{-i}}=k\wedge 2^{i}c=a))\wedge TRUE]$$

$$= \forall c[2^{i}c=a \Rightarrow zv^{a2^{-i}}=k\wedge 2^{i}c=a]$$

$$= \forall c[2^{i}c=a \Rightarrow zv^{a2^{-i}}=k] \quad . \tag{5}$$

Clearly $zv^{a2^{-i}}=k$ implies (5).

<u>proof of inner-loop invariant</u>: In order to apply the axiom for the repetitive command, we must satisfy the antecedent of the axiom, namely, we must demonstrate

$$((zv^{a2^{-i}}=k\wedge(\neg f\Rightarrow 2^{i}u=a))\wedge f)\Rightarrow(zv^{a2^{-i}}=k\wedge(\neg f\Rightarrow 2^{i+1}u=a)) \quad (6)$$

and

$$((zv^{a2^{-i}}=k\wedge(\neg f\Rightarrow 2^{i}u=a))\wedge f)\Rightarrow(zv^{a2^{-i}}=k) \quad . \quad (7)$$

(7) is clearly valid; (6) is valid as well when we note that $f\Rightarrow(\neg f\Rightarrow F)$ is valid for any formula F.

<u>outer loop</u>: This loop constitutes the familiar program for computing exponentiation. The loop invariant is $zv^{u}=k$. Upon exit u=0 by the consequent to Axiom 6 (there is no G_i) and thus z=k. We shall demonstrate the validity of the triple involving the I/O command.

<u>proof of $\{zv^{u}=k\}A!u\{zv^{u}=k\wedge a=u\wedge i=0\}$:</u>

From Axiom 3(ii), $wlp(A!u, zv^{u}=k\wedge a=u\wedge i=0)$ =

$\forall a[(TRUE\Rightarrow wp(i:=0, zv^{u}=k\wedge u=u\wedge i=0))\wedge TRUE]$ =

$\forall a[zv^{u}=k]$ = $zv^{u}=k$.

step b) The annotated version of Q is

Q:: {TRUE}

[{TRUE}A?x->{i=0∧a=x}[{2^{i}x=a}even(x)->{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} .

The inner-loop invariant is $2^{i}x=a$; the outer-loop invariant is trivial (TRUE). We will demonstrate that triples involving the I/O commands are valid.

<u>proof of {TRUE}A?x{i=0∧a=x}:</u>

wlp(A?x,i=0∧a=x) = ∀a[(TRUE=>wp(i:=0,i=0∧a=a))∧TRUE] = TRUE.

<u>proof of $\{2^{i}x=a\}$B!yes()$\{2^{i-1}x=a\}$:</u>

wlp(B!yes(),$2^{i-1}x=a$) = [(TRUE=>wp(i:=i+1,$2^{i-1}x=a$))∧TRUE]

= $(2^{i}x=a)$.

<u>proof of $\{2^{i}x=a\}$C!x{TRUE}:</u>

wlp(C!x,TRUE) = ∀c[(TRUE=>wp(SKIP,TRUE))∧$2^{i}x=a$]

= $(2^{i}x=a)$.

Thus, we have shown that {TRUE}Q{TRUE} holds against the Port set.

step c) By a simple application of the axiom for parallel composition, we arrive at the validity of (3). With only two processes there can never be problems of ownership of interface variables. Q.E.D.

A different version of Q is proposed below. Verification of [P||Q] using this version requires only reperforming step b).

```
Q::   *[A?x->n:=0;
          *[even(x/2^n)->n:=n+1];x:=x/2^n;
          *[n>0->B!yes();n:=n-1];C!x]     .
```

3. Partitioning a Set:

This is a variation of a program appearing in Apt [80]. Given two disjoint sets of integers S and T, $S \neq \phi$, $S \cup T$ is to be partitioned into subsets S' and T' such that $|S|=|S'|$, $|T|=|T'|$, and every element of S' is smaller than every element of T'. The program is given by

```
A:PORT(DYNAMIC,integer);B:PORT(STATIC,integer);[P||Q]
```

where P and Q are

```
P::      mx:=max(S);
         A!mx;
         S:=S-{mx};
         B?x;
         S:=S∪{x};
         mx:=max(S);
     *[mx>x->A!mx;S:=S-{mx};B?x;S:=S∪{x};mx:=max(S) ]
Q::  *[A?y->T:=T∪{y};mn:=min(T);B!mn;T:=T-{mn}]     .
```

P and Q exchange the current maximum of S with the current minimum of T until the last integer received by P from Q is the maximum of S.

Let $S_0 = \{s_1, \dots ,s_n\}$ and $T_0 = \{t_1, \dots ,t_m\}$ be constant disjoint sets of integers. Without loss of generality we can assume $s_1>s_2> \dots >s_n$ and $t_1<t_2< \dots <t_m$. We wish to verify

$$\{S=S_0\neq\phi,T=T_0\}[P||Q]\{|S|=|S_0|,|T|=|T_0|,S\cup T=S_0\cup T_0,T\neq\phi \Rightarrow \min(T)>\max(S)\} \quad . \tag{8}$$

For readability we have substituted a comma for the conjunction sign

A. The Ports are declared as follows; i is an auxiliary variable of type integer that counts the number of exchanges between P and Q:

	A	B
ACTION	$i:=i+1$	SKIP
α	TRUE	TRUE
β	$(a=s_{i+1},(i>0\Rightarrow a>b))$	$(b=s_i \vee b=t_i)$.

Let S_{ij} denote the set $S_0-\{s_1, \ldots ,s_i\}\cup\{t_1, \ldots ,t_j\}$ and T_{ij} the set $T_0\cup\{s_1, \ldots ,s_i\}-\{t_1, \ldots ,t_j\}$. The proof proceeds in three steps:

a) establishing $\{i=0,S=S_0\neq\phi\}P\{i>0,max(S)=b,$
$((S=S_{(i-1)(i-1)},b=s_i)\vee(S=S_{ii},b=t_i))\}$ against the Port set;

b) establishing $\{i=0,T=T_0\}Q\{i\geq 0,((i>0,T\neq\phi)\Rightarrow b<min(T)),$
$((T=T_0,i=0)\vee(T=T_{ii},i>0,b=t_i)\vee(T=T_{(i-1)(i-1)},i>0,b=s_i))\}$
against the Port set;

c) Applying the axioms for parallel composition and substitution to conclude that (8) is valid from steps a) and b).

step a) The annotated version of P is

P:: $\{i=0,S=S_0\neq\phi\}$

mx:=max(S)

$\{i=0,mx=s_1,S=S_0\}$;

A!mx

$\{i=1,mx=s_1,S=S_0\}$;

S:=S-{mx}

$\{i=1,mx=s_1,S=S_0-\{s_1\}\}$;

$$B?x$$

$$\{i=1,mx=s_1,S=S_0-\{s_1\},(x=s_1 \vee x=t_1),x=b\};$$

$$S:=S \cup \{x\}$$

$$\{i=1,mx=s_1,S=S_0-\{s_1\} \cup \{x\},(x=s_1 \vee x=t_1),x=b\};$$

$$mx:=max(S)$$

$$\{i=1,mx=max(S),S=S_0-\{s_1\} \cup \{x\},(x=s_1 \vee x=t_1),x=b\};$$

$$*[\{IL_p\}mx>x->\{i>0,mx=max(S),S=S_{ii},mx=s_{i+1},mx>b\}A!mx$$

$$\{i>0,mx=s_i,s_i=max(S),S=S_{(i-1)(i-1)}\};\ S:=S-\{mx\}\{i>0,s_i>max(S),$$

$$S=S_{i(i-1)}\};B?x\{i>0,s_i>max(S),S=S_{i(i-1)},(x=s_i \vee x=t_i),x=b\};$$

$$S:=S \cup \{x\}\{i>0,((S=S_{(i-1)(i-1)},x=s_i) \vee (S=S_{ii},x=t_i)),x=b,$$

$$max(S) \geq x,S \neq \phi,max(S)>x=>(max(S)=s_{i+1},x=t_i)\};$$

$$mx:=max(S)\{IL_p\}] \quad ,$$

where the loop invariant IL_p is given by

$$i>0,((S=S_{(i-1)(i-1)},x=s_i) \vee (S=S_{ii},x=t_i)),x=b,mx \geq x,$$

$$mx=max(S),mx>x=>(mx=s_{i+1},x=t_i) \quad .$$

Since (ILp,mx>x) implies the precondition to A!mx appearing in the loop, the antecedent of Axiom 6 is satisfied and thus ILp is indeed a valid invariant. Upon loop termination, mx=x which together with ILp implies the desired postcondition of step a). We will demonstrate the validity of the triples involving I/O commands in the loop; those preceeding the loop are similar and will be omitted.

<u>proof of $\{i>0,mx=max(S),S=S_{ii},mx=s_{i+1},mx>b\}A!mx$</u>

<u>$\{i>0,mx=s_i,s_i=max(S),S=S_{(i-1)(i-1)}\}$</u>: By Axiom 3(ii),

$wlp(A!mx,(i>0,mx=s_i,s_i=max(S),S=S_{(i-1)(i-1)})) =$

$\forall a[(TRUE=>wp(i:=i+1,(i>0,mx=s_i,s_i=max(S),S=S_{(i-1)(i-1)}))),$

$mx=s_{i+1},(i>0=>mx>b)] =$

$\forall a[i>-1,mx=s_{i+1},s_{i+1}=max(S),S=S_{ii},(i>0=>mx>b)] =$

$$i>-1,mx=s_{i+1},s_{i+1}=max(S),S=S_{ii},(i>0=>mx>b) \quad . \tag{9}$$

The precondition implies (9) and so the triple is established.

<u>proof of $\{i>0,s_i>max(S),S=S_{i(i-1)}\}B?x\{i>0,s_i>max(S),$ $S=S_{i(i-1)},(x=s_i \vee x=t_i),x=b\}$:</u> By Axiom 3(i),

$wlp(B?x,(i>0,s_i>max(S),S=S_{i(i-1)},(x=s_i \vee x=t_i),x=b)) =$

$\forall b[(i>0=>(b=s_i \vee b=t_i))=>wp(SKIP,(i>0,s_i>max(S),$

$S=S_{i(i-1)},(b=s_i \vee b=t_i),b=b))] =$

$$\forall b[(i>0=>(b=s_i \vee b=t_i))=>i>0,s_i>max(S),S=S_{i(i-1)},(b=s_i \vee b=t_i)]. \tag{10}$$

The precondition implies (10) and so the triple is established.

step b) The annotated version of Q is

$Q:: \quad \{i=0,T=T_0\}$

$*[\{IL_Q\}A?y->\{i>0,T=T_{(i-1)(i-1)},y=s_i\}T:=T \cup \{y\}\{i>0,T=T_{i(i-1)}\};$

$mn:=min(T)\{i>0,T=T_{i(i-1)},mn=min(T)\};B!mn\{i>0,T=T_{i(i-1)},$

$mn=min(T),b=mn\};T:=T-\{mn\}\{IL_Q\}]$,

where the loop invariant IL_Q is given by

$i \geq 0,(i>0=>(b=mn,(T \neq \phi=>mn<min(T)))),$

$((T=T_0,i=0) \vee (T=T_{ii},i>0,mn=t_i) \vee (T=T_{(i-1)(i-1)},i>0,mn=s_i)).$

Using the Hoare rule of inference

$$\frac{(i=0,T=T_0)\Rightarrow IL_Q,\ \{IL_Q\}Q\{IL_Q\},\ (IL_Q\Rightarrow \text{step b) postcondition})}{\{i=0,T=T_0\}Q\{\text{step b) postcondition}\}}\quad,$$

we see that we arrive at the desired triple from the consequent once we establish IL_Q as the loop invariant. We shall demonstrate the validity of triples in the annotated text that involve I/O statements.

<u>proof of $\{IL_Q\}A?y\{i>0,T=T_{(i-1)(i-1)},y=s_i\}$:</u>

$wlp(A?y,(i>0,T=T_{(i-1)(i-1)},y=s_i))=$

$\forall a[(a=s_{i+1},(i>0\Rightarrow a>b))\Rightarrow wp(i:=i+1,(i>0,T=T_{(i-1)(i-1)},a=s_i))]$

$=\forall a[(a=s_{i+1},(i>0\Rightarrow a>b))\Rightarrow(i>-1,T=T_{ii},a=s_{i+1})]$

$=\forall a[(a=s_{i+1},(i>0\Rightarrow a>b))\Rightarrow(i>-1,T=T_{ii})]$.

Since $(IL_Q,a=s_{i+1},(i>0\Rightarrow a>b))$ implies $(i>-1,T=T_{ii})$, the triple is established.

<u>proof of $\{i>0,T=T_{i(i-1)},mn=min(T)\}B!mn\{i>0,T=T_{i(i-1)},mn=min(T),b=mn\}$:</u>

$wlp(B!mn,(i>0,T=T_{i(i-1)},mn=min(T),b=mn))=$

$\forall b[(TRUE\Rightarrow wp(SKIP,(i>0,T=T_{i(i-1)},mn=min(T),mn=mn))),$

$(mn=s_i\vee mn=t_i)]=$

$i>0,T=T_{i(i-1)},mn=min(T),(mn=s_i\vee mn=t_i)$,

which is implied by the precondition.

step c) By applying the axiom for parallel composition and the Hoare rule of inference, we arrive at a triple identical to (8) except for the presence of i=0 in the precondition. The axiom of substitution permits the elimination of i by substituting for it the constant 0; (0=0) is identical to TRUE and thus the entire term disappears from the precondition. Q.E.D.

D. MULTI-PORTS

1. GENERAL DESCRIPTION

Up to this point the semantic development of Ports has pertained almost exclusively to simple Ports, i.e., to Ports that are named in I/O commands by no more than a pair of lowest level processes (the term lowest level process refers to a process containing no nested subprocesses). Thus, if Port A is used for communication by P, R, and S in [P||Q::[R||S]], then A cannot be simple and must be a multi-Port.

Since a lowest level process cannot communicate with itself through I/O commands, we can state affirmatively that a multi-Port is defined to be a Port that is used by more than one lowest level process for input, or more than one lowest level process for output. Hence, any Port that is used by three or more processes, not necessarily lowest level, is necessarily a multi-Port. The converse is not true, however, since we allow the use of a Port for both input and output by a single process. As with simple Ports, multi-Ports are either of type STATIC or DYNAMIC.

As an example of a typical multi-Port, consider Port A in the Producer-Consumers system (example 3 in section entitled EXAMPLES OF PORT-CSP PROGRAMS) described earlier. A is used by both P and each of the 100 consumer processes C comprising the program. Similarly, in the Distributed Operating System described above, JOBRUN, JOBFIN, JOBNEW, DRIVER(i) are all multi.

As inferred from the examples just cited, Ports appearing in parallel systems - as opposed to parallel computations - tend to be multi-Ports. Since the power of Ports finds its fullest expression in parallel systems, a complete formal description of multi-Port semantics

is central to the issues addressed by this dissertation. To this end we shall describe the ancillary components of a multi-Port and generalize the axioms presented earlier. First, however, we shall demonstrate the limitations of simple-Port semantics as applied in the general case of mulit-Ports.

The semantics developed for the simple Port are not directly applicable to the more general multi-Port case, the reason being that it is unsafe to share interface variables by more than a pair of processes. Consider processes P, Q, and R sharing multi-Port A as shown.

```
P::     A!1;                Q::   A?x           R::   A?y
        A!2
```

Establishing the intuitively valid triple {TRUE}P{a=2} presents no difficulties when we use the axioms presented earlier. But we can also establish {TRUE}Q{a=x} and {TRUE}R{TRUE}, the first triple being an outright falsehood. From the axiom for parallel composition we would wrongly conclude that {TRUE}[P||Q||R]{x=2} holds. Moreover, we would like to have the ability of establishing $(x=2 \vee y=2)$ as a postcondition, but this is not possible using simple semantics (example 2 below proves a similar postcondition to a similar program using the extended semantics of multi-Ports). The idea behind the development, then, is to extend the sharing (i.e. referencing) of interface variables in a safe manner.

We distinguish two types of multi-Ports: the <u>many-to-many Port</u> and the <u>many-to-one Port</u>. The many-to-many Port is the more general type and is used in configurations in which there exists more than one lowest level input process and more than one lowest level output

process naming the Port. The many-to-one Port is distinguished by there being no more than one lowest level process naming it for input or no more than one lowest level process naming it for output. Many-to-one Ports resemble the Port as described and implemented by Silberschatz in Silberschatz [80].

Since virtually all reasonable and interesting applications of multi-Ports assume the many-to-one configuration -- see Producer-Consumers and Distributed Operating System examples -- all of the examples presented below will be of that type, even when we are demonstrating many-to-many semantics.

2. FORMAL DESCRIPTION

As previously demonstrated, the sharing of interface variables among more than a pair of processes is unsafe and therefore forbidden by the simple-Port axioms. In introducing legitimate, permissible sharing, we replace interface variables by vector type equivalents, each component of which corresponds to a particular process, and admit a relation on the components that holds at all points of the text in the scope of the Port set (see next section). The power of selected, undesired components is nullified in uncertain program regions by the use of the universal quantifier.

Throughout what follows, the term process refers to lowest level process. A general (i.e. many-to-many) multi-Port is defined as follows.

1. Multi-Port Port variables, and auxiliary variables that are referenced by multi-Port A are of the vector type $\vec{i}$ = $(i_{P_{1_{in}}}, i_{P_{1_{out}}}, i_{P_{2_{in}}}, \ldots, i_{P_{n_{out}}})$, 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 <u>in</u> (or <u>out</u>) and write simply i_j.
2. Associated with multi-Port A is a <u>multi-Port variable</u> $\vec{a}$. Synchronization of A?x in process P_j and A!y in process P_k is interpreted as atomic execution of the sequence

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

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 multi-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. A typical S_A might be $\vec{i}:=\vec{j}+1$. Synchronization of A?x in process P_j and A!y in process P_k is interpreted to mean atomic execution of the sequence

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

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

5. Associated with A are assertions $\beta_A(\vec{a})$ and α_A, counterparts to simple-Port assertions of the same designation. 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 Ports). A typical $\beta_A(\vec{a})$ might be $\vec{a}=\vec{b}$.

Thus, each process using a multi-Port has its own private data base, the variables of which are the vector components corresponding to that process. As developed thus far, they are strictly disjoint data bases.

3. SEMANTICS

General multi-Port semantics are expressed as a generalized form of simple-Port semantics. We introduce an invariant relation on the components of certain interface variables and the axioms that constitute the verification methodology. Axioms 1, 4, 5, and 6 from the simple case automatically carry over; thus, we shall specify axioms that are counterparts to 2 and 3.

1. For multi-Port A, Kirchoff's Law is an invariant relation on $\vec{t}_A$ and $\vec{a}$:

   (i) Initially, $t_{A_{j_{in}}} = t_{A_{j_{out}}} = 0$ holds for all j.
   This is used when applying the axiom of substitution in that only the constant $\vec{0}$ may be subtituted for $\vec{t}_A$ in a process precondition.

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

   is such that $\{TRUE\}S\{KL_A\}$ is valid for any S in the scope of A.

Intuitively, Kirchoff'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.

2'. General Multi-Port Axiom for Parallel Composition

$$\frac{\{L_i\}P_i\{M_i\} \quad \text{for } i=1, \ldots, n}{\{\prod_{i=1}^{n} L_i\} \begin{array}{l}\text{declare}\\ \text{multi-Ports}\end{array} PO_1, \ldots, PO_m; [P_1 || P_2 || \ldots || P_n] \{\prod_{i=1}^{n} M_i \wedge \prod_{i=1}^{m} KL_{PO_i}\}}$$

, where each Port or auxiliary variable of scalar type appears free in the pre- and/or postcondition of only those 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. $S_A(\vec{a})$ is the multi-Port action defined for A. For A?x in process P_j and A!y in process P_k

(i) $wlp(A?x, M(x)) =$

$$(\prod_i \forall a_{j_{in}} \forall v_r, \vec{v} \in V, r \neq j_{in} [(\beta_A(a_{j_{in}})_{i_{out}} \wedge \prod_{P \in U} KL_P) \Rightarrow wp(t_{A_{j_{in}}} := t_{A_{j_{in}}} + 1;$$

$$S_A(\vec{a})_{j_{in}}, M(a_{j_{in}}))]) \wedge \alpha_{A_{j_{in}}}$$

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

$$(\prod_i \forall a_{k_{out}} \forall v_r, \vec{v} \in V, r \neq k_{out} [(\alpha_{A_{i_{in}}} \wedge \prod_{P \in U} KL_P) \Rightarrow wp(t_{A_{k_{out}}} := t_{A_{k_{out}}} + 1;$$

$$S_A(y)_{k_{out}}, R(y))]) \wedge \beta_A(y)_{k_{out}}$$

, where j_{in} indicates that all references to vector variables in generic form (e.g. $\vec{t}_A$) refer to components corresponding to $P_{j_{in}}$; similarly for k_{out}. $\forall \vec{v} \in V$ means "for all variables $\vec{v}$ in V;" $r \neq j_{in}$ means "for all r not equal to j_{in}," and similarly for k_{out}.

4. EXAMPLES OF MULTI-PORT PROOFS

The examples presented in this section are chosen to illustrate how the axioms for general multi-Ports are applied in verifying properties about parallel programs. They come from the realm of parallel systems and for this reason perform computations that are not particularly significant. Their significance comes from their use as building blocks in the construction of large systems. Note the prominent and central role played by Kirchoff's Law in the proofs.

1. Simple Example:

Consider the three processes 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. 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 wish to show that $\{x\geq 0 \wedge x=c\}[P\|Q\|R]\{x=0 \wedge m+n=c\}$ is valid. A is defined such that $\alpha_A \equiv \beta_A \equiv$ TRUE and $S_A \equiv$ SKIP. The annotated text appears as shown.

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

$*[\ \{I_R\}x>0\rightarrow\{(c=t_{A_{R_{out}}}+x)\wedge(x>0)\}A!count()\{(c=t_{A_{R_{out}}}+x-1)\wedge$

$(x>0)\}; x:=x-1\{IL_R\}] \{IL_R \wedge (x>0)\}$

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

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

$m:=0$

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

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

$\{IL_P\}$

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

Direct application of Axioms 3'(i) and (ii) establish the assertions surrounding the I/O statements in both proofs. Thus far we have established $\{x \geq 0 \wedge x=c \wedge t_{A_{R_{out}}}=0\} R \{IL_R \wedge (x>0)\}$ and $\{t_{A_{P_{in}}}=0\} P \{m=t_{A_{P_{in}}}\}$ against the interface. Q being virtually identical to P, the validity of $\{t_{A_{Q_{in}}}=0\}$ $Q\{n=t_{A_{Q_{in}}}\}$ follows from the proof of P. Since KL_A implies

$$t_{A_{R_{out}}} = t_{A_{P_{in}}} + t_{A_{Q_{in}}}, \qquad (1)$$

applying the axioms for parallel composition (Axiom 2') and substitution yields the desired result. Q.E.D.

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 (1). Processes may be added without need for total reverification as long as they preserve (1).

2. Who Received the Last Item:

Consider P, Q, and R operating in parallel and sharing DYNAMIC Port A:

```
R:: *[even(x)->x:=x/2;A!x]
P:: *[A?z->SKIP]
Q:: *[A?y->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 as shown.

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

$*[\{IL_R\}even(x)->\{TRUE\}x:=x/2\{TRUE\};A!x\{IL_R\}]$

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

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

$*[\{IL_P\}A?z->SKIP\{IL_P\}]$

$\{IL_P\}$

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

Once again, direct applications of the communication axioms justify most triples; the remaining ones are sequential and handled in the routine fashion. The proof $\{t_{A_{Q_{in}}}=0\}Q\{t_{A_{Q_{in}}}>0\Rightarrow y=a_{Q_{in}}\}$ is similar to the one for P and will be omitted.

Now $(IL_R\wedge odd(x)\wedge KL_A)\Rightarrow$

$$[(odd(x))\wedge(x=a_{R_{out}})\wedge(t_{A_{P_{in}}}>0\vee t_{A_{Q_{in}}}>0)\wedge (a_{R_{out}}=a_{P_{in}}\vee a_{R_{out}}=a_{Q_{in}})\wedge(t_{A_{P_{in}}}=0\Rightarrow a_{R_{out}}=a_{Q_{in}})\wedge (t_{A_{Q_{in}}}=0\Rightarrow a_{R_{out}}=a_{P_{in}})] \quad ; \qquad (2)$$

therefore $(IL_R\wedge odd(x)\wedge IL_P\wedge IL_Q\wedge KL_A)\Rightarrow(odd(x)\wedge(odd(y)\vee odd(z)))$.

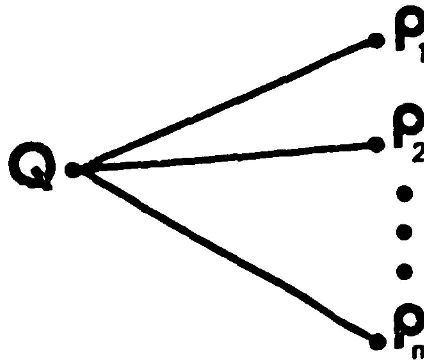
$(IL_R \wedge odd(x) \wedge IL_P \wedge IL_Q \wedge KL_A)$ is obtained as the postcondition of $[P||Q||R]$ upon applying the axiom for parallel composition, Axiom 2', and the result is established by an appeal to the axiom for substitution that eliminates $\vec{t}_A = \vec{0}$ from the precondition. Q.E.D.

The addition of other processes that use Port A for input invalidates the proof since (2) would not hold; KL_A lacks sufficient strength in that case.

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 that no such variable spans more than a pair of processes. We expand the general semantics developed thus far and conclude with two examples.

Let $P = \{P_1, \ldots, P_n\}$ be a collection of lowest level processes and let U be the set of Ports used for communication only between Q and some subset of P in a many-to-one configuration as shown.



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 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 the following conditions must hold:

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

   for $A \in U$,

   (i) 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 (i.e. $\beta_A(\vec{a})$ mentions no components);

   (ii) 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. Local process proof assertion (free) references to variables of V must be to components, subject to the following:

   (i) Q can mention any component;

   (ii) P_i can mention only components subscripted i.

   (Hence components subscripted i are <u>owned</u> by processes Q and P_i).

The revised axiom for parallel composition appears as follows.

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

$$\frac{\{L_i\}P_i\{M_i\} \quad \text{for } i=1,\ldots,n}{\{\prod_{i=1}^{n} L_i\}\text{declare many-to-one Ports } PO_1,\ldots,PO_m;[P_1||P_2||\cdots||P_n]\{\prod_{i=1}^{n} M_i \wedge \prod_{i=1}^{m} KL_{PO_i}\}}$$

, where condition 4 above holds.

The revised axioms of communication are given as follows.

3''. Axioms of Communication For Many-to-One Port

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

(i) wlp(A?x,M(x)) =

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

$$t_{A_Q} := t_{A_Q}+1; S_A(\vec{a})_Q, M(a_Q)))] \wedge \alpha_A$$

(ii) wlp(A!y,R(a_i)) =

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

$$R(y))])\wedge\beta_A(y)_i$$

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

(iii) wlp(A!y,R(a_Q)) =

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

$$S_A(y)_Q, R(y)))]\wedge\beta_A(y)$$

(iv) $wlp(A?x, M(x)) =$

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

, where subscript i indicates vector components corresponding to process P_i; subscript Q indicates vector components corresponding to process Q. $\vec{v} \in V$ and $j \neq i$ have meanings similar to their meanings in Axiom 3'.

6. EXAMPLES OF MANY-TO-ONE PORT PROOFS

1. Simple Example:

Consider the system A: PORT(DYNAMIC,integer); B:PORT(STATIC,integer);[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] .

We will prove {TRUE}[P||Q(i:1..n)]{x=y} .

A and B are defined

	A	B
S	SKIP	SKIP
α	TRUE	TRUE
β	TRUE	$\vec{t}_A = \vec{t}_B+1>0$

Subscript j denotes 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

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

$*[\{IL_p\}m>0->\{\prod_{j=1}^{n} t_{A_j}=t_{B_j}\}A!x\{\sum_{j=1}^{n}[t_{A_j}=t_{B_j}+1 \wedge x=a_j \wedge$

$\prod_{k=1}^{n}(k\neq j=>t_{A_k}=t_{B_k})]\};B?y\{\prod_{j=1}^{n}(t_{A_j}=t_{B_j})\wedge(\sum_{j=1}^{n}(x=a_j \wedge y=b_j \wedge$

$t_{A_j}>0))\}$;m:=m-1{ILp}]{ILp$\wedge$m$\leq$0}

, where the loop invariant ILp is

$$m\geq 0 \wedge \prod_{j=1}^{n} (t_{A_j}=t_{B_j}) \wedge (m=0 \Rightarrow \sum_{j=1}^{n} (x=a_j \wedge y=b_j \wedge t_{A_j}>0)) \quad .$$

(ILp$\wedge$m$\leq$0), the loop exit condition, implies the desired postcondition. We will demonstrate the validity of the triple involving A!x to illustrate the use of the axioms. From

$$3''(iii),\ wlp(A!x, \sum_{j=1}^{n} [t_{A_j}=t_{B_j}+1 \wedge x=a_j \wedge \prod_{k=1}^{n} (k\neq j \Rightarrow t_{A_k}=t_{B_k})]) =$$

$$[\prod_{i=1}^{n} \forall a_i((TRUE \wedge KL_A \wedge KL_B) \Rightarrow wp(a_i:=x; t_{A_i}:=t_{A_i}+1; SKIP;$$

$$t_{A_P}:=t_{A_P}+1;\ SKIP,\ \sum_{j=1}^{n} [t_{A_j}=t_{B_j}+1 \wedge x=a_j \wedge \prod_{k=1}^{n} (k\neq j \Rightarrow t_{A_k}=t_{B_k})])))] \wedge$$

TRUE

$$= \prod_{i=1}^{n} \forall a_i((KL_A \wedge KL_B) \Rightarrow wp(a_i:=x; t_{A_i}:=t_{A_i}+1, \sum_{j=1}^{n} [t_{A_j}=t_{B_j}+1 \wedge$$

$$x=a_j \wedge \prod_{k=1}^{n} (k\neq j \Rightarrow t_{A_k}=t_{B_k})]))$$

$$= \prod_{i=1}^{n} \forall a_i((KL_A \wedge KL_B) \Rightarrow wp(a_i:=x, x=a_i \wedge \prod_{k=1}^{n} (t_{A_k}=t_{B_k})))$$

$$= \prod_{i=1}^{n} \forall a_i((KL_A \wedge KL_B) \Rightarrow \prod_{k=1}^{n} (t_{A_k}=t_{B_k}))$$

$$= (KL_A \wedge KL_B) \Rightarrow \prod_{k=1}^{n} (t_{A_k}=t_{B_k}) \quad . \qquad (1)$$

The precondition found in the annotated text clearly implies (1).

The proof of $\{t_{A_i}=t_{B_i}=0\}Q(i)\{t_{A_i}>0 \Rightarrow a_i=b_i\}$

is as follows.

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

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

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

The final result is a consequence of the axioms for parallel composition (Axiom 2") and substitution. Q.E.D.

2. Subroutines:

We prove the single entry subroutine, which is similar to that described in Hoare [78]. A subroutine process is patterned as

SUBR:: initialization; *[A?(input parameters)-> ... ;B!(output parameters)] , where ... computes the output parameters from the input parameters. The user calls SUBR with the pair of commands

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

, where ... is executed concurrently with the subrouti.e. There are many calling processes but only one copy of SUBR; hence, A and B are many-to-one Ports. A is DYNAMIC and B is STATIC since we do not want SUBR to terminate (synonymous with disappear) until all its users have.

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

SINE:: $*[A?\theta \rightarrow s:=\sin(\theta);B!s]$.

The proof verifies that if all users adhere to the calling sequence protocal, then y=sin x holds immediately following B?y, modulo termination. The Ports are defined as shown.

	A	B
S	SKIP	SKIP
α	TRUE	$\vec{t}_A=\vec{t}_B+1$
β	TRUE	$\sum_{j=1}^{n} [t_{A_j}=t_{B_j}+1 \wedge \vec{b}=\sin a_j \wedge \prod_{k=1}^{n} (k\neq j \Rightarrow t_{A_k}=t_{B_k})]$

β_B tells the inputting process that the data value, $\vec{b}$, equals $\sin a_j$ for some calling process j; the calling process is characterized at this point of execution by the fact that its number of transactions through A exceeds its number of transactions through B by one. If all processes adhere to the protocal, the inputting process and the calling process are one and the same.

The annotated text follows:

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

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

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

$$b_k=\sin a_k)]]\};B!s\{IL_{SIN}\}]\{IL_{SIN}\}$$

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

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 ℓ violates the calling sequence protocal, then $(t_{A_\ell}=t_{B_\ell})$ will not hold as a valid postcondition, and the axioms for parallel composition (Axiom 2") and substitution will yield {TRUE}[SINE||USER(i:1..n)]{FALSE}. If all users adhere to the protocal, {TRUE}[SINE||USER(i:1..n)]{TRUE} holds. We shall demonstrate the validity of the triple

$\{x=a_i \wedge t_{A_i}=t_{B_i}+1\}B?y\{y=\sin x \wedge t_{A_i}=t_{B_i}\}$.

From Axiom 3" (iv), $wlp(B?y, y=\sin x \wedge t_{A_i}=t_{B_i})$ =

$(\forall b_i \forall v_j, j \neq i, [(\beta_B(b_i) \wedge KL_A \wedge KL_B) \Rightarrow wp(t_{B_i} := t_{B_i}+1; SKIP,$

$b_i = \sin x \wedge t_{A_i} = t_{B_i})]) \wedge \alpha_{A_i}$

$$= (\forall b_i \forall v_j, j \neq i, [(\beta_B(b_i) \wedge KL_A \wedge KL_B) \Rightarrow (b_i = \sin x \wedge t_{A_i} = t_{B_i}+1)]) \wedge \alpha_{A_i} . \quad (2)$$

We drop $KL_A \wedge KL_B$ from (2) since we have no need for it; besides, we can always weaken the antecedent. Thus (2) becomes

$$(\forall b_i \forall v_j, j \neq i, [(\sum_{j=1}^{n} [t_{A_j} = t_{B_j}+1 \wedge b_i = \sin a_j \wedge \prod_{k=1}^{n} (k \neq j \Rightarrow t_{A_k} = t_{B_k})]) \Rightarrow$$

$$b_i = \sin x \wedge t_{A_i} = t_{B_i}+1])\wedge t_{A_i} = t_{B_i}+1 \quad . \quad (3)$$

The precondition tells us that j in (3) is equal to i and hence $b_i = \sin a_i$. Thus, the precondition implies (3). Q.E.D.

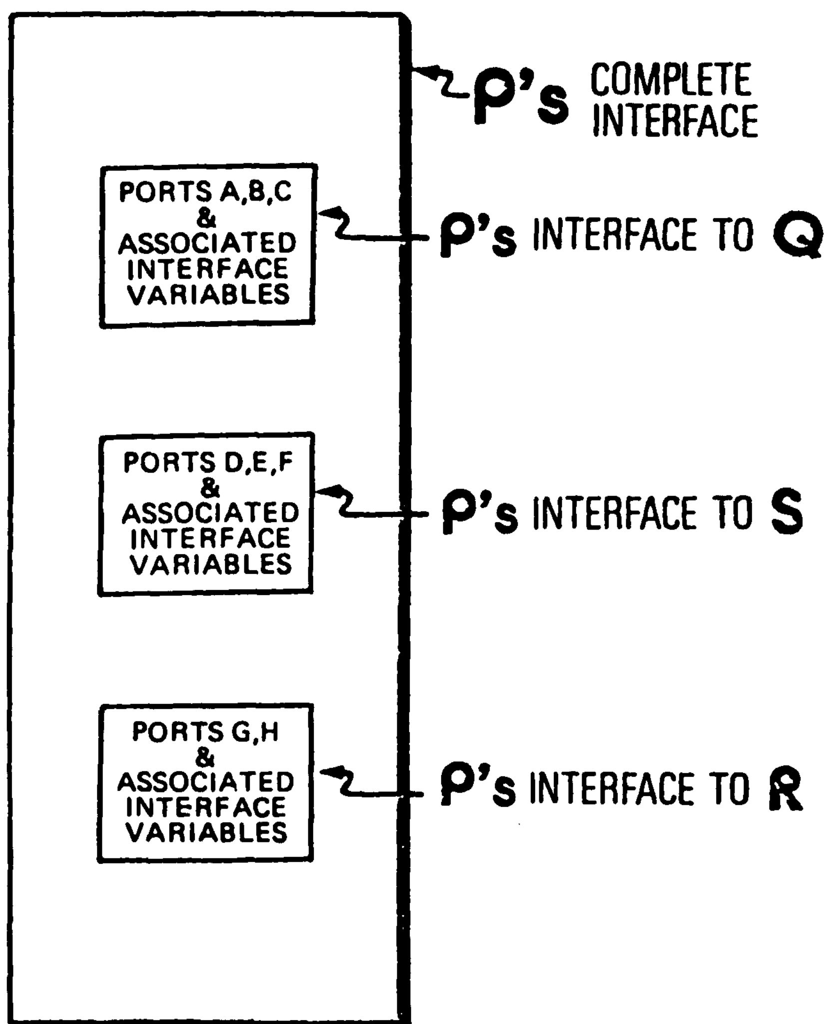
VI CONCLUSION

It is, perhaps, fitting at this point to mention a slightly varied approach that some may find preferable, but is, to an extent, more restrictive. In practice, however, the restrictions indicated below are generally inconsequential since all reasonable programs follow the pattern anyhow.

First, 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. Appropriate syntactic modifications can be made to accomodate this.

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 the following situation. Given a process P, its 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, as described above:



The benefits gained by this approach are the following:

1) While logically equivalent to the original methodology, this is, perhaps, a better organized and more structured arrangement;
2) Simplification of closure dynamics of DYNAMIC Ports;
3) Elimination of subscripts "in"/"out" when dealing with multi-Ports;
4) Elimination of the notion of ownership of interface variables (interface variables are now automatically owned by only 2 processes), with a resulting simplification of the axiom for

parallel composition;

5) The generalization from simple-Port semantics to multi-Port semantics becomes slightly more natural in this context.

Looking back to the introduction, it seems that we have reached our goal: the establishment of modularity and hierarchical structure within CSP, and the establishment of a proof system with those same characteristics. What we have created, then, is a methodology whereby the proof of a system is constructed along the same lines as the system itself. The proof of an entire system can be easily constructed from proofs of the individual components. This advantageous quality has always been characteristic of strictly sequential programs; unfortunately it did not easily carry over to parallel systems.

This we have re-established the isomorphic relationship between the structure of programs and the structure of proofs. We have effectively eliminated the need for many extraneous devices in the proof, such as the global invariant, that have no counterpart within the actual program. Parallel programs and their proofs can share a common form.

VII APPENDIX

BNF DESCRIPTION OF CSP

The following is a complete BNF description of CSP syntax taken from Hoare [78]. Types, declarations, and expressions have been left unspecified and are assumed to be Pascal-like. Braces {} denote none or more repetitions of the enclosed text.

```
<command>::=<simple command>|<structured command>
<simple command>::=<null command>|<assignment command>
      |<input command>|<output command>
<structured command>::=<alternative command>
      |<repetitive command>|<parallel command>
<null command>::SKIP
<command list>::={<declaration>;|<command>;}<command>
```

Parallel Command:

```
<parallel command>::=[<process>{||<process>}]
<process>::=<process label><command list>
<process label>::=<empty>|<identifier>::
      |<identifier>(<label subscript>{,<label subscript>})::
<label subscript>::=<integer constant>|<range>
<integer constant>::=<numeral>|<bound variable>
<bound variable>::=<identifier>
<range>::=<bound variable>:<lower bound>..<upper bound>
<lower bound>::=<integer constant>
<upper bound>::=<integer constant>
```

Assignment Command:

```
<assignment command>::=<target variable>:=<expression>
<expression>::=<simple expression>|<structured expression>
<structured expression>::=<constructor>(<expression list>)
<constructor>::=<identifier>|<empty>
<expression list>::=<empty>|<expression>{,<expression>}
<target variable>::=<simple variable>|<structured target>
<structured target>::=<constructor>(<target variable list>)
<target variable list>::=<empty>|<target variable>
       {,<target variable>}
```

Input and Output Commands:

```
<input command>::=<source>?<target variable>
<output command>::=<destination>!<expression>
<source>::=<process name>
<destination>::=<process name>
<process name>::=<identifier>|identifier>(<subscripts>)
<subscripts>::=<integer expression>{,<integer expression>}
```

Alternative and Repetitive Commands:

```
<repetitive command>::=*<alternative command>
<alternative command>::=[<guarded command>
       {□<guarded command>}]
<guarded command>::=<guard>-><command list>
       |(<range>{,<range>})<guard>-><command list>
<guard>|:=<guard list>|<guard list>;<input command>
       |<input command>
<guard list>::=<guard element>{;<guard element>}
```

<guard element>::=<boolean expression>|<declaration>

VIII BIBLIOGRAPHY

[Akkoyunlu 72] E.A. Akkoyunlu, A.J. Bernstein, R. Schantz, "An operating system for a network environment," 22d Int. Symposium on Computer Communications, Networks, Teletraffic, New York, April 1972.

[Akkoyunlu 78] E.A. Akkoyunlu, A.J. Bernstein, F.B. Schneider, A. Silberschatz, "Conditions for the equivalence of synchronous and asychronous systems," IEEE Trans on Software Eng., November 1978.

[Apt 80] K. Apt, N. Francez and W. de Roever, "A Proof System for Communicating Sequential Processes," ACM Trans. Program Lang. Syst., 2,3, July 1980, (pp 359-385).

[Ashcroft 75] E. Ashcroft, "Proving Assertions About Parallel Programs," J. Comput. Syst., 10, 1975, (pp 110-135).

[Baker 72] F.T. Baker, "Chief Programmer Team Management of Programming," IBM Systems Journal, 11,1, 1972 (pp 56-73).

[Bernstein 80] A. Bernstein, "Output Guards and Nondeterminism in 'Communicating Sequential Processes'," ACM Trans. Prog. Lang. Syst., 2,2, April 1980, (pp 234-238).

[Böhm 66] C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with only two formation rues," CACM, Vol 9, May 1966, (pp 366-371).

[Brinch Hansen 73a] Per Brinch Hansen, The RC 4000 Operating System, Prentice-Hall, 1973.

[Brinch Hansen 73b] Per Brinch Hansen, Operating System Principles Prentice-Hall, 1973.

[Brinch Hansen 75] Per Brinch Hansen, "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering SE-1,2 June 1975, (pp 199-206).

[Brinch Hansen 78] Per Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," CACM, Nov. 1978, (pp 934-941).

[Bruno 72] Bruno and Steiglitz, "The Expression of Algorithms by Charts", JACM, July 1972.

[Campbell 74] R. Campbell, and A.N. Habermann, Specification of Process Synchronization by Path Expressions, Lecture Notes in Computer Science, Springer Verlag, 1974.

[Dahl 68] O.J. Dahl, B. Myhrhaung and K. Nygaard, The Simula 67 Base Language, Norwegian Computing Center, Oslo, Norway, 1968.

[de Bakker 80] J. de Bakker, Mathematical Theory of Program Correctness, Prentice-Hall International, 1980.

[Dijkstra 68a] E.W. Dijkstra, "The Structure of the THE Multiprogramming System," CACM, Vol. 11, May 1968.

[Dijkstra 68b] E.W. Dijkstra, "Co-operating Sequential Processes," in Programming Languages, F. Genuys (ed.), Academic Press, New York, (pp 43-112).

[Dijkstra 68c] E.W. Dijkstra, "Go To Statement Considered Harmful," Communications of the ACM, 11,3 March 1968 (pp 147-148).

[Dijkstra 75] E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs," CACM, 18,8, 1975.

[Dijkstra 76] E.W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

[Eswaran 76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," CACM, Vol. 19, No. 11, November 1976.

[Floyd 67] Robert W. Floyd, "Assigning Meanings to Programs," Proc. Symp. in Applied Mathematics, Vol 19 (J.T. Schwartz, ed.), American Mathematical Society, 1967 (pp 19-32).

[Hoare 69] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," Communications of the ACM, 12,10, October 1969 (pp 576-580,583).

[Hoare 72] C.A.R. Hoare, "Proof of Correctness of Data Representations," Acta Informatica, 1,4, 1972 (pp 271-281).

[Hoare 73] C.A.R. Hoare, "Procedures and Parameters, an Axiomatic Approach," Symp. on Semantics of Prog. Lang., Lecture Notes in Mathematics, Springer Vol 188, 1973.

[Hoare 74] C.A.R. Hoare,"Monitors: An Operating System Structuring Concept," Comm. ACM, vol. 17 No. 10, October 1974.

[Hoare 76] C.A.R. Hoare, "Some Properties of Nondeterministic Computations," unpublished, March 1976.

[Hoare 78] C.A.R. Hoare, "Communicating Sequential Processes," Commun. ACM, 21,8, August 1978, (pp 666-677).

[Howard 76a] J.H. Howard, "Proving Monitors," Comm. ACM, vol. 19, No. 5, May 1976, (pp 273-279).

[Howard 76b] J.H. Howard, "Signaling in Monitors," Proceedings of 2nd Annual Conference on Software Engineering, San Francisco, California, pp 47-52, October 1976.

[Jones 76] Anita K. Jones and Barbara H. Liskov, "An Access Control Facility for Programming Languages," Computation Structures Group Memo 137, Massachusetts Institute of Technology and Carnegie-Mellon University Technical Report, 1976.

[Kessels 77] J. Kessels, "An Alternative to Event Queues for Synchronization in Monitors," CACM, July 1977, (pp 500-503).

[Kieburtz 77] R.B. Kieburtz, J.L. Hennessy, "Axioms for Monitors," SUNY Stony Brook, TR 61, February 1977.

[Kieburtz 79] R.B. Kieburtz, A. Silberschatz, "Comments on 'Communicating Sequential Processes'," ACM Trans. on Programming Languages and Systems, 1,2, October 1979, (pp 218-225).

[Lamport 77] L. Lamport, "Proving the Correctness of Multiprocess Programs," IEEE Trans. Softw. Eng., 3,2, 1977, (pp. 125-143).

[Liskov 73] B.H. Liskov, "The Venus Operating System," CACM, 1973.

[Liskov 74] Barbara Liskov, "A Note on CLU," MAC-TR, Massachusetts Institute of Technology, Nov. 1974.

[Manna 74] Zohar Manna, Mathematical Theory of Computation, McGraw-Hill, 1974.

[Mao 80] T. Mao and R. Yeh, "Communication Port: A Language Concept for Concurrent Programming," IEEE Trans. Softw. Eng., 6,2, March 1980, (pp 194-204).

[Mills 71] H.D. Mills, "Top Down Programming in Large Systems", in Debugging Techniques in Large Systems, Prentice-Hall, 1971.

[Nassi 74] I. Nassi, E.A. Akkoyunlu, "Verification Techniques for a hierarchy of Control Structures," Technical Report 26, SUNY at Stony Brook, January 1974.

[Owicki 76a] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," CACM Vol. 19, No. 5, May 1976, (pp 279-289).

[Owicki 76b] S. Owicki, "A Consistent and Complete Deductive System for the Verification of Parallel Programs," Proc. 8th ACM Symp. on Theory of Computing, 1976, (pp 73-86).

[Parnas 71] D.L. Parnas, "Information Distribution Aspects of Design Methodology," IFIP Congress 1971, Booklet TA-3 (pp 26-30).

[Parnas 72a] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, 15,12, December 1972 (pp 1053-1058).

[Parnas 72b] D.L. Parnas, "A Technique for Software Module Specification with Examples", Communications of the ACM, 15,5, May 1972 (pp 330-336).

[Silberschatz 77] A. Silberschatz, R. Kieburtz, and A. Bernstein, "Extending Concurrent Pascal to Allow Dynamic Resource Management," IEEE Trans. on Softw. Eng., 3,3, May 1977, (pp 210-217).

[Silberschatz 79] A. Silberschatz, "Communication and Synchronization in Distributed Systems," IEEE Trans. Softw. Eng., 5,6, Nov. 1979, (pp 542-546).

[Silberschatz 80] A. Silberschatz, "Port-Directed Communication," private manuscript, 1980.

[Spitzen 75] Jay Spitzen and Ben Wegbreit, "The Verification and Synthesis of Data Structures," Acta Informatica, 4,2, 1975 (pp 127-144).

[Wegbreit 76] Ben Wegbreit and Jay M. Spitzen, "Proving Properties of Complex Data Structures," Journal of the ACM, 23, 2 April 1976 (pp 389-396).

[Wirth 71] Niklaus Wirth, "Program Development by Stepwise Refinement", Communications of the ACM, 14, 4, April 1971 (pp 221-227).

[Wirth 77a] N. Wirth, "Modula: A Language for Modular Multi-Programming," Software-Practice and Experience, Vol. 7, No. 1, January 1977.

[Wirth 77b] N. Wirth, "Toward a Discipline of Real-Time Programming," CACM, Vol. 20, No. 8, August 1977.

[Wulf 74] W. Wulf, "ALPHARD: Towards a Language to Support Structured Programs," Carnegie Mellon University, Computer Science Department, Pittsburgh, Pa., April 1974.

[Wulf 76] W. Wulf, "Abstraction and Verification on ALPHARD", Carnegie Mellon University, TR, June 1976.

[Zilles 75] S.N. Zilles, "Abstract Specifications for Data Types", IBM Research Laboratory, San Jose, January 1975.