SPECIFICATION AND VERIFICATION OF COMMUNICATION IN PARALLEL SYSTEMS

Carl H. Hauser

Ph.D. Thesis

TR 80-420

SPECIFICATION AND VERIFICATION OF COMMUNICATION IN PARALLEL SYSTEMS

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment for the Degree of

Doctor of Philosophy

by

Carl Howard Hauser

May 1980

SPECIFICATION AND VERIFICATION OF COMMUNICATION

IN PARALLEL SYSTEMS

Carl Howard Hauser, Ph.D.
Cornell University, 1980

This thesis develops a verification theory for systems of parallel processes communicating with one another by sending messages. The goal is independent specification of processes in a system, so that the system can be verified from the specifications of its component processes.

A process is regarded as a generator of a formal language called its communication language. A specification is written as a first order predicate calculus formula about this formal language. Processes are verified by including assertions in them and proving that the assertions hold, much as is done for sequential programs.

An important aspect of the work is its treatment of non-determinism and its effect on the termination, non-termination, and deadlock properties of systems and processes. It is argued that non-determinism should be curtailed if specifications are to accurately reflect the specifier's wishes.

Rules relating assertions and statements are similar to those used in Floyd-Hoare logic. However, we deal here with both partial correctness, sometimes called safety in the literature, and liveness, which is a concept analagous to termination of sequential programs. Justification of liveness assertions takes advantage of statements that

occur both before and after the assertion, introducing a new kind of proof rule which we prove is correct.

Biographical Sketch

Carl Howard Hauser was born on June 9, 1953, in Seattle, Washington. He graduated from North Thurston High School in 1971 and was chosen a United States Presidential Scholar in that same year. He received the Bachelor of Science degree in Computer Science from Washington State University in June, 1975, awarded Summa Cum Laude. In May, 1977, he received the Master of Science degree in Computer Science from Cornell University.

To my parents, Ralph and Ardis, whose unflagging
support can never be repaid

Acknowledgments

I would like to express my deep appreciation to my thesis advisor, Robert Constable, for his encouragement, support, and advice throughout my years at Cornell; to Greg Andrews for sparking my interest in parallel programs; to David Griese and Jim Donahue for their continued interest in this work; and to Gary Levin and Jim Archer whose friendly criticism led to significant insights.

I would especially like to thank Jim Boyle at Argonne National Laboratory for encouraging me to take up graduate work in the first place.

Table of Contents

page

1. Introduction 1

2. An overview of specification and verification of program behavior 7

2.1 Formal meaning of programs 8

2.1.1 Sequential programs 8

2.1.2 Parallel programs 12

2.2 Formal specifications 19

2.2.1 Sequential programs 20

2.2.2 Parallel programs 20

2.3 Verification 22

2.3.1 Sequential programs 22

2.3.2 Parallel programs 23

2.4 Summary 25

3. A simple parallel programming language and its semantics 26

3.1 A simple parallel programming language (SPPL) 27

3.2 The formal semantics of SPPL 30

3.2.1 The semantics of program traces 31

3.2.2 Feasible traces of processes 33

3.2.3 Relational semantics of processes 35

3.2.4 Feasible traces of parallel programs 36

3.2.5 Example of a system 37

3.3 Conclusion 38

4. Program semantics and specifications........................ 39
4.1 A more abstract semantics of processes...................... 40
4.1.1 The language of a process.............................. 41
4.1.2 Process specifications................................. 42
4.1.3 Examples of process specifications..................... 46
4.1.4 Consistency of the language semantics for process specifications............................ 50
4.2 Language semantics of systems and subsystems................ 53
4.2.1 Systems of processes................................... 54
4.2.2 Example of a system.................................... 59
4.2.3 Subsystems... 62
4.2.4 Example.. 69
4.3 Conclusion.. 70
5. Verification of processes and systems.......................... 72
5.1 Rules for processes... 73
5.1.1 Proving safety properties of processes................. 79
5.1.2 Proving liveness properties of processes............... 82
5.2 Rules for systems... 91
5.3 Conclusion.. 92
6. Conclusion and Analysis.. 94
6.1 Extension of SPPL's communication mechanism................. 95
6.2 Directions for further research............................. 100
7. Bibliography... 101

Chapter One: Introduction

This research examines some of the problems involved in the specification and verification of concurrent systems. One class of concurrent systems, called the apparently deterministic systems, can be specified and verified in a natural way. The verification of certain _liveness_ properties of parallel system components employs a new technique for reasoning about programs that generalizes the standard termination argument based on well-founded orderings.

In the late 1950's the recognition that relatively independent computations could be managed by an operating system led to the second and third generation operating systems supporting multiple simultaneous users. An execution of a single user's program was named a _process_. The process abstraction is so useful that modern programming languages are incorporating it as a basic control construct, on a par with procedures. Operating systems themselves are typically regarded as being structured into processes for two reasons: modern hardware supports simultaneous activity at many levels, ranging from concurrent arithmetic-and-logic unit (alu) and peripheral device activity to multiple alu's in some machines; secondly, the resources of a system are relatively independent of one another. A relatively loosely coupled set of computations adequately manages them. Even this loose coupling, however, is tighter than that between the user processes of a multiprogrammed system. Therefore, verification of operating systems poses problems not found in the verification of user programs for a multiprogrammed environment.

Having disposed of the reasons for writing systems of parallel processes, the problem of how to go about writing them remains. Complexity in a programming problem can be dealt with by decomposing the problem into conceptual subproblems and dealing with the subproblems individually. In sequential programs, proper choice of subproblems leads to a simple interface between each subproblem solution and the rest of the program. One regards a subproblem solution as a relation between a set of states at the beginning of its execution and a set of states at the end of the execution. This is the basis of Dijkstra's predicate transformers and Hoare's axioms.

But a process, intended to cooperate over a period of time with other processes, cannot be regarded as simply expressing a relation between its input states and its output states. Consider, for example, processes One and Two in Figure 1.1. They have exactly the same input-output relation, but the behavior of a system comprising processes One and Three is very different than the behavior of a system comprising processes Two and Three. Why? Process Two, during an execution references variable x twice but process One only references it once. In an environment where x may be changed between these references the two processes will behave differently.

```
One: x := 0;        Two: x := 0;        Three: x := 0;
     y := 1              y := x+1              x := 1
```

Figure 1.1 Processes are not relations

In short, the techniques we use to describe processes must capture their ability to interact with other processes throughout their execution.

In writing a concurrent system three phases seem to be necessary:

1: Identifying the subproblems that should be expressed as processes, as opposed to those that should be solved using sequential programs.

2: Specifying the subproblems accurately, and verifying that their correct implementation will imply the correctness of the entire system.

3: Writing the programs that solve the subproblems and verifying that they meet their specifications.

The thesis of this work is that a reasonable approach to specifying and verifying processes and systems will support all three of the above phases. The careful programmer approaches a task intending to convince himself that his solution is correct. To do this he needs an accurate understanding of what he writes. The myriad of programming logics are attempts to capture, clarify, and mathematicize programmers' ways of understanding programs. The need for this is twofold. First, mathematical treatment may lead to better understanding, hence better programs developed at less cost, with less anguish on the part of programmers. Understanding a loop by seeing its invariant and its termination function can be a useful technique even if one never writes proofs in a Hoare style system. On the other hand, even with an excellent understanding of the programming constructs he uses, a programmer faces a great deal of complexity in a large program. This brings us to the second need for mathematical treatment of programming. Mechanical assistance for programmers in dealing with the semantic complexities of large programs requires that programs be very rigorously understood. I think we are already seeing benefits along the former lines as programmers become better educated in the foundations of programming. Mechanical aides like proof checkers and program verifiers are still in their infancy, but we look forward to systems that assist programmers throughout the programming process, such as that of Bates[6].

If a programming logic is to serve both of these purposes (the former being the more demanding), it seems that it must satisfy at least the following two criteria:

1: it must not depend on translations of programs into a form that obscures the subproblem and normal sequencing structure of the program;

2: it must be based on a programming language for which meaningful specifications for the solutions to subproblems can be given.

The methods used in the specification and verification of systems of processes depend on the constructs used to express the interactions between processes. Basically, processes interact by synchronization and communication. Methods can be distinguished by how they deal with these two forms of interaction.

One class of techniques expresses communication using variables shared between processes. Synchronization is accomplished using semaphores, conditional critical regions, monitors, or any of the other synchronization primitives that have been proposed. Such techniques could be said to use _explicit synchronization_ and _implicit communication_ since individual uses of shared variables look much like uses of ordinary, non-shared variables, while synchronization is done with new, distinctive operations.

Shared variable techniques may be contrasted with message passing techniques which we will say use _explicit communication and synchronization_. Communication is explicit because data to be transmitted is explicit in primitive communication statements. A process never takes an action that surprises another process. Explicit communication statements also serve as synchronization statements because processes may be delayed while waiting to send or receive data. In one variation of this basic scheme, messages are buffered and sending a value will not cause a delay of the sending process provided there is room in a buffer to hold the sent value. Of course, receivers are delayed when there are no messages to be received, and senders are delayed if there is no room in the buffer.

This thesis is organized as follows. Chapter Two is a general survey of techniques for specifying programs, together with some background information on formal semantics. Chapter Three presents a parallel programming language with explicit unbuffered communication and synchronization between processes. The semantics of the language is given both informally and formally in terms of simpler programs, called program traces, for which a semantics is given in terms of a a theory of strings. Chapter Four introduces another semantic characterization of processes in this language--one that does not require the explicit traces used in Chapter Three, but instead deals solely with the behavior of processes as seen by other process.

Chapter Four also introduces process specifications as predicate calculus assertions about processes' semantics. It is this use of the predicate calculus for specification that leads to the unification, accomplished in Chapter Five, of the theory of explicitly communicating processes with that of sequential programs and that of processes that share variables described by Owicki and Gries[58,59]. Two kinds of process specifications are used--safety specifications, analogous to partial correctness specifications of sequential programs, and liveness specifications, a generalization of the termination requirement of sequential programs. It is shown that both are necessary to adequately specify processes. Chapter Five deals with verifying that processes meet their specifications. To adequately deal with liveness specifications a new style of reasoning is introduced that generalizes the well foundedness arguments used in proving termination of sequential programs. The rules for constructing arguments are shown consistent with the formal semantics given earlier. Chapter Six summarizes the contributions of the thesis and points out some problems deserving future study.

Those unfamiliar with PL/CV are referred to [10] for a complete description. The proofs of the theorems in this thesis are themselves presented in the style of PL/CV, although they are informal. I hope their structure is pleasing to the reader.

Chapter Two: An Overview of Specification and Verification of Program Behavior

2. Introduction

Faced with the problem of writing a program to accomplish a particular task, a programmer brings to bear three kinds of knowledge: knowledge about how program statements work--the programming language's *semantics*; knowledge about what the problem is--*specifications*; and knowledge relating the way statements work to whether or not a given program solves a given problem--*verification*. Using this knowledge the programmer attempts to find a program that solves his problem.

A good deal of the research that carries the name "Computer Science" is directed at improving programmers' knowledge in one or another of the above areas. We try to put the knowledge on a sound formal and theoretical foundation, systematizing it, and making it easier to learn and use. We improve our programming languages by simplifying them and reducing the number of special cases that occur in their definitions; we improve techniques for formulating and communicating specifications; and we enable demonstration of correctness for all possible inputs with techniques that do not require that every possible input be tested.

Since Chapters Three, Four, and Five touch on all of these areas, in turn, this chapter reviews literature pertaining to all three. Although the work deals primarily with parallel programs, the techniques for sequential programs are reviewed as well, since concurrent activity is most frequently organized as multiple sequential activities (called processes).

2.1. Formal Meaning of Programs

2.1.1. Sequential Programs

Donahue[16] contains an essential review of the techniques of formal semantics of sequential programming languages. The three broad categories of semantic techniques discussed there are the operational, denotational, and propositional definitions of ALGOL-like (or imperative as opposed to LISP-like or applicative) languages. Each of these will be discussed in turn and then we will examine some of the work that has been done to extend the techniques to languages with concurrency.

The idea of "state" is pervasive in semantic definitions of ALGOL-like languages. For our purposes, a state is simply an interpretation of the free variables of logical formulae. The semantics of a programming language tells us the meaning of a particular program in that language by telling us something about the way execution of the program changes a state.

2.1.1.1. Operational Semantics

An operational semantics, as defined in [16], is a fairly low-level description of the state changes made by executing a program. Each basic statement of the programming language is defined by a "meaning function" mapping one state into another. Compound statements such as while-do and begin-end are defined to be functions mapping states into sequences of states. The meaning of a program is just such a map from states into sequences of states. The final state in a state sequence is the "output" of the program when executed with first state of the sequence as "input". Observe that the basic approach in this method is to define compound statements as appropriate compositions of the meaning function of the basic statements.

Contrast this with the approach taken by Hoare[31]. He uses _program traces_ to study correctness properties of non-deterministic programs. A program trace is simply a list of assignments and tests. It can itself be viewed as a particularly simple form of program--a straight line program. An execution of a trace, given an input state, either produces an output state or it aborts. A program that contains iteration and alternation statements can be represented by a set, possibly infinite, of program traces. Given an input state, a set of traces can be executed either producing the output produced by one of the traces or aborting. Characterization of the cases in which the execution of a set of traces aborts and in which it terminates in a final state are the subject of Hoare's paper.

The important observation here is that the semantics determines the final state by composition of the meaning function for straight line programs with a function mapping _while_ programs into sets of straight line programs. The function that produces a set of straight line programs from a _while_ program is perhaps more deserving of the name operational semantics than is the function we gave that name to earlier. When applied to a program it produces the sequence of operations (assignments and tests) that the program can perform. This technique will be exploited in Chapter 3 to define the semantics of a parallel programming language.

2.1.1.2. Denotational Semantics

Denotational (or functional) semantics regards a program as a function from an input state to an output state. The meaning of a program, F:S->S, is expressed as a recursive functional equation F=*F*(F) where *F* is a functional term involving F. The accepted meaning of the program is the least fixed point of this functional equation. As Donahue points out, the importance of this is that it leads to a different induction principle (fixed point induction) for proving properties of programs. Denotational semantics will play very little role in this thesis.

2.1.1.3. Propositional Semantics

Propositional semantics characterizes the behavior of a program by a formula in a logical system. The most widely known example of a propositional semantics is Hoare's axiomatic definition of PASCAL[33]. Sentences of the logic are of the form {P} S {Q} where S is a program and P and Q are first order predicate calculus formulae over a data domain (frequently the integers). Axioms of the logic define the basic statements of the language and inference rules generate new sentences describing compound statements such as <u>if</u>-<u>then</u>-<u>else</u>, <u>while</u>-<u>do</u>, and <u>begin</u>-<u>end</u>.

Viewed as a purely formal system, a propositional semantics has no notion of state at all. States are introduced by interpreting {P} S {Q} to mean that if execution of S is begun in a state satisfying P, and if the execution of S terminates, then it terminates in a state satisfying Q. Donahue gives complementary denotational and axiomatic (Hoare style) definitions of a subset of PASCAL and proves that the above

Unfortunately, many applications require that a process be able to receive from any one of several channels depending upon which have values available. The merge operator, mapping a pair of input channels into a single output channel is an example of such a process. merge('a', 'b') cannot be required to be 'ab' since that would require the 'a' to be available for input before the 'b' could be output, and it cannot be 'b' for similar reasons. The output of the merge function must be either 'ab' or 'ba' depending upon whether the 'a' or the 'b' was available first. Thus, the merge process cannot be represented by a <u>function</u> from its inputs to its output. Introducing such a non-functional process into a system means that the system can no longer be analyzed using the fixed point equations.

2.1.2.2. Infinite Sequence Semantics

The proceedings of the recent "Semantics of Concurrent Computation" symposium [37] contains many papers of interest to those working in this field. Papers by Pnueli[60] and Lynch and Fischer[45] are particularly germane to this discussion.

Pnueli uses a straightforward operational model of parallel computations as the foundation for his work in the temporal logic of parallel systems. The operational model that Lynch and Fischer use is similar. Both represent systems by infinite traces of the commands that they execute. Pnueli's model is the simpler of the two and I have chosen to present it in some detail.

In Pnueli's model, a concurrent system, S, consisting of n disjoint processes

$$S = ((G_1 \mathbin{//} G_2 \mathbin{//} \cdots \mathbin{//} G_n), \bar{\eta}_0)$$

is a set of n single entry transition graphs (SETGs), $\{G_1, G_2, \cdots, G_n\}$, together with a function $\bar{\eta}_0$ mapping the program variables to their initial values. ($\bar{\eta}_0$ is the initial *interpretation* of the program variables.) A SETG is a directed, labelled graph $G = (V, E, v)$ where V is a set of vertices, $v \in V$ is the start vertex, and E is a set of labelled edges. The labels on the edges are of the form $c \rightarrow \bar{y}b \rightarrow f(\bar{y})$, representing the indivisible test of condition c followed by the assignment of new values to the program variables $\bar{y}$ if c is true. If c is false, the assignment cannot be done. In this model all the program variables are shared between the processes, and synchronization is accomplished by testing the values of the variables. Each processor is located at a node in one of the graphs and may only move to an adjacent node along an edge whose label condition is true, performing the assignment specified by the edge. Figure 2.1 is an example of a system in this model. Notice that programs in languages with P and V semaphore operations or conditional critical regions (*with* resource *when* boolean *do* statement) [59] are easily represented by this model. Pnueli gives the translation.

A *state* in this model consists of a vector of n vertices, one from each of the processes, and an interpretation of each of the program variables. The vector $\langle \mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_n \rangle$ represents the location of n independent program counters in the n SETG's. Obviously, $\mathcal{V}_i \in V_i$, $1 \le i \le n$.

An *execution sequence* of a system is a sequence of states satisfying certain properties. Although not explicitly stated in the paper, there is a clear intention that execution sequences must be infinite. The properties that an execution sequence interpretation of the axiomatic definition is consistent with the denotational definition.

Dijkstra[13] proposes a logic of programs using the so-called weakest precondition operator. Sentences in this logic are first order predicate calculus formulae with the addition of the predicate transformer wp(S, R), where S is a program and R is a predicate. The predicate wp(S, R) characterizes all of the states s such that execution of S begun in state s satisfying wp(S, R) terminates in a state satisfying R. The axioms in this logic are expressed as equalities between formulae. The formula $P \Rightarrow wp(S, Q)$ is very close in meaning to the formula {P} S {Q} of Hoare's logic, but the former includes the implication that S terminates when started in any state satisfying P while the latter does not. This <u>total correctness</u> aspect of the weakest precondition logic is the subject of much recent study because termination properties of programs are very sensitive to the underlying model of computation. When the model allows non-determinism the term "non-determinism" must be defined accurately since there are several reasonable definitions with very different termination behaviors[31,42].

Propositional semantics is of particular interest in program verification: it is closely related to the standard specification technique for sequential programs that is described in 2.2.1. Section 2.3.1 examines the programming logic proposed by Constable and O'Donnell[11] as the basis of a program verification system. While it is another example of semantics in the propositional style, it is more appropriately discussed with material on verification.

2.1.2. Parallel Programs

The semantic characterization of parallel programming languages is considerably less well understood than that of sequential languages. In part, this stems from the variety of synchronization and communications mechanisms that have been put forth as components of parallel programming languages. Each seems to require a rather specialized treatment at any but the most mechanistic levels. More fundamentally, however, in a concurrent system with several processes each process must react not only to an input state, but also to changes made in the state while it is executing. Thus, the semantic techniques used for sequential programs which rely on mathematical function composition to model sequencing do not apply very well to systems with concurrency. This section describes some of the attempts at giving formal semantics for parallel programming languages.

2.1.2.1. Functional Semantics of Functional Processes

Kahn[36] and Keller[39] give denotational semantics for networks of processes communicating on infinitely buffered channels. Primitive operations are sending a value on a channel and receiving a value from a channel. A process cannot be delayed by executing a send operation, but a receive operation may cause a delay until there is a value available on the channel. The key to the success of the denotational semantics for this computation model is that every process computes a set of continuous functions from the histories of its input channels to the histories of its output channels. The semantics of a system is defined to be the least fixed point of a set of functional equations involving the functions computed by processes.

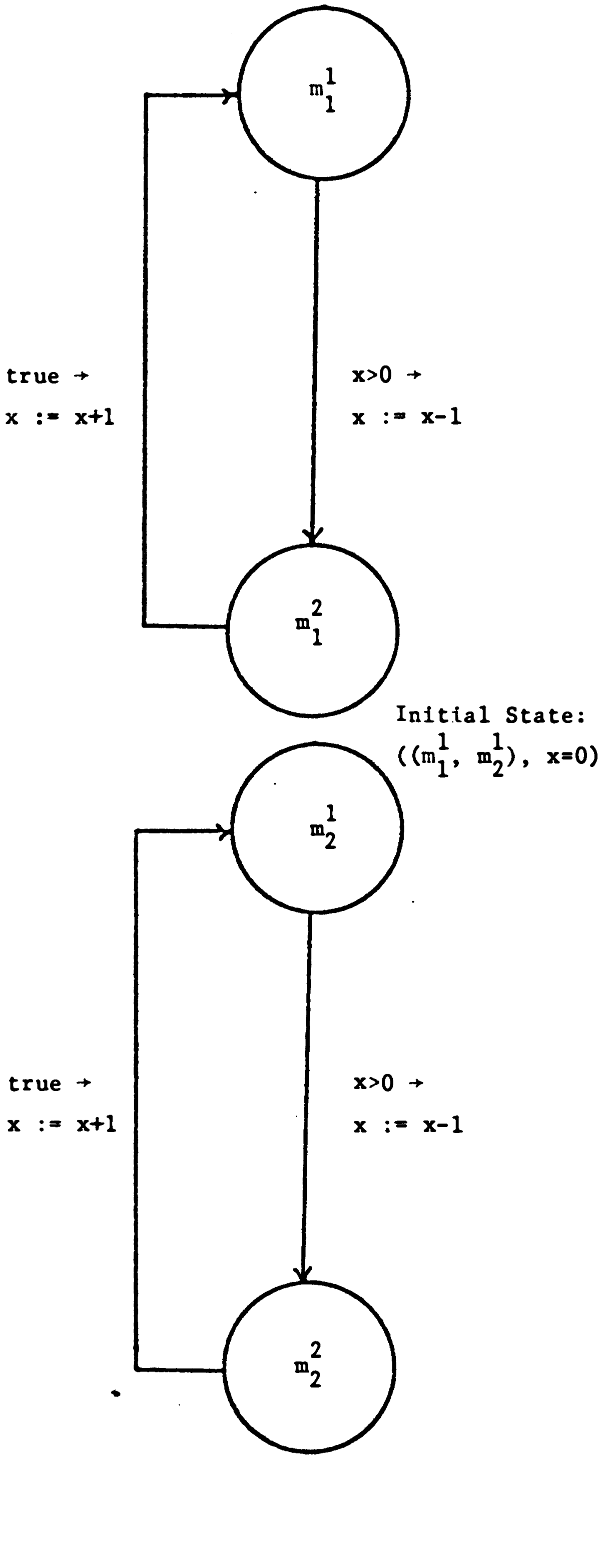

Figure 2.1
Pnueli's model

$<s_0, s_1, \ldots, s_i, \ldots>$ must satisfy are:

1: It must start in the correct state. That is, $s_0 = (<v_1, v_2, \ldots, v_n>, \bar{\eta}_0)$.

2: In the sequence, any pair of adjacent states s_j and s_{j+1} represent execution of at most one step by at most one processor. That is, at most one processor, say processor k, is at a different vertex in s_{j+1} than in s_j. Furthermore, it has moved along an edge of G_k such that the condition of the edge is true in state s_j, and the interpretation of the program variables in state s_{j+1} is the result of the application of the function on that edge to the interpretation of the variables in state s_j.

3: The sequence is <u>fair</u>. This means that if some processor is at a node almost everywhere in the sequence, then the disjunction of the conditions on the edges leaving that node is true at only a finite number of states in the sequence.

Refer again to Figure 2.1. The system there has execution sequences such as

$$<((m_1^1, m_2^1), 1), ((m_1^2, m_2^1), 0), ((m_1^1, m_2^1), 1), ((m_1^1, m_2^2), 0)>^{\infty}$$

but would not have any traces of the form

$$<((m_1^1, m_2), 1), ((m_1^2, m_2^1), 0)>^{\infty}$$

because they would violate the fair scheduling constraints.

The above is the operational description of Pnueli's parallel computation model. The paper also contains a formal description of these properties in terms of the <u>temporal logic</u>. Temporal logic is a logic allowing the expression of the time dependent characteristics of programs. Thus, formalizations of statements such as "eventually every execution of the program reaches a state s where P(s) is true" exist in temporal logic.

The Lynch and Fischer semantics differs from Pnueli's in assigning meaning to individual processes. The meaning of a system is defined by the shuffle composition of the meanings of the processes that comprise it. Both finite and infinite execution histories are treated in this model. The model has no synchronization mechanism built into it, forcing the programmer to implement interprocess synchronization in terms of the test and set operations that are primitive in the model.

2.1.2.3. Communicating Sequential Processes

The direct ancestors of the work in this thesis are the papers about Communicating Sequential Processes by Hoare[27,28,29,32]. The 1978 CACM paper [27] is expository, presenting a new set of language communication and synchronization primitives to be added to a sequential language based on Dijkstra's[13] guarded commands. In this paper Hoare proposes that communications between processes be made synchronous. This is equivalent to having unbuffered channels between processes--a sender cannot send until the receiver is ready to receive. The non-deterministic <u>do</u> and <u>if</u> of the guarded command language are used to express the possibility that a process may wait to receive any one of several messages at a time, then act upon whichever it receives first.

Hoare's later papers [28,29,32] discuss a simplified programming language in which processes and recursion replace local variables and iteration. This certainly does not make programs in the language more readable, but the semantics can be made very elegant.

The basic idea behind this language is that the execution of a process "communicates" a symbol with its environment, then invokes a new process. A process named "A" whose execution communicates an "a" then invoke s a process named "B" is written $A \equiv a \rightarrow B$. There is a special, pre-defined process (named ∇) that communicates a special symbol "∇" and terminates an execution. It is possible that an execution will be allowed to choose which of several symbols will be communicated at the first step. This is written $a_1 \rightarrow B_1 \square a_2 \rightarrow B_2 \square \ldots \square a_n \rightarrow B_n$. An execution communicates exactly one of $a_1 \ldots a_n$ then invokes the corresponding B process, but which one of $a_1 \cdots a_n$ will be communicated is unknown in advance.

As thus far described, a system of named processes may be looked upon as a regular grammar with the communication symbols as terminals and the process names as non-terminals. Taking A as the start symbol of the grammar, the prefix closure of the corresponding regular language is called the communication language language of of process A.

The class of communication languages is extended to the prefix closures of all context free languages by allowing the sequential composition of processes, written A;B. An execution of A;B is an execution of A followed by an execution of B. A string s is in the communications language of A;B if it is a string in the communication language of A, or s=xy where x∇ is in the language of A and y is in the language of B.

Processes also can be composed using the operator // denoting their concurrent execution. Communications force executions of processes to synchronize their behavior with executions of other processes. The communications language of the parallel composition of processes comprises the strings formed by shuffling the strings in the languages of the component processes, taking into account the synchronization rules.

Consider, for example, the system of processes

$A \,//\, B$

where

$A \equiv a \rightarrow C \quad B \equiv b \rightarrow D \quad C \equiv c \rightarrow A \quad D \equiv c \rightarrow B$

The system can communicate "a", "b", "ab", "ba", "abc", "bac", etc., but cannot communicate "aa", "bb", "acb", etc., because the two processes A and B share "c" as a communication symbol and must communicate it simultaneously. That is how synchronization is done in this programming language.

Chapters Three and Four of this thesis are an application of the above techniques to an ALGOL-like programming language.

2.2. Formal Specifications

A specification is an independent representation of what a program is supposed to do. Good specifications for programs are important because they can be used to increase our confidence that a program accomplishes a desired task. This comes about in two ways: first, as a formal tool, a specification language has different features than does a programming language. It requires different insight into an informal problem to formally specify it than it does to program a solution to it. Although a formal specification may itself be a program that could compute the desired results, albeit slowly, there is no requirement that this be so. In fact it probably should not be required that specifications be constructive. This brings us to the second aspect of increase in confidence by giving specifications: a program and its specification can be compared by verifying that the program and the specification agree. Since programs are inherently constructive, such a verification shows that the specification is consistent and constructively feasible.

Thus, we see programs and specifications as being mutually reinforcing. While there can be no guarantee that a formal specification exactly captures our intuitive, informal understanding of a problem, we have greater confidence in the program and specification together than in either alone.

2.2.1. Sequential Programs

Sequential programs are frequently specified by stating a relationship that must hold between an input state and an output state. This relationship is usually stated as a formula in a logical system--frequently the first order predicate calculus, with the definitions of the arithmetic operators assumed. Notice the assumption that the input state and the output state are all that matter to the users of the program. This is indeed usually the case, particularly if one looks upon any files the program uses as being components of the state.

2.2.2. Parallel Programs

Owicki[56,57], Lamport[41], Owicki and Gries[58,59], Levin[44], and Apt et al[4] have all looked at the problem of verifying the input-output state behavior of systems with concurrently executing processes. Each component process is specified by _its_ input-output state behavior, then the program implementing that process is shown to meet its specification. Finally, the _proofs_ of the component processes are combined to obtain a proof of the entire system. Owicki, Lamport, and Owicki and Gries treat systems of processes that share variables. In this case the combining criterion is non-interference between the proofs of the component processes. Levin and Apt et al treat the Communicating Sequential Processes mechanism proposed by Hoare[27]. Here, an additional combining criterion called satisfaction or cooperation is required.

The proofs of the component processes using these methods implicitly become a part of the specifications, since we saw in Example 1.1 that input-state output-state specifications cannot adequately characterize parallel systems. The assertions that occur in the proof are really stating the assumptions that are made about the environment in which the process must be run. This is unfortunate because it means that a specification for a process cannot exist independently of the process text and its proof.

The other major class of specification techniques for processes and systems of processes is that based on regular expressions. The earliest example of such a technique was Habermann's[22] path expressions. Originally, path expressions were proposed not so much as a specification technique, but more as a synchronization primitive that could be included in a programming language. A set of path expressions could be associated with a "type definition" similar to a SIMULA class[7]. The path expressions constrain the order in which processes can call the operations of the type.

Path expressions have found more use recently as specification languages. COSY[43] and DREAM[70] are examples of system design disciplines that have adopted path expressions or their derivatives as part of their specification languages. Recognition of the limited power of regular expressions has led to proposals for more powerful expressions to provide for more complicated synchronization disciplines.

In particular, direct extensions of path expressions such as Andler's predicate path expressions[1], Riddle's event expressions[69] and Shaw's flow expressions[72]. A path, event, or flow expression defines a language. As a specification, a path, event or flow expression means that at any time the execution history of specified program is a prefix of some string in the generated language. Flow expressions can generate languages containing infinite strings. Such strings are intended to imply that the specified program can run for infinite amounts of time. We infer the same property from a path or event expression that generates an infinite language of finite strings.

Flow and event expressions can be shown to be universal. That is, for any language that can be recognized by a Turing machine, there are a flow expression and an event expression, each of which can generate the language.

2.3. Verification

2.3.1. Sequential Programs

Several verification techniques exist for sequential programs. The Stanford PASCAL verifier[35] uses the verification condition generation technique, in which cuts are found in a program's flow graph, and a theorem of the form P(program variables at beginning of path) implies Q(program variables at the end of the path) is proved for each path. The proofs of these theorems may be aided by a mechanical theorem prover. The PL/CV[11] system takes a different approach, integrating the logic of the data domains (integers, arrays, and strings) with the programming logic. The input to this verifier is an argument that a program is correct (rather than just a program). The argument may contain reasoning both about program structure (*if-then-else*, *while-do*, etc.) and reasoning about the data domains. In fact, programs may themselves be used in proving theorems about the data domains. Appendix 1 contains a summary of the proof rules of PL/CV and an example of their use in a complete proof.

A third approach to verification is to reason using the functional semantics of programs. Milner's LCF [51] is an example of this approach. The logic provides a way to reason about sets of recursively defined functions (applicative programs), which is precisely what a functional semantics of an imperative or ALGOL-like program is.

2.3.2. Parallel Programs

Because of the complexity of their interactions, parallel programs are inherently more difficult to specify, write, and verify than are sequential programs. A verification technique for parallel programs needs to help control this complexity by giving proofs a manageable structure. Earliest approaches to verifying parallel programs showed that a program computed the correct answer regardless of the interleavings of its processes by considering all possible interleavings. This technique requires consideration of a number of different executions that is exponential in the size of the system (measured in number of statements).

Owicki[56] and Lamport[41] independently discovered that the proof of a parallel system could be structured so that a simple condition called *non-interference* between the proofs of the component processes guaranteed that a correct output was obtained from all interleavings. Checking proofs to see if they interfere is quadratic in the size of the proofs. Furthermore, the technique gives the programmer insight into how the processes can interfere with each other and into the assumptions that each process makes about its environment.

A system $S = S_1 // S_2$ satisfies {P} S {Q} if P implies $P_1 \wedge P_2$, $Q_1 \wedge Q_2$ implies Q, and the proofs of $\{P_1\}\ S_1\ \{Q_1\}$ and $\{P_2\}\ S_2\ \{Q_2\}$ are interference free. The two proofs are interference free if for each statement T in S_1 and each assertion A in the proof of $\{P_2\}\ S_2\ \{Q_2\}$, T does not falsify A. In proving that T does not falsify A, one may assume that the assertion immediately before T in the proof of $\{P_1\}\ S_1\ \{Q_1\}$ (called pre(T)) holds; therefore, the theorem to be proven is $\{pre(T) \wedge A\}\ T\ \{A\}$.

Levin[44] and Apt et al[4] have extended this method for use with Hoare's Communicating Sequential Processes. The reception of a value from another process is followed by an assertion about the value received. Such an assertion cannot be proved strictly within the confines of the process containing it, so the global requirements on the proofs of the component processes include a new condition called satisfaction or cooperation. The satisfaction criteria require that when a send statement in one process sends a value to another process the postcondition of the receive statement that receives the value is true. The fact that it is true must follow from the preconditions of the send and receive statements involved in the communication.

The other technique that has been used for showing correctness of parallel programs with respect to input-state output-state specifications is translation of parallel programs to non-deterministic sequential programs. Flon and Suzuki[17] and van Lamsweerde and Sintzoff[74] take this approach. The former deal with correctness of parallel programs while the latter are interested in the synthesis of parallel programs. Because the translation obscures any structure that might exist in the component processes, it seems that this approach can never be too helpful in giving people a better understanding of their programs. Nevertheless, it may be that a mechanical verification tool could take advantage of these results.

2.4. Summary

Parallel programs continue to challenge the capabilities of programmers with their complexity. Owicki's techniques made a significant difference in the way that people understood the interactions between parallel processes, but the preceding discussion has shown that introduction of new programming language features requires new verification techniques. One hopes that there is traffic in the other direction as well and that the goal of understanding programs well enough to be able to verify them leads to language features that support, rather than hinder, a programmer's understanding of his problems and the programs that solve them.

Chapter Three: A Simple Parallel Programming Language and Its Semantics

1. Introduction

This chapter presents a simple language for parallel programming based on message passing. The language was designed for the succinctness of its description rather than for succinctness of programs written in it. It is a vehicle for exploring techniques for reasoning about parallel programs that communicate entirely by passing messages. Thus, sequential programming language features (like procedures and pointers), which are useful but not particularly relevant to this discussion, are omitted.

The most difficult aspect of parallel program semantics has also been avoided by omitting variables shared between processes. The omission of the sequential constructs above can be justified on the grounds that they have been formally described and their inclusion is not relevant. The decision to omit shared variables is rather arbitrary, and serves mainly to delimit the boundaries of the problem being considered. Certainly, the same issues that are addressed here for parallel systems with message passing need to be addressed for systems with shared variables and for systems with both.

Initially we consider a programming language in which the messages are chosen from a finite set of uninterpreted message names. That is, processes communicate by agreeing that a message with a particular name has been sent. In such a language, there is no explicit send or receive operation, just a communicate operation, and the interpretation of a message name is entirely up to the processes that use it for communication. This symmetry between send and receive is unusual but not restrictive, since the interpretation of the communication of a particular message named "AB", for example, may be that process A has sent process B a particular value.

The restriction that messages are chosen from among a predefined finite set is also not really a restriction, since processes can encode long messages as sequences of short ones (in binary, say). Nevertheless, such an argument is unappealing when one is forced to write a real program and understand what it does. Hence such encodings do not fit very well into the spirit of this work. Chapter Six shows how to overcome this limitation.

3.1. A Simple Parallel Programming Language (SPPL)

The Simple Parallel Programming Language (SPPL) adds parallelism and communications features to a sequential programming language with null, assignment, alternative and repetitive statements. This section presents a formal syntax of the language together with an informal semantics in the style of the Algol60 report[54].

The syntax is presented using the following meta-notations: { $item_1$ § $item_2$ } represents a list of one or more $item_1$'s separated by $item_2$'s. [item] represents zero or one occurrence of the item.

The programming language is

```
<program> ::= { <process> § // }
```

A program is a system of processes that can run concurrently. All variables are local to the process in which they occur. If the same variable name occurs in more than one process the occurrences reference different variables.

<process> ::= process <statement> endprocess

Each individual process is just a statement. Notice that parallelism does not occur within a process.

<statement> ::= <lstatement> | <statement> ; <statement>

Every lstatement is labelled with a unique label. These labels are present only to make the semantics work and can be generated automatically for any program in which they are omitted.

```
<lstatement> ::= <label>: <sstatement>
<sstatement> ::= <variable> := expression
               | skip
               | if <boolean-expression>
                   then <statement>
                   else <statement>
                   fi
               | while <boolean expression> do
                      <statement>
                      od
```

The common sequential programming statements have their usual meaning.

```
<sstatement> ::= comm { <guard> → <statement> § □ } endcomm
<guard>      ::= [<boolean-expression>:] <identifier>
```

The comm statement is the only way that processes can communicate with one another. The guards of a comm statement determine which communications can be done by the process executing that statement. The boolean expression in a guard is optional, the default being true. The identifier appearing in the guard is a _communication symbol_. A given communication symbol may occur at most once in the guards of a particular _comm_ statement. The set of all communications symbols appearing in process P is the _communication alphabet_ of P, denoted by $\Sigma(P)$.

Informally, a _comm_ statement means: evaluate the boolean expressions of all of the guards. The process is _ready to communicate_ the symbols of the guards with true boolean expressions. When _all_ processes that have a particular communication symbol in their communication alphabet are ready to communicate that symbol, a communication may occur. The processes that communicated the symbol are not ready to communicate anything more until they execute another _comm_ statement, and they proceed to execute the statement corresponding to the guard of the symbol communicated. Note that if communication of more than one symbol is possible at some time, only one communication will occur. As a result of that communication, other communications that were possible may no longer be possible, since some processes have ceased being ready to communicate some symbols. However, processes that did not take part in the communication may still be ready to communicate other symbols in which case additional communications may occur. These communication rules may be implemented by supervisor program (_kernel_) in a single processor implementation of this language. In a multiple processor implementation distributed synchronization mechanisms such as those described in Schneider[71] and Levin[44] seem to be advantageous.

Two remarks are in order at this point. First, the _if_ and _while_ statements of this language are deterministic as in Algol 60, but the _comm_ statement is non-deterministic as in Dijkstra's[13] guarded command language. This apparent inconsistency and the restriction on multiple occurrences of a communication symbol in the guards of a given comm statement arise from a need to make processes deterministic in a sense that will be made precise in Chapter Four. Informally, determinism in this context means that given the input to a process (its initial state and the communications that it has performed) one can determine uniquely the next comm statement it will execute. The form of the comm statement, itself, is dictated by the need to express waiting for a communication of any of several kinds. Indeed, it is precisely this ability that distinguishes this language from that analyzed by Kahn and MacQueen[38]. Second, symbols that appear in the alphabet of exactly one process in a system are never communicated between the processes of a system, but can never cause a process to delay, according to the above description of the comm statement. Therefore, communication of these symbols may be regarded as input or output by the entire program.

3.2. The Formal Semantics of SPPL

The elementary formal semantics of SPPL is intended to capture our intuition about how programs in this language should work. Since this is the defining semantics it is important that it be reasonable. Henceforth it will be the final source of appeal as to what a program means. The meaning of each program of SPPL will be defined as a set of execution traces or, more accurately, a function mapping an initial state into a set of execution traces. As we will see later, such a semantics serves nicely as a foundation for more abstract definition techniques.

3.2.1. The Semantics of Program Traces

A trace, as used by Hoare[31], is a list of assignments and tests, to which we will add communications. The syntax of traces is defined by the following grammar:

<trace-el> ::= <label>: <command>

<command> ::= <variable> := <expression>
| skip
| <boolean expression>
| comm
| <identifier>
| ∇

<trace> ::= { <trace-el> § , }

A trace may be thought of as a straight line program that will abort if any of its tests are false. An occurrence of an <identifier> as an element of a trace corresponds to a communication of that identifier by the trace. The symbol ∇ is used at the end of a trace that represents a terminated execution of a program or program segment. It is very easy to give a functional semantics for traces. First of all, a trace is a program. We will regard it as a map from states to states, so the meaning function Mt will map traces into functions from states to states:

Mt: Traces $\rightarrow$ States $\rightarrow$ States

The notation follows Donahue[16] in enclosing syntactic arguments to meaning functions in [], while semantic arguments (such as states) are enclosed in (). The state domain is assumed to contain an undefined element $\perp_{States}$, which will be abbreviated $\perp$ when no confusion can arise. The $\perp$ state arises when a trace aborts.

Another mechanism we need to define the semantics of traces is a function to evaluate expressions. We will call this function Eval. It maps expressions and states into values. The interested reader should refer to [16] for a formal definition of Eval. Others will be satisfied with knowing that Eval is strict (i.e., bottom preserving) and that it gives the standard interpretation of boolean expressions; in particular, Eval(¬bexp, s) = ¬Eval(bexp, s).

We begin to define the functional semantics of traces:

$\underline{Mt}[\![\,]\!] = \underline{Mt}[\![\underline{skip}]\!] = \lambda s.\ s$

$\underline{Mt}[\![\text{label:command}]\!] = \lambda s.\ \underline{Mt}[\![\text{command}]\!]$

$\underline{Mt}[\![\underline{comm}]\!] = \lambda s.\ s$

The empty trace is assigned the identity function, as are the <u>skip</u> command and the <u>comm</u> command. Notice that the symbol <u>comm</u> is really unnecessary in traces, just as the symbols <u>while</u> and <u>if</u> are unnecessary. The symbol <u>comm</u> provides a handy reference point for some of the arguments in Chapter Four, so it is included in traces. If b is a test (boolean expression) then

$\underline{Mt}[\![b]\!] = \lambda s.\ \text{if Eval}(b,s) \text{ then } s \text{ else } \perp$

A test is assigned a function that is the identity on states where the test is true and that aborts where the test is false.

$\underline{Mt}[\![x{:=}e]\!] = \lambda s.\ \lambda\ id.\ \underline{\text{if}}\ id{=}x\ \underline{\text{then}}\ \text{Eval}(e,s)\ \underline{\text{else}}\ s(x)$

Assignment produces a new state, identical to s except at x, where the value is the result of the expression evaluation in s.

$\underline{Mt}[c] = \lambda s.\ s$

$\underline{Mt}[\nabla] = \lambda s.\ s$

Communication symbols and the end symbol have no effect on the state of a process. If c is a communication symbol, then

$\underline{Mt}[t_1 . t_2] = \lambda s.\ \underline{Mt}[t_2](\underline{Mt}[t_1](s))$

Compound traces are assigned the functional composition of their individual meanings.

3.2.2. Feasible traces of processes

A trace t is *feasible* for a state s if $\underline{Mt}[t](s) \neq \perp$. The feasible trace semantics of processes is a function from processes into functions from states to sets of feasible traces. The meaning function is $\underline{Mp}$: <statement> $\rightarrow$ state $\rightarrow 2^T$. The set of traces that is produced by this function is closed under the prefix operation i.e., if it contains a trace t, then it also contains all prefixes of t, including the empty trace, ϵ. For convenience in defining $\underline{Mp}$ we will use the prefix closure operator $\leq*$: $T \rightarrow 2^T$ defined by

$\leq* t = \{\ t' \mid t' \text{ is a prefix of } t\ \}$

and append: $T \times T \rightarrow T$ defined by

append(s,t) $\equiv$ if s = x,∇ then x,t else s.

The feasible traces of an assignment statement comprise the prefix closure of the trace consisting of the assignment and the termination symbol and the feasible traces of a *skip* statement comprise the prefix closure of the trace consisting of *skip* and the termination symbol.

$\underline{Mp}[L\!: x\!:=\!e] = \lambda s.\ \leq*(L\!: x\!:=\!e, \nabla)$

$\underline{Mp}[L\!: \underline{skip}] = \lambda s.\ \leq*(L\!: \underline{skip}, \nabla)$

Notice that in neither of these cases does the set of traces depend on the initial state.

The traces of the conditional statement depend on the value of the boolean expression.

$\underline{Mp}[$L: $\underline{if}$ b $\underline{then}$ $stmt_1$ $\underline{else}$ $stmt_2] = \lambda s.$

if Eval(b,s)

then (≤*(L: b,∇)) ; $\underline{Mp}[stmt_1](s)$

else (≤*(L: ¬b,∇)) ; $\underline{Mp}[stmt_2](s)$

The traces of the repetition statement are recursively defined in terms of single executions of the statement.

$\underline{Mp}[$L: $\underline{while}$ b $\underline{do}$ stmt $\underline{od}] = \lambda s.$

if Eval(b,s)

then ≤*(L: b,∇) ; $\underline{Mp}[$stmt; L: $\underline{while}$ b $\underline{do}$ stmt $\underline{od}](s)$

else ≤*(L: ¬b,∇)

Statement composition is concatenation of the two trace sets, maintaining prefix closure.

$\underline{Mp}[stmt_1;\ stmt_2] = \lambda s.$

{ append(x,y) | x∈$\underline{Mp}[stmt_1](s)$

and y∈$\underline{Mp}[stmt_2](\underline{Mt}[x](s))$ }

A $\underline{comm}$ statement is described by the union of the trace sets of its branches. If all the boolean conditions of a $\underline{comm}$ statement are false, the statement is equivalent to a $\underline{skip}$. To make it equivalent to an abort instead, we would simply delete the last three lines of the definition of the feasible traces, leaving the trace set empty in that case.

$$
\begin{array}{l}
\underline{Mp}[\![L: \underline{comm}\ b_1:i_1 \rightarrow stmt_1 \\
\qquad \Box \quad b_2:i_2 \rightarrow stmt_2 \\
\qquad\qquad \cdot \\
\qquad\qquad \cdot \\
\qquad\qquad \cdot \\
\qquad \Box \quad b_n:i_n \rightarrow stmt_n \\
\quad \underline{endcomm}]\!] = \\
\quad\quad \lambda s. \\
\qquad\quad \bigcup_{j=1}^{n} \text{ if } Eval(b_j, s) \\
\qquad\qquad \text{then } (\ \leq^*(L{:}\ \underline{comm}, L{:}\ b_j, i_j, \nabla)\ ;\ \underline{Mp}[\![stmt_j]\!](s)\ ) \\
\qquad\qquad \text{else } \emptyset \\
\qquad\quad \cup \text{ if } \bigwedge_{j=1}^{n} Eval(\neg b_j, s) \\
\qquad\qquad \text{then } \leq^*(L{:}\underline{comm}, L{:}\ \bigwedge_{j=1}^{n} \neg b_j, \nabla) \\
\qquad\qquad \text{else } \emptyset
\end{array}
$$

3.2.3. The relational semantics of processes

The relational semantics of processes is not of much interest to us in reasoning about programs with parallel processes, but it does serve to point out the utility of the trace semantics as a tool for developing relational and functional semantics for sequential programming languages.

In deriving a relational or functional semantics from the trace semantics, rather than stating it directly, we have simply replaced composition of functions from statements into functions from states to states, with composition of the trace meaning function and the trace producing function. There is no particular benefit in doing so in the case of purely sequential programs, but for processes the retention of the information in the traces (the statements that are executed in producing a state) is very useful.

Just to show how closely connected the relational and trace semantics are, the relational semantics of statements can be stated in one line:

$$\underline{Mpr}[\![stmt]\!] = \{ (s_i, s_o) \mid \exists(t,\nabla)\in\underline{Mp}[\![stmt]\!](s_i).\ \underline{Mt}[\![t]\!](s_i) = s_o \}$$

The functional semantics is just as easily stated for statements containing no comm statements:

$$\underline{Mpf}[\![stmt]\!] = \lambda s.\ \underline{Mt}[\![t]\!](s)$$

$$\text{where } (t,\nabla) \text{ member } \underline{Mp}[\![stmt]\!](s)$$

If there is no trace ending in ∇ then the program diverges so $\underline{Mpf}[\![stmt]\!](s)$ should be $\perp$.

3.2.4. Feasible traces of parallel programs

This section develops trace semantics for systems of processes. The semantics of individual processes has already been presented but nothing in that definition tells us anything about how processes interact with one another. What remains to be done is to formally define the interactions that occur at comm statements.

Let us call the set of all trace-el's that can occur in the traces of process P the trace alphabet of P and denote it by $\tau(P)$. Let $S \equiv P_1 \,//\, P_2 \,//\, \cdots \,//\, P_n$ be a system of processes. Let $T_1, T_2, \cdots, T_n$ be the corresponding meaning functions (i.e., $T_i = \underline{Mp}[\![P_i]\!]$), $\tau_1, \tau_2, \cdots, \tau_n$ the corresponding trace alphabets, and $\Sigma_1, \Sigma_2, \cdots, \Sigma_n$ the corresponding communications alphabets. Notice that $\tau_i \sqcap \tau_j$ is $\Sigma_i \sqcap \Sigma_j$ whenever $i \sim= j$ (because all elements of the trace alphabet of a process that are not in the communication alphabet are tagged with a label unique to the process).

Let $\downarrow: \Sigma^* \times 2^\Sigma \to \Sigma^*$ be the <u>projection</u> of its first argument onto its second. That is $\downarrow$ is an erasing homomorphism that deletes all symbols from its first argument that do not appear in the second. Then the meaning function for systems is defined by

$$\underline{Mp}[S] = \lambda \begin{bmatrix} s_1 \\ \cdot \\ \cdot \\ \cdot \\ s_n \end{bmatrix} . \{s \in (\bigcup_{i=1}^{n} \tau_i)^* \mid \bigwedge_{i=1}^{n} s \downarrow \tau_i \in T_i(s_i)\}.$$

Therefore, each process is computing according to one of the traces in its own trace set, but it is also unable to execute some of the traces because the other processes are unwilling to communicate at the proper times. Furthermore, since a communication symbol is not marked by a label the definition of <u>Mp</u> requires that all processes are able to communicate it whenever it occurs in a trace of the system.

3.2.5. <u>Example of a System</u>

Consider the two processes in Figure 3.1. The first loop of process BC can be executed an arbitrary number of times, producing a "b" each time, if the process is executed by itself; the second loop is then executed one fewer times than the first, communicating d's. But an execution of the system consisting of processes BC and B executes the first loop of BC exactly eight times and the second exactly seven, since the communications of b's and d's must be coordinated between the processes. Process BC is free to communicate c's because c is not in the communication alphabet of process B.

```
process B:                         process BC:
    i := 7;                            i := 0;  morebs := true;
    while i>0 do                       while morebs do
        comm                               comm
            b → i := i-1;                      b → i := i+1
        endcomm                             □  d → morebs := false
    od;                                    endcomm
    comm d → skip endcomm              od;
endprocess                             while i>0 do
                                           comm
                                               c → i := i-1
                                           endcomm
                                       od
                                   endprocess
```

Figure 3.1 The Effect of Including a Process in a System

3.3. Conclusion

SPPL captures the most interesting feature of Hoare's Communicating Sequential Processes proposal, namely synchronization in multi-process sytems by unbuffered communication. The formal description in this chapter is reasonably intuitive but it seems inconvenient as a tool for reasoning about processes, hiding their interesting behavior--their communications properties--in the details about their implementation. The next chapter addresses this issue.

Chapter Four: Program Semantics and Specifications

4. Introduction

The semantics of SPPL given in Chapter Three is concrete. It indicates exactly how a process works. This chapter explores a semantics for SPPL that relates a process to what it does; that is, one limited to describing the interaction between a process and its environment without describing the details of the implementation. The interactions of a process with its environment are the key to specifying and understanding it. This is true regardless of the formalisms and programming languages that are used.

For sequential programs the interactions with the environment are easily identified (if not easily described). A program is executed beginning in input state s_i, and eventually terminates in some output state, s_o. A semantics assigns meaning to the program by relating s_i to s_o. Recall from Chapter Two that the common ways of doing so are to describe the function, F, of s_i that the program computes--that is $s_o=F(s_i)$--and to establish the truth of a formula $P(s_i) \Rightarrow Q(s_o)$.

The interaction abilities of a process intended to run concurrently with others are not so easily identified. Instead of having only two points of interaction with its environment, the beginning and the end, a process interacts with others whenever it accesses a shared variable or sends or receives a message. Because SPPL uses message passing, exclusively, for interprocess communications it lends itself to a simple, easy to understand formal semantics that clearly exposes a process's interactions with its environment. Furthermore, systems of processes in SPPL have essentially the same interface to their environment as processes; so the formal semantics extends naturally to them as well.

The utility of the more abstract semantics comes from our ability to use it in specifying the behavior of processes and systems. Therefore, a specification technique is presented slong with the semantics. As one would expect, the development of the semantics was significantly influenced by the specification technique and vice versa, and both, in turn, influenced the design of SPPL. Section 4.1.4 discusses these issues.

In section 4.2 we turn to the problems posed by semantic characterization and specification of systems of processes. The particular problem of sub-systems whose internal communications are hidden from the rest of a system is examined, and results proven for the case when the sub-systems are "well behaved".

4.1. A more abstract semantics

The semantics given in this chapter captures the input-output properties of processes and systems without mentioning their internal structure the way the trace semantics of Chapter Two did. The communication of a symbol by a process requires that other processes in the system communicate the symbol as well. Communications, therefore, cannot be hidden in our semantics, but are in fact the core of it. In the semantics given in the next section, all of a process's activity, except its communications, are hidden.

4.1.1. The language of a process

For any process P we let $L_s(P)$ be the *language of P* started in state s, defined by

$$L_s(P) = T_s(P) \downarrow \Sigma(P)$$

where $\downarrow : 2^{\Sigma^*} \times 2^{\Sigma} \rightarrow \Sigma^*$ is the obvious extension of $\downarrow$ to sets:

$$T \downarrow \Sigma = \{ t \downarrow \Sigma \mid t \in T \}.$$

We will henceforth assume that execution of process P always starts in a particular state s_P and abbreviate $L_{s_P}(P)$ by L(P). L(P) contains all the strings that can be communicated by executing a finite number of steps of P starting in state s_P. Furthermore, if $h \in L(P)$ and $t \in T(P)$ and $t \downarrow \Sigma(P) = h$ we say that h *represents* t in L(P).

If one observes the communications that process P does while interacting with other processes, the ordered sequence will always be a member of L(P). Since processes interact with one another using only these explicit communications, the language of a process really is a description of its input and output properties.

Is the language of a process a good enough description of it to let us describe reasonable properties of the process and determine whether or not they hold? The answer to this question is a qualified yes. We will show later in this chapter that when certain desired properties of P are naturally stated as formulae involving L(P), the formulae are true only if P has the desired property; but we also show that this *consistency* is critically dependent on the restrictions made in SPPL regarding uniqueness of communication symbols in *comm* statements. The next section relates some potentially interesting properties of processes to formulae involving the languages of the processes.

Sufficient conditions on processes are developed so that the inconsistency mentioned above does not arise for these properties. Finally, the theory is extended to systems of processes.

4.1.2. Process Specifications

From the point of view of a system designer, processes are tools used in decomposing large, difficult problems into smaller problems of manageable complexity. When the subproblems are correctly specified, their correctness should imply the correctness of the entire system.

The specification of a process is a description of its language. A complete specification is one that allows us to determine from the specification whether or not a given string is or is not in the language of any process that meets the specification. Complete specifications are not always necessary to solve interesting problems, as later examples show. Indeed, undecidability theory tells us that the language of a process that communicates an arbitrary Turing machine index (suitably encoded) followed by a '1' if that Turing machine halts on blank tape and a '0' otherwise, can have no complete specification. On a more practical level, however, a designer starts with a rough idea of a process's role in the system. Insights into the structure of the problem point to refinements of these rough specifications until they are sufficient to prove the correctness of the system.

A grammar could serve as a process specification, as could a Turing machine, but in many cases, it seems that both are likely to be derived from a higher level specification. Consider, for example, a general semaphore, with initial value k. Informally, it is specified by saying that the semaphore blocks P operations until the number of previously completed V operations plus the initial value, less the number of previously completed P operations, is non-negative. One seems far more likely to understand this description than the more formal, more accurate, but more or less incomprehensible grammar description of the language:

$$G=(\{s,c\}, \{P,V\}, \{s\to c^k, c\to P, c\to Vc\}, s)$$

The latter is more accurate, because it guarantees that the language contains arbitrarily long strings, while the informal description does not. The intuitiveness of the informal description, though, makes it worthy of further pursuit. We ask how informal specifications can be formalized, and what needs to be added to them to express such desirable properties as the fact that a language contains arbitrarily long strings.

What kinds of properties are important in specifying the behavior of processes and systems? First we recognize that a process may be constrained to act only in certain ways. Some behaviors are disallowed. These are the so-called safety properties of its language. They can be expressed by a formula with the form

$$\forall h\in L(P).\ Q(h)$$

where Q is a Boolean formula containing no quantifiers over L(P). A safety specification, or language invariant, insures that P does not misbehave. It deals individually with the strings in the language (since it contains only one quantified variable ranging over L(P)), and is trivially satisfied by the empty language. Hence it can be satisfied by a process that does <u>no</u> communications at all] We need more than safety specifications to adequately characterize process behavior, just as we need more than partial correctness specifications to adequately characterize sequential program behavior. The additional specification of sequential programs is one that says they terminate. What property of processes is analogous to termination of sequential programs? Surely not termination. Non-terminating processes are an important part of many systems; even if termination is required, a process that terminates prematurely is just as erroneous as one that doesn't terminate at all. (Termination or local looping by a single process can be looked at as a local deadlock]) Instead, a process should be able to service the requests that will be made of it in the context of the system in which it runs.

Ultimately, a system is the servant of its human users, whose communications with the system may be modelled by processes. A user communicates information to the system and expects results. There is no need for the system to terminate upon producing these results; indeed, the very essence of interactive computing is the ability of the computing system to produce intermediate results that guide the future behavior of users in making requests of the system.

The ability to service requests is expressed in a liveness specification. Informally, a liveness specification says that if certain input has been received by a process or system, then it must be able to eventually produce certain output. The process may or may not be required to produce the output immediately. If the output is not required immediately then there is the chance for later input to suppress the output. Of course, in SPPL input and output are not distinguishable, so the liveness specifications can be reduced to the form:

$\forall h \in L(P).\ [Q(h) \Rightarrow \exists h' \in L(P).\ (h' > h \wedge Q'(h,h'))].$

A specification formula is either a safety specification formula or a liveness specification formula. It is wise now to insure that we understand the meaning of these specifications by giving their interpretations in terms of traces. A safety specification

$\forall h \in L(P).\ Q(h)$

is to be interpreted as

$\forall t \in T(P).\ Q(t \downarrow \Sigma(P))$

That is, any communication that a trace can give rise to must satisfy the invariant. Liveness specifications require considerably more care in interpretation. Informally, the specification

$\forall h \in L(P).\ [Q(h) \Rightarrow \exists h' \in L(P).\ (h' > h \wedge Q'(h,h'))].$

means that whenever an execution of P has communications history h satisfying Q(h), the execution must be able to go on to communicate h' satisfying Q'(h',h). If history h can arise from more that one trace then this must be true of all such traces. The obvious interpretation of the specification is

$$\forall t \in T(P).\ [Q(t \downarrow \Sigma(P)) \Rightarrow$$
$$\exists t' \in T(P).\ (t' \downarrow \Sigma(P) > t \downarrow \Sigma(P) \wedge Q'(t \downarrow \Sigma(P),\ t' \downarrow \Sigma(P)))]$$

but of course this is incorrect, since it allows a program to misbehave on traces that are represented by histories that represent other traces as well. Therefore, the intended interpretation of a liveness specification in the form shown above is

$$\forall t \in T(P).\ [Q(t \downarrow \Sigma(P)) \Rightarrow \exists t' \in T(P).\ (t' > t \wedge Q'(t \downarrow \Sigma(P),\ t' \downarrow \Sigma(P)))].$$

Section 4.1.4 and Theorem 4.1 show that the determinacy constraints made on SFPL insure the consistency of a specification and its interpretation.

4.1.3. Examples of process specifications

This section is a collection of small examples of process specifications. For the most part, they are standard examples from the literature of concurrent programmming.

4.1.3.1. Binary Semaphore B

Although the language used here does not have shared variables one can conceive of the need for one process to have exclusive ability to communicate with some other process for awhile. If such a situation arises we know that the power of semaphores is sufficient to synchronize accesses to such shared processes. A process B implementing a binary semaphore with initial value 1 and used to synchronize n processes can be specified as follows:

$$\Sigma(B) = \{P_1, V_1, P_2, V_2, \ldots, P_n, V_n\}$$

$$\forall h \in L(B).$$

$$0 \le |h \downarrow P| - |h \downarrow V| \le 1.$$

where $P = \{P_1, P_2, \cdots, P_n\}$ and $V = \{V_1, V_2, \cdots, V_n\}$. This is the safety specification of the binary semaphore. In addition we specify that process B has liveness properties

$$\forall h \in L(B). \forall i,j \in (1..n).$$

$$hP_iV_j \in L(B) \vee hV_iP_j \in L(B)$$

Notice that the above specifications imply $L(B) = (PV)^* + (PV)^*P$, but in general we don't require a specification to tell us exactly what L(B) is -- only some of its properties.

An interesting variant of the binary semaphore is a process that enforces the restriction that no process using the semaphore may do a V operation without having previously done a corresponding P operation.

The variant binary semaphore B' uses all of the specifications of B. In addition it is required that

$\forall h \in L(B').\ \forall i \in (1..n).$

$$0 \leq |h \downarrow P_i| - |h \downarrow V_i| \leq 1.$$

In many applications the variant binary semaphore is interchangeable with the original because the users of the semaphore already impose the "P then V" discipline.

4.1.3.2. General Semaphore G (with initial value k)

The safety specification requires that the number of V's plus the initial value not be exceeded by the number of P's at any time.

$\Sigma(G) = \Sigma(B)$

$\forall h \in L(G).\ 0 \leq |h \downarrow V| + k - |h \downarrow P|$

The liveness specification is that a V must be accepted at any time and a P must be accepted whenever allowed by the safety specification.

$\forall h \in L(G).\ \forall i \in (1..n).$

$$hV_i \in L(G) \wedge$$

$$0 < |h \downarrow V| + k - |h \downarrow P| \Rightarrow hP_i \in L(G)$$

4.1.3.3. Readers and writers allocator process. RW

Suppose several processes read and write a common data base. Any number of processes may simultaneously read the data base but no process may read while another is writing and no two processes may simultaneously write in the data base. Specify a process that can mediate between these competing processes and insure that the above constraints hold.

Each user process, P_i, is assumed to have a language of the form $(sr_i\ x\ er_i + sw_i\ y\ ew_i)^*$ where x and y represent strings from the language of the shared database. (sr, er, sw, and ew are mnemonics for "start read", "end read", "start write", and "end write" respectively.) Of course, the processes may perform other communications, but we assume that those symbols are not in the alphabet of either the database or the allocator process.

The situation is as pictured in Figure 4.1. The customer processes collectively form the environment of the allocator process RW. The assumption above about the behavior of the user processes is expressed by the specification of the environment (called ENV)

$$\forall h \in L(ENV),\ i \in (1..n).$$
$$[0 \le |h{\downarrow}sr_i| - |h{\downarrow}er_i| \le 1 \wedge$$
$$0 \le |h{\downarrow}sw_i| - |h{\downarrow}ew_i| \le 1]$$

The safety specification of the allocator is

$$\forall h \in L(RW),\ i,j \in (1..n).\ \{\ h{\downarrow}\Sigma(ENV) \in L(ENV) \Rightarrow$$
$$[\ ((|h{\downarrow}sr_i| - |h{\downarrow}er_i| \ne 0) \Rightarrow$$
$$(|h{\downarrow}sw_j|-|h{\downarrow}ew_j| = 0))$$
$$\wedge\ ((|h{\downarrow}sw_i| - |h{\downarrow}ew_i| \ne 0) \Rightarrow$$
$$(\ (i \ne j \Rightarrow (|h{\downarrow}sw_j| - |h{\downarrow}ew_j| = 0))$$
$$\wedge\ (|h{\downarrow}sr_j| - |h{\downarrow}er_j| = 0)))]\}$$

The liveness specification of the allocator is

$$\forall h \in L(RW).$$

$$[\ (\ \sum_{i=1}^{n} (|h{\downarrow}sr_i| - |h{\downarrow}er_i|)) \ne 0 \Rightarrow$$
$$\forall j \in (1..n).\ h\ sr_j \in L(A)$$

$$\wedge\ (\ \sum_{i=1}^{n} (|h{\downarrow}sr_i| - |h{\downarrow}ew_i| + |h{\downarrow}sr_i| - |h{\downarrow}er_i|)) = 0 \Rightarrow$$
$$\forall j \in (1..n),\ h\ sw_j \in L(A)$$

$$\wedge\ \forall i \in (1..n),\ (h\ ew_i \in L(A) \wedge h\ er_i \in L(A))]$$

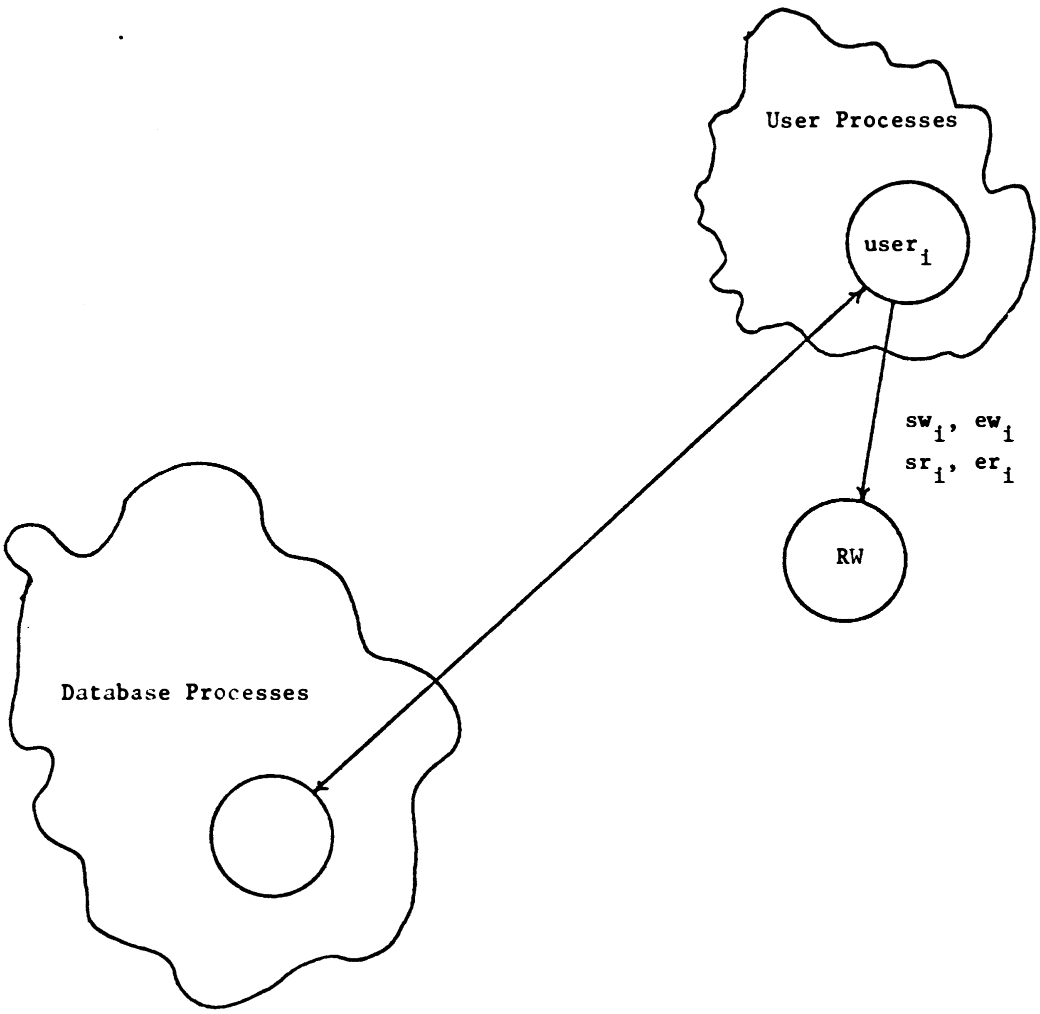

Figure 4.1

A simple database system

4.1.4. Consistency of the Language Semantics for Process Specifications

In the previous section we gave an intended interpretation for process specification formulae in terms of formulae about trace sets. This section constitutes a proof that whenever a process specification formula is true its intended interpretation is true as well. This is the key to being able to use a process's language as a meaningful description of its behavior. Recall that the intended interpretation of a safety specification

(1) $\forall h \in L(P).Q(h)$

is

(2) $\forall t \in T(P).Q(t \downarrow \Sigma(P))$

The consistency theorem is $(1) \Rightarrow (2)$ for any formula Q. The consistency theorem for safety specifications is proved by observing that every $t \in T(P)$ is represented in L(P) by $t \downarrow \Sigma(P)$.

The consistency proof for liveness specifications is not so easily come by. In fact, it is the justification for the restrictions made on the communications symbols appearing in the guards of <u>comm</u> statements in SPPL as well as the choice of the deterministic forms of <u>if</u> and <u>while</u>.

Let us first examine some of the consequences of violation of these rules. This will lead to some insight into what the real requirements are and point out the reasons for the choices made in SPPL. Let BAD be the process:

```
BAD: comm
          a -> L2: comm b -> L4: skip endcomm
       □  a -> L3: comm c -> L5: skip endcomm
     endcomm
```

in which the restriction of SPPL that guards in a <u>comm</u> statement must be distinct is violated. The traces of BAD are

$\le$*(BAD:<u>comm</u>, a, L2:<u>comm</u>, b, L4:<u>skip</u>, ∇) $\cup$

$\le$*(BAD:<u>comm</u>, a, L3:<u>comm</u>, c, L4:<u>skip</u>, ∇)

and L(BAD) is

$\{\epsilon, a, ab, ab\nabla, ac, ac\nabla\}$.

Now consider the liveness specification

$F = \forall h\in L(BAD). [h=a \Rightarrow \exists h'\in L(BAD). (h'>h \wedge h'=ab)]$.

F is true but the intended interpretation of F is false since

(BAD:<u>comm</u>, a, L3:<u>comm</u>)$\downarrow\Sigma$ = a

and there is no extension of this trace communicating a 'b'. Thus we see by counterexample that a specification technique based on L(P) and process specification formulae will not work, in general, for nondeterministic processes. On the other hand we do not want to use trace sets as our tool for specifying and reasoning about processes. They are clumsy to use in specifications. Furthermore, we want to insure that we can only talk about the external behavior of a process in a specification. Traces give access to all the details of the implementation of the process.

The inconsistency for the process specification formula F for BAD arises from the representation of two traces in T(BAD) by a single string in L(BAD). There is no problem when one of the traces is an extension of the other. Indeed, this happens all the time even when processes do not violate the rules of SPPL. However, when a string in a process's language represents two incomparable traces, consistency for process specification formulae is jeopardized. The following restriction is therefore placed on processes: every process must be <u>deterministic</u> in the sense that

$\forall t_1 . t_2 \in T(P).\ t_1 \downarrow \Sigma(P) < t_2 \downarrow \Sigma(P) \Rightarrow t_1 < t_2.$

In SPPL this restriction is met by insisting that the guards in a given comm statement must all have different communication symbols. Note that it would be sufficient to require that the boolean expressions of guards with identical communication symbols be mutually unsatisfiable. Since this is, in general, an undecidable property it is mentioned only for completeness. The syntactic restriction on communication symbols in guards is adequate and easy to live with since two guards with the same symbol can easily be rearranged into a single guard with that symbol followed by an alternative statement choosing between the two statement lists involved. Admittedly (and intentionally), this removes potential nondeterminism from the program.

At this point we are ready to prove the consistency theorem for processes:

Theorem 4.1:

If P is a deterministic process, then for any process specification formula F, $F(P) \Rightarrow Int[F(P)]$.

Proof:

Assume F is a safety formula $\forall h \in L(P).\ Q(h)$

Assume $\forall h \in L(P).\ Q(h)$

arbitrary $t \in T(P)$

$t \downarrow \Sigma(P) \in L(P)$

(definition of L(P))

$Q(t \downarrow \Sigma(P))$

$\forall t \in T(P).\ Q(t \downarrow \Sigma(P))$

$F \Rightarrow Int(F)$

Assume F is a liveness formula

$\forall h_1 \in L(P).\ [Q_1(h) \Rightarrow \exists h_2 \in L(P).\ (h_2 > h_1 \wedge Q_2(h_1, h_2))]$

Assume $\forall h_1 \in L(P).\ [Q_1(h) \Rightarrow \exists h_2 \in L(P).\ h_2 > h_1 \wedge Q_2(h_1, h_2))]$

arbitrary $t_1 \in T(P)$ where $Q_1(t_1 \downarrow \Sigma(P)$

$\exists h \in L(P).\ h > (t_1 \downarrow \Sigma(P)) \wedge Q_2(t_1 \downarrow \Sigma(P), h)$

(assumed liveness formula)

choose $h \in L(P).\ h > (t_1 \downarrow \Sigma(P)) \wedge Q_2(t_1 \downarrow \Sigma(P), h)$

$\exists t_2 \in T(P).\ t_2 \downarrow \Sigma(P) = h$

(definition of L(P))

choose $t_2 \in T(P).\ t_2 \downarrow \Sigma(P) = h$

$t_2 > t_1$

(determinism of P)

$\forall t_1 \in L(P).\ [Q_1(t_1 \downarrow \Sigma(P)) \Rightarrow$
$\exists t_2 \in T(P).\ (t_2 > t_1 \wedge Q_2(t_1 \downarrow \Sigma(P), t_2 \downarrow \Sigma(P)))]$

$F \Rightarrow Int(F)$

QED

4.2. Language Semantics of Systems and Subsystems

Just as we needed a more abstract semantics for processes, we need one for systems of processes. The same insights are applicable: we don't want to concern ourselves with the details of the statement interleavings of the component processes when we talk about a system's behavior. The goal, of course, is to assign meaning to the system by looking only at the meaning assigned to the component processes. If this cannot be done then the semantics of processes is not useful.

This section extends the concept of "language of a process" to "language of a system". The strings in the language of a system are the potential communications histories for that system. The first situation considered is the one in which the language represents all the communications among the processes of the system and from the system to the environment. Next attacked are the difficulties that arise when some set of processes is considered to be a closed subsystem whose internal communications are not visible.

4.2.1. Systems of Processes

The language of a system $S = P_1//P_2//\cdots//P_n$ is denoted $L(S)$ and defined by $L(S) \equiv \{t\downarrow\Sigma(S) \mid t\in T(S)\}$ in analogy with the definition of $L(P)$ for a process P. The following theorem shows that the language of the system is defined in terms of the languages of its component processes.

Theorem 4.2: System Representation

If $S = P_1//P_2//\ldots//P_n$ and

$$\bar{L}(S) = \{ h \mid \bigwedge_{i=1}^{n} (h\downarrow\Sigma(P_i))\in L(P_i) \wedge h\in(\Sigma(S))^{*}\}$$

then $\bar{L}(S) = L(S)$.

There are two parts to the proof of this theorem. First we show that $L(S) \subseteq \bar{L}(S)$.

Proof:

$$L(S) = \{t\downarrow\Sigma(S) \mid t\in T(S)\}$$

$$= \{t\downarrow\Sigma(S) \mid \bigwedge_{i=1}^{n}(t\downarrow\tau_i\in T_i) \wedge t\in(\bigcup_{i=1}^{n} \tau_i)^{*}\}$$

(by the definition of T(S))

arbitrary $t\in T(S)$

$$\bigwedge_{i=1}^{n} (t\downarrow\tau_i)\in T_i$$

$$\bigwedge_{i=1}^{n} (t\downarrow\tau_i)\downarrow\Sigma_i\in L_i$$

$$\bigwedge_{i=1}^{n} (t\downarrow\Sigma_i)\in L_i$$

$$t\downarrow\Sigma(S)\in(\Sigma(S))^{*}$$

$$t\downarrow\Sigma(S)\in L(S)$$

$L(S) \subseteq L(S)$

QED

The second part of the proof shows that $\bar{L}(S) \subseteq L(S)$ by constructing a trace t in T(S) from an arbitrary history h in $\bar{L}(S)$ such that

$t{\downarrow}\Sigma(S) = h$. The basic difficulty to be overcome in this proof is to show that there is an execution of each process that is able to communicate the symbols in the order called for by h. We will construct a trace for each process and show that they can be merged to produce the required trace. The proof consists of an algorithm to construct the desired trace.

Proof:

arbitrary $h \in \overline{L}(S)$
Use the algorithm of Figure 4.2 to construct a trace $t \in T(S)$
such that $t{\downarrow}\Sigma = h$.
The algorithm is correct by the argument below.
QED

Refer to Figures 4.2 and 4.3. The algorithm in Figure 4.2 correctly computes the required trace if the loop maintains the invariant

$$(s{\downarrow}\Sigma)g = h \wedge (s{\downarrow}t_i)s_i = t_i \wedge (s{\downarrow}\Sigma_i)(s_i{\downarrow}\Sigma_i) = h_i$$

illustrated in Figure 4.3. When the loop terminates, all of the s_i are empty so $s{\downarrow}\Sigma = h \wedge s \in T(S)$. The loop must terminate because at least one of the s_i is shortened on each iteration. The <u>else</u> branch of the <u>if</u> statement contains a proof that the invariant continues to hold when that branch is executed. The <u>then</u> branch of the <u>if</u> statement corresponds to the case where no communication is done. It obviously preserves the invariant.

```
g := h;
for each i ∈ 1..n do
    h_i := h↓Σ_i;
    h_i∈L_i
    let t_i be an element of T_i such that t_i↓Σ_i = h_i;
        (such t sub i exists by the definition of L_i)
    s_i := t_i
    od
s := ε;
while any s_i is non-empty do
    if any s_i starts with a symbol other than a symbol from Σ_i
    then
        choose such an s_i;
        s := s first(s_i);
        s_i := rest(s_i);
        assert invariant
    else
        v := first(g);
        ∀i∈ 1..n. v∈Σ_i ⇒ first(s_i) = v
            proof:
                arbitrary i ∈ 1..n where v∈Σ_i
                (s↓t_i)s_i = t_i
                    (invariant)
                ((s↓t_i)s_i)↓Σ_i = t_i↓Σ_i = h_i
                    (application of ↓ function to equal quantities)
                (s↓Σ_i)(s_i↓Σ_i) = h_i        (*)
                    (distributivity of ↓ over concatenation)
                we also know that
                (s↓Σ)g = h
                    (invariant)
                ((s↓Σ)g)↓Σ_i = h↓Σ_i = h_i
                    (application of ↓ to equal quantities)
                (s↓Σ_i)(g↓Σ_i) = h_i        (**)
                    (distributivity)
                (g↓Σ_i) = (s_i↓Σ_i)         (***)
                    (* and **)
                first(s_i) = v
                    (first(g) = v∈Σ_i ∧ first(s_i)∈Σ_i and ***)
```

Figure 4.2 Trace Construction

(continued)

```
                QED
            g := rest(g);
            s := sv;
            for each i such that first(s_i) = v do
                s_i := rest(s_i)
                od
            assert invariant
        fi
    od
```

Figure 4.2 Trace Construction (continued)

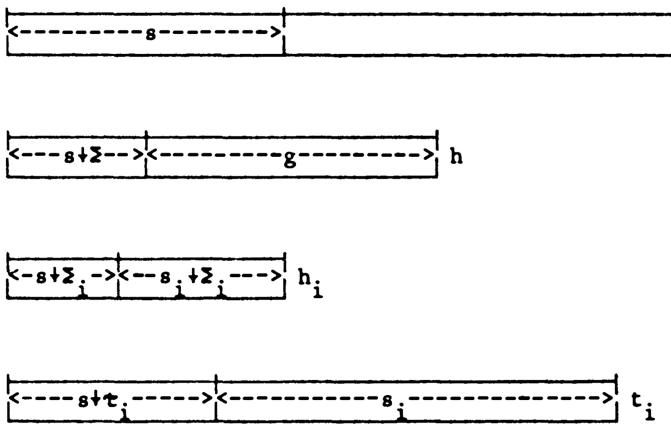

Figure 4.3 Invariant for Loop in Figure 4.2

The above theorem is central to this thesis. It says that if we know the language of each process in a system of processes we need not ever consider their trace semantics in order to derive properties of the entire system. The examples have indicated that in some instances the language of each process doesn't need to be known completely in order to derive interesting properties of the system.

The consistency theorem together with the system representation theorem indicate that the languages of processes and systems may not be an unreasonable way to approach the problems of specification and verification that these programs pose. Notice that the consistency theorem for systems has not yet been proven. This is deferred until subsystems have been discussed since it is just a special case of the consistency theorem for subsystems.

4.2.2. Example of a System

Frequently, set of processes need to share a resource process according to a mutual exclusion discipline. We will call the part of each process that requires access to the resource the critical section. Suppose that each process has the form shown in Figure 4.4.

```
M_i: while true do
          noncritical;
          comm
              P_i -> critical;
                    comm
                        V_i → skip
                    endcomm
          endcomm
     od
```

Figure 4.4 Cyclic Process with a Single Critical Section

Observe that an execution of process M_i is in a critical section if and only if the communications history of the execution contains more P_i's than V_i's. Then the mutual exclusion requirement for a system $S = (M_0//M_1//\cdots//M_{n-1})$ is expressed by the system specification

$$\forall h \in L(S).$$
$$\sum_{i=0}^{n-1} (|h\downarrow P_i| - |h\downarrow V_i|) \le 1.$$

Experience with such problems leads us to expect that a binary semaphore should be able to insure the desired synchronization and indeed this is the case. Referring back to our earlier specification of the binary semaphore we find that

$\forall h \in L(B).$

$0 \le |h{\downarrow}P| - |h{\downarrow}V| \le 1.$

Because $\sum_{i=0}^{n-1} |h{\downarrow}P_i|$ equals $|h{\downarrow}P|$ only one process at a time can be in its critical section. Notice that either the binary semaphore, its variant requiring each V to come from the same process as the preceding P, or a general semaphore with initial value 1 may be used in this situation.

The previous solution was rather uninteresting. Let us now consider a different approach to solving the same problem. This is a distributed semaphore. The version presented here is a simplification of one proposed by A. J. Martin and presented by Dijkstra[15,14]. After we talk about subsystems in the next section, we will see how to correct some of the shortcomings of this simplification. Each of our original processes M_i will communicate solely with a service process m_i. Visualize the processes m_i arranged in a ring, so that m_i communicates with processes m_{i+1} and m_{i-1} in addition to M_i. Refer to figure 4.5. Imagine that initially one of the m processes holds a token that it may either give to its left neighbor (facing the center of the ring) or to its associated M process. An M process will only enter the critical section when it has the token. Since there is only one token mutual exclusion is guaranteed. Of course, no process is allowed to "manufacture" a token.

M_i giving the token to its left neighbor is represented by communicating symbol t_i. Receiving from the right neighbor is represented by communicating t_{i-1}. V_i and P_i represent M_i giving the

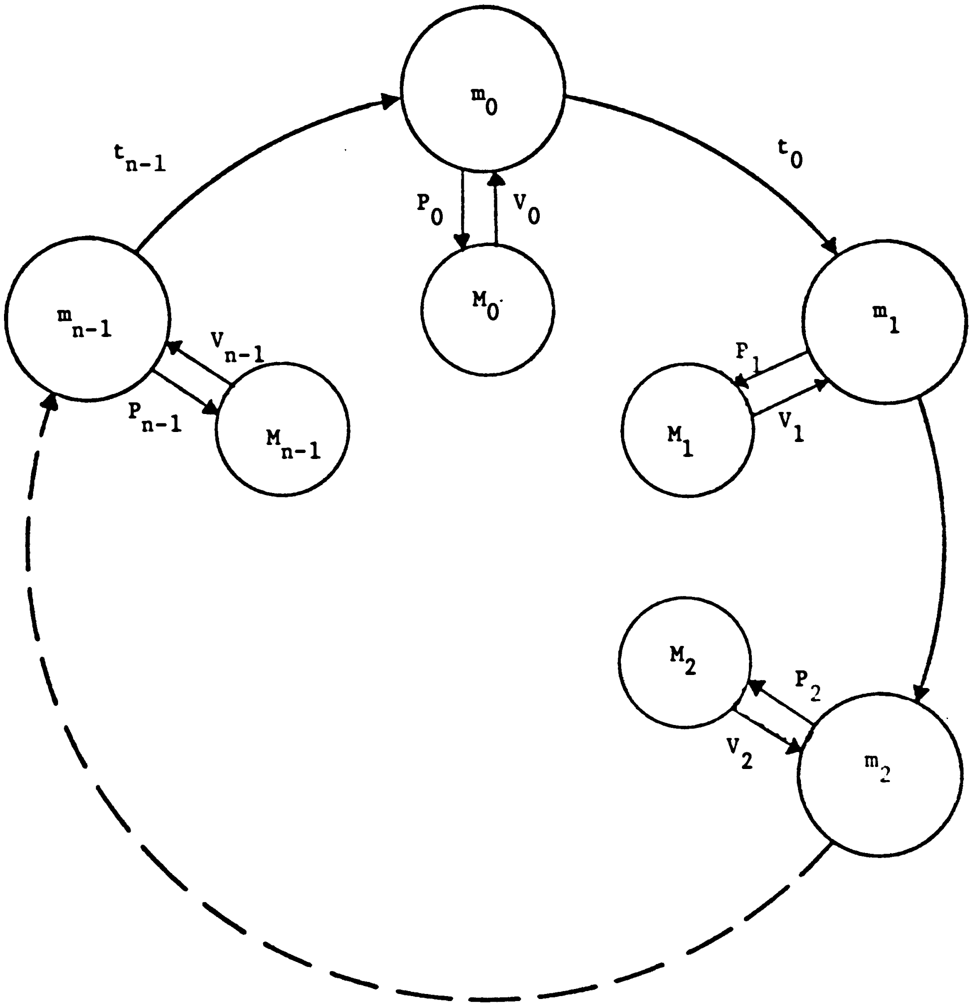

Figure 4.5
Distributed semaphore

token to m_i and receiving it back from that process.

Let us now derive specifications for the processes m_i so that they insure mutual exclusion between the M processes. Let T_i be 1 if process m_i initially has a token and 0 otherwise. Define functions F_i and G_i from histories into integers by

$$F_i(h) = T_i + |h{\downarrow}V_i| + |h{\downarrow}t_{i-1}| - |h{\downarrow}P_i| - |h{\downarrow}t_i|$$

$$G_i(h) = |h{\downarrow}P_i| - |h{\downarrow}V_i|$$

Notice that $G_i(h)$ is constrained to be either 0 or 1 by process M_i. Assume that T_i=1 for exactly one i. Then we see that

$$\forall h \in \Sigma^*. \sum_{i=0}^{n-1} (F_i(h) + G_i(h)) = 1.$$

and discover that

$$\forall i \in (0..n-1),\ h \in L(m_i).\ F_i(h) \geq 0$$

is sufficient to insure the mutual exclusion property

$$\forall h \in L(S). \sum_{i=0}^{n-1} G_i(h) \leq 1.$$

Why is it not also required that $F_i(h)$ be bounded above by 1? Process M_i already insures $G_i(h) \geq 0$ so the structure of the system already enforces the upper bound on each F_i; there is no need for it to be included in the specification of the individual processes. Recall at this point that <u>adding processes restricts potential communications of other processes with which they share communications symbols</u>. Thus, a safety property of a process automatically becomes a safety property of any system containing the process. In Chapter Five this is formalized as an inference rule for systems.

The liveness specifications for process m_i are given in Figure 4.6.

$\forall h \in L(m_i).\ \exists h' \in L(m_i).(h' > h \wedge |h' \downarrow P_i| > |h \downarrow P_i|)$ and

$\forall h \in L(m_i).\ hV_i \in L(m_i) \wedge ht_{i-1} \in L(m_i)$

$\forall h \in L(m_i).\ F_i(h)=1 \Rightarrow ht_i \in L(m_i).$

Figure 4.6 Specification of a Ring Process

The process must always be able to accept a token, it must be able to pass it around the ring if it has it, and it must eventually be able to give it to process M_i.

The distributed semaphore in the previous example introduces possibilities for anomalous behaviors not allowed by the monolithic semaphore. For instance, the ring processes are permitted to communicate among themselves for periods of unbounded length without letting any M process into its critical section. Furthermore, message traffic around the ring continues even when none of the M processes desires entry to its critical section. These problems are deferred until subsystems have been discussed.

4.2.3. Subsystems

One of the goals of this work was to be able to deal with processes and systems in almost the same way. So far, we have seen that the semantics of both can be expressed either with sets of traces or with sets of communications histories. However, the results of the previous section imply that the language of a system is formed over the union of the alphabets of the component processes. One would expect that the similarity of the interfaces between a process and its environment and a system and its environment should lead to the ability to replace a process with a system having the same meaning and vice versa. This is indeed the case, being a simple corollary of the system representation theorem.

Replacing a process with a subsystem of processes may introduce a need for communications that are known only to the processes of the subsystem. The rules as described thus far do not allow this. The other processes of the system must see all of the communications that any process does, and, by having a particular symbol in their alphabet, acquire the right to veto communication of that symbol.

Suppose that S is a subsystem and $\Sigma(S)$ is partitioned into $\Sigma_{private}(S)$ and $\Sigma_{visible}(S)$. If S is incorporated into a larger system, S', then S' should only be able to affect those communications of S from $\Sigma_{visible}$, while those communications that come from the private alphabet remain hidden. This section explores the technical issues involved in extending the simple parallel programming language by such a mechanism. In particular we identify a class of subsystems with hidden communications that behave much like deterministic processes. Although they are not deterministic the consistency theorem for systems containing them is a generalization of that for systems containing only processes.

The notation for a subsystem with hidden communication will be $(S{\downarrow}\Omega)$ standing for the system that behaves like S with only symbols from Ω available to the environment. $(S{\downarrow}\Omega)$ is defined by

$$\Sigma(S{\downarrow}\Omega) = \Omega$$

$$\tau(S{\downarrow}\Omega) = \tau(S)$$

$$L(S{\downarrow}\Omega) = (L(S){\downarrow}\Omega) = \{h{\downarrow}\Omega \mid h \in L(S)\}.$$

Subsystems with hidden communications cannot be introduced into the programming language with the expectation that the consistency theorem for systems will continue to hold. After all, that theorem dealt with systems composed of deterministic processes and $(S\downarrow\Omega)$ may not be deterministic. To obtain a consistency theorem for systems with subsystems a notion of determinism applicable to such systems is defined.

Definition 4.3: A subsystem $(S\downarrow\Omega)$ is apparently deterministic if

$$\forall t_1,t_2\in T(S).[(t_2\downarrow\Sigma(S)) > (t_1\downarrow\Sigma(S)) \Rightarrow$$
$$\exists t_3\in T(S).\ (t_3>t_1 \wedge (t_3\downarrow\Sigma(S)) = (t_2\downarrow\Sigma(S)))]$$

Corollary 4.4:

If a process P is deterministic then it is apparently deterministic.

Proof:

Let t_2 and t_3 be identified in the definition of apparent determinism and observe that $t_2>t_1$ by determinism of P.

QED

Although the definition of apparent determinism appeals to the set of traces of a system, we can show that if all the component processes and subsystems of a system are apparently deterministic then the system itself will be apparently deterministic. Furthermore, a simple condition on the language of the system insures that hiding some of the communications of the system preserves the apparent determinism. Using Lemma 4.5 in Figure 4.7, the proof of apparent determinacy of a system of apparently deterministic processes is stated and proved as Theorem 4.6 in Figure 4.8. The proofs are rather tedious. We thus see that apparent determinism is a useful restriction of non-determinism for reasoning about systems of parallel processes.

Lemma. 4.5:

If (1) $S = S_1 \,//\, S_2 \,//\, \ldots \,//\, S_n$, and

(2) each of the S_i is apparently deterministic

then $\forall t \in T(S),\ x \in \Sigma(S),\ (t \downarrow \Sigma(S))x \in L(S) \Rightarrow$

$\exists t' \in T(S).\ t' > t \wedge (t' \downarrow \Sigma(S)) = (t \downarrow \Sigma(S))x$

Proof:

Arbitrary $t \in T(S)$, $x \in \Sigma(S)$ where $(t \downarrow \Sigma(S))x \in L(S)$.

let $h = t \downarrow \Sigma(S)$

let $g_i = t \downarrow \Sigma_i \quad 1 \le i \le n$

let $s_i = t \downarrow \tau_i \quad 1 \le i \le n$

$g_i \in L_i$ (by the system representation theorem)

$s_i \in T_i$ (by the definition of $T(S)$)

for each i such that $x \in \Sigma_i$

$g_i x \in L_i$ (by the system rep. thm.)

$\exists r_i \in T_i$ with $r_i > s_i$ and $(r_i \downarrow \Sigma_i) = g_i x$

furthermore $\mathrm{last}(r_i) = x$

Let q_i be defined by

$r_i = s_i q_i x$

q_i contains no communication symbols

(if it did $r_i \downarrow \Sigma_i) \neq g_i x)$

for each i such that x is not a member of Σ_i

let $q_i = \epsilon$.

let $t' = t q_1 q_2 \ldots q_n x$.

$t' \downarrow \Sigma(S) = hx$

$t' \downarrow \tau_i = r_i \in T_i$

so $t' \in T(S)$.

QED

Figure 4.7 Lemma 4.5

Theorem 4.6:

If assumptions (1) and (2) of Lemma 4.5 hold
then S is apparently deterministic.

Proof:

Arbitrary t_1, $t_2 \in T(S)$ where $(t_2 \downarrow \Sigma) > (t_1 \downarrow \Sigma)$
the proof proceeds by induction on $|t_2 \downarrow \Sigma| - |t_1 \downarrow \Sigma| \equiv n$

Base case: n=1
Instance of Lemma 4.5 since there is an $x \in \Sigma(S)$
such that $|t_2 \downarrow \Sigma| = |t_1 \downarrow \Sigma| x$

Induction step: assume the theorem holds when n = i; show that
it holds when n = i+1.
$(t_2 \downarrow \Sigma) = (t_1 \downarrow \Sigma) yx$ where $y \in \Sigma^*$, $x \in \Sigma$
$|y| = i$
$(t_1 \downarrow \Sigma) y \in L(S)$
(by prefix closure of L(S))
$\exists t' \in T(S).\ (t' \downarrow \Sigma) = (t \downarrow \Sigma) y$
(by the definition of L(S))
$\exists t'' \in T(S).\ t'' > t \wedge (t'' \downarrow \Sigma) = (t \downarrow \Sigma) y$
(by the induction hypothesis)
$\exists t_3 \in T(S).\ t_3 > t'' > t_1 \wedge$
$(t_3 \downarrow \Sigma) = (t_2 \downarrow \Sigma)$
(by Lemma 4.5)
QED

Figure 4.8 Theorem 4.6

With Theorem 4.6 we can prove the central result of this section: If some of the communications of a subsystem are to be hidden, we need only prove that its communication language, and not its trace set, satisfy a certain property. This is Theorem 4.8 in Figure 4.10, in which we also use Lemma 4.7 of Figure 4.9

Lemma 4.7:

If (3) $\forall h \in L(S).\ x \in \Omega.\ (h \downarrow \Omega)x \in L(S \downarrow \Omega) \Rightarrow$
$\quad \exists h' \in L(S).\ h' > h \wedge (h' \downarrow \Omega) = (h \downarrow \Omega)x$

then $\forall h_1.\ h_2 \in L(S).(h_2 \downarrow \Omega) > (h_1 \downarrow \Omega) \Rightarrow$
$\quad \exists h_3 \in L(S).$
$\quad\quad h_3 > h_1 \wedge (h_3 \downarrow \Omega) = (h_2 \downarrow \Omega)$

Proof: Obvious. By induction on $|h_2 \downarrow \Omega| - |h_1 \downarrow \Omega|$

Figure 4.9 Lemma 4.7

Theorem 4.8:

If conditions (1) and (2) of Lemma 4.5 hold
and (3) of Lemma 4.7 holds
then $(S \downarrow \Omega)$ is apparently deterministic.

Proof:

arbitrary $t_1.\ t_2 \in T(S)$ where $(t_2 \downarrow \Omega) > (t_1 \downarrow \Omega)$
let $h_1 = (t_1 \downarrow \Sigma(S)),\ h_2 = (t_2 \downarrow \Sigma(S))$
$\exists h_3 \in L(S).\ h_3 > h_1 \wedge (h_3 \downarrow \Omega) = (h_3 \downarrow \Omega)$
(by Lemma 4.7)
$\exists t_3 \in T(S).\ (t_3 \downarrow \Sigma(S)) = h_3$
(by the definition of L(S))
$\exists t_4 \in T(S),\ t_4 > t_1 \wedge (t_4 \downarrow \Sigma(S)) = h_3$
(by Theorem 4.6)
hence$(t_4 \downarrow \Omega) = (h_2 \downarrow \Omega)$
so $(S \downarrow \Omega)$ is apparently deterministic
QED

Figure 4.10 Theorem 4.8

One concludes from this theorem that if the processes making up a system and its subsystems are all deterministic, as required by the syntax of SPPL, then no trace sets need to be considered to show that the antecedent of the consistency theorem holds. Therefore, one can be convinced of the correctness of one's reasoning without thinking about traces. It would be nice to have a corresponding completeness theorem stating that any system specification formula that is true can be proven without appealing to the trace set.

Theorem 4.9: (Consistency for apparently deterministic systems)
If S is any apparently deterministic system
then for any system specification formula F,

$$F(P) \Rightarrow Int[F(P)].$$

This proof is very similar to the consistency proof for deterministic processes. The safety consistency proof is identical, since safety properties are independent of determinism and non-determinism. The proof of consistency for liveness properties is different in that it appeals to the definition of apparent determinism rather than determinism and this requires the introduction of one extra step into the proof.

If F is a safety formula, $\forall h \in L(S).\ Q(h)$:

Assume $\forall h \in L(S).\ Q(h)$

arbitrary $t \in T(S)$.

$t \downarrow \Sigma(S) \in L(S)$ definition of L(S)

$Q(t \downarrow \Sigma(S))$

hence $\forall t \in T(S).\ Q(t \downarrow \Sigma(S))$

$F \Rightarrow Int(F)$

If F is a liveness formula,

$\forall h_1 \in L(S).\ Q_1(h) \Rightarrow \exists h_2 \in L(S).\ h_2 > h_1 \wedge Q_2(h_1, h_2)$:

Assume $\forall h_1 \in L(S).\ Q_1(h) \Rightarrow \exists h_2 \in L(S).\ h_2 > h_1 \wedge Q_2(h_1, h_2)$

arbitrary $t_1 \in T(S)$ where $Q_1(t_1 \downarrow \Sigma(S)$

$\exists h \in L(S).\ h > (t_1 \downarrow \Sigma(S)) \wedge Q_2(t_1 \downarrow \Sigma(S), h)$

choose $h \in L(S).\ h > (t_1 \downarrow \Sigma(S)) \wedge Q_2(t_1 \downarrow \Sigma(S), h)$

$\exists t_2 \in T(S).\ t_2 \downarrow \Sigma(S) = h$ definition of L(S)

choose $t_2 \in T(S).\ t_2 \downarrow \Sigma(S) = h$

$\exists t_3 \in T(S).\ t_3 > t_1 \wedge (t_3 \downarrow SIGMA(S)) = h$ apparent determinism

choose $t_3 \in T(S).\ t_3 > t_1 \wedge (t_3 \downarrow SIGMA(S)) = h$

$\forall t_1 \in L(S).\ Q_1(t_1 \downarrow \Sigma(S)) \Rightarrow$

$\exists t_2 \in T(S).\ t_2 > t_1 \wedge Q_2(t_1 \downarrow \Sigma(S), t_2 \downarrow \Sigma(S))$

$F \Rightarrow Int(F)$

QED

4.2.4. Example

Let us return to the distributed semaphore example and consider the subsystem $m' \equiv m_0 \,//\, m_1 \,//\, \cdots \,//\, m_{n-1}$. We would like to look at this subsystem as an implementation of a semaphore. We could then ignore the communications between the ring processes when using the subsystem in a larger system, simplifying any arguments about the containing system. Let $PV = P \cup V$. Can we show that $(m' \downarrow PV)$ is apparently deterministic using the specifications of the component processes? Yes.

We are asked to show that whenever $h \in L(m')$ and $(h \downarrow PV)P_i \in L(m' \downarrow PV)$ that there is an extension of h in L(m') whose next communication from PV is P_i. From the earlier discussion of the distributed semaphore we know that $G_i(h) = 0$ since a P can be communicated by some history that has P's and V's in the same order and number as h does. Thus there is a j such that $F_j = 1$, meaning some process has the token. The liveness specifications of processes m_j and m_{j+1} require that they be able to communicate a t_i, and the token is then at process m_{j+1}. This can be repeated until the token reaches process m_i. As written the specifications of m_i don't require it to then be able to communicate a P_i, but allow it to pass the token on to its neighbor, whence it must be able to again reach m_i. The specifications of m_i do require that if it receives the token enough times it will be able to communicate a P_i.

The argument in the case where V_i must be the next symbol from PV to be communicated is even simpler. In this case the liveness specification of process m_i insures that it can communicate V_i immediately.

The distributed semaphore is disturbing because of the possibility that it will delay all of the M processes from entering their critical sections for an arbitrary long time. The monolithic semaphore doesn't have this problem--it can delay a given process arbitrarily long, but some process can enter whenever the critical section is free. In [15,14] this difficulty is resolved by partitioning the P operations into R(equest) and G(rant) operations. This makes information about a request available to the ring processes before the request is granted so some short term scheduling of communication can occur.

The short term scheduling of communication insures that the token will remain in one ring process as long as no requests are pending. When a request is made by an M process, the m process that receives it will either deliver the token to the requesting process or request the token from its right neighbor. Eventually the process that has the token receives the request and starts the token back toward the requesting process. If a request from another M process is accepted by a ring process before the token reaches the requesting process, the token may not move all the way around to the originally requesting process. Nevertheless, this implementation can be made to share the property of the monolithic semaphore that once one M process wants to enter the critical section some M process will enter within a bounded number of communications by the system.

4.3. Conclusion

Partitioning a communication like P, which combines a request to do something with permission to do it, into a message that requests permission and one that grants it is a fairly general technique for solving short term scheduling problems in parallel systems. In the readers and writers example, the given specifications represent the minimum necessary to insure the desired mutual exclusion without deadlock. But priority between readers and writers cannot be specified because there is no way for the allocator to see a request before it is granted. By partitioning the "start read" communication into "request read permission" and "permission granted" communications and similarly for the "start write" message we could begin to talk about priority between reads and writes.

There is room for a good deal of exploration along the lines of the above analysis. For example, there is some interest in language features that support short term scheduling[3]. What are the advantages of such language features over the primitive and powerful techniques that allow programming of arbitrary short term scheduling policies?

The next chapter develops a verification theory for specifications based on communication languages, but the techniques presented in this chapter could well be put to broader use in analyzing short term scheduling features and network communication protocols.

Chapter Five: Verification of Processes and Systems

5. Introduction

This chapter presents techniques for proving that processes and systems meet their specifications. The proof rules are used to prove properties of the language of a process or system without actually computing its language. This is analogous to the way that Hoare's axiomatic rules allow us to prove properties of a sequential program's functionality without figuring out the actual function that it computes.

Verifying that processes and systems meet their specifications can be addressed in any number of ways. The obvious approach is to work directly with the inductive definition of the communications language in terms of the set of feasible traces. Working with the traces, however, is made difficult by their size and the fact that a presentation of a program doesn't communicate the trace set to the reader very well at all.

Another possible approach, suffering from much the same defect as the one above, is to develop an axiomatic logic like Hoare's axiom system for sequential programs or Pratt's dynamic logic and process logic. Dynamic logic and process logic, in their propositional forms, seem ideally suited to studying properties of programming logics rather than properties of programs. A "proof" of a program using dynamic logic is typically presented in outline form by placing assertions into the text of the program. This improves the readability of the "proof" but does not really constitute a proof--the justification for making the assertions has been omitted.

The rules for reasoning about processes are presented in the style of the PL/CV programming logic, extending the logic to parallel programs using the constructs of SPPL. PL/CV formalizes the widely used proof outline technique based on axiomatic semantics and described by Owicki and Gries in [59,58]. It is appealing because of the compactness of its proofs compared with formal proofs in the pure axiomatic system and because it treats a program and its proof as a single logical entity called an argument. Since arguments are the basic objects in PL/CV, programs and proofs must be considered simultaneously.

The rules for showing correctness of systems are simply axioms that state the definition of the language of a system in terms of the languages of the component processes.

5.1. Rules for processes

PL/CV style proof rules for safety and liveness specifications can be developed by observing the relationship between the communications language of a process and the possible communications histories of executions that end at each point in the program. The communications language is simply the union of the sets of communications histories possible at each point in the process. Therefore, an assertion $\forall h \in L(P).\ Q(h)$ may be proven by proving $Q(h)$ of all possible histories at each point in the program. How this is done depends upon whether one is dealing with a safety specification or with a liveness specification.

Safety specifications are proved with the techniques of sequential PL/CV. In addition to the rules of PL/CV for the *if*, *while*, *skip*, and assignment statements, there are new rules for the comm statement, process initialization and process termination. These statements are described by their effects on the value of an auxiliary variable h that records the communications history of any particular execution of the process. The inference rules for the comm statement describe the effect of the statement on the value of h. The fact that h is an auxiliary variable means that no storage need ever be allocated to it by an implementation. This is important since the value of h may become arbitrarily long.

To make a liveness assertion about a point in a program, more than just the values of the variables at that point must be known, since liveness depends upon the future (from that point) potential executions of the process. This is reflected in the liveness assertions that are written into a process by the explicit occurrence of the name of the process. A liveness assertion contains references to the process itself by referencing the language of the process. This phenomenon can be observed in this simple example: consider the two processes in Figure 5.1. Process P1 satisfies the liveness assertion at point L1, but process P2 does not satisfy the liveness assertion at L2, even though the states at the two points are identical.

```
P1: process                          P2: process
        x := 3;                              x := 3;
        y := 4;                              y := 4;
    L1: h∇∈L(P1)                         L2: h∇∈L(P2)
    endprocess                               while true do skip od
                                         endprocess
```

Figure 5.1 Liveness Depends on More than State Alone

Of course, the problem is that the concept of state that is being used is not adequate. If we are going to talk about states satisfying formulae, then the state must in some sense contain a copy of the program, otherwise, the phenomenon of the above example occurs. Introducing such a component explicitly into the state is not too appealing because an entire program is a rather bulky thing. However, in a programming logic like PL/CV, where asserted program text is the fundamental structure in the logic, this problem can be overcome by *implicitly* incorporating the program into the state. In PL/CV, assertions are only made directly in the text of the program to which they refer. An assertion has, simply by its position in the text, some contextual meaning that cannot be expressed as an assertion about the values of the variables at that point alone.

In the existing PL/CV, this contextual meaning component is used in the proof rules to require that some assertions (notably having to do with loops and conditional statements) have certain physical positions in the text before they can be used in justifying other assertions. Essentially, however, the rules take advantage of only half of the context available at a given position--the statements and assertions that have occurred previously in the argument. In reasoning about the input-state output-state behavior of sequential programs the preceding or left context is sufficient since the state at any point depends completely on the input state and the statements that are executed in reaching that point.

In reasoning about the communications potential of a process, though, the left context is not adequate. We are interested in what can happen later in an execution passing through a point as well as what has happened previously to passing through the point. The previous comments about the state depending only on the input state and the previously executed statements must be tempered somewhat, or no new power could be gained by looking at the following, or right, context. Of course the state can still only depend on the input state and previously executed statements. However, there are now assertions that do not depend only on the state, but also on the entire program text. The prime example of such an assertion is $\underline{h} \in L(P)$ which is valid anywhere inside a process labelled P, regardless of the input state or what statements have been previously executed. L(P) is defined to be the set of all strings P can communicate on any partial execution, and if a partial execution passes through a particular point, $\underline{h}$ records the communications history of that execution.

Let us now try to understand how the context following a position can influence the truth of assertions at that position. Suppose that S is a statement and P is an assertion immediately following S that has been proven. What is known about the states possible at the point immediately preceding S? In any state, s, possible immediately before S it must be the case that execution of S begun in that state results in a state satisfying P. That is $s \in wp(S,P)$ provided execution of S terminates. Thus if Q is any formula such that $wp(S,P) \Rightarrow Q$ then all the states satisfying $wp(S,P)$ (all the states possible at that point) satisfy Q and Q may be asserted at that point.

The rules that follow implement this observation for the statements of SPPL. The introduction of rules that allow justification on the basis of text in the right context of a position must be done carefully to avoid circular arguments. In PL/CV circular arguments are impossible because every assertion must be justified on the basis of assertions that lexically precede it. This linearly orders the assertions, guaranteeing that there can be no circular dependencies. Rules that reference right context destroy the property that every assertion is justified on the basis of lexically previous assertions, so further steps must be taken to prevent circular reasoning. The approach taken here is to write assertions with any ancestor assertion derived from left context (<u>left assertions</u>) to the left of a colon while assertions derived solely from right context (<u>right assertions</u>) are written to the right of a colon. Obviously a right assertion cannot depend upon itself because it depends only on its right context. A left assertion cannot depend upon itself because it is derived only from its left context and right assertions in its right context.

In this system as in PL/CV an assertion can be justified at a point in an argument if it follows from assertions that are accessible at that point and the inference rules for programs and the underlying data domains. The rules of accessibility are therefore extremely important.

In PL/CV, where all justifications are made on the basis of left context, an assertion A (starting) at position p with free variables

$x_1, \ldots, x_n$ is accessible from position q in an argument if and only if

1. p lexically precedes q, and

2. Any proof block containing p also contains q, and

3. Any proof block containing q that qualifies any of $x_1,\ldots,x_n$ also contains p.

4. None of $x_1,\ldots,x_n$ is changed on any execution path from p to q.

If argument on the basis of right context is to be possible, the accessibility rules must be changed. In particular, the restriction that an assertion is accessible only from lexically later points must be relaxed. Nevertheless, a proof must have the property that the assertions made in it are not self-dependent. In justification by left context, lexical ordering insured this property. An ordering on the assertions in an argument that prevents potential dependencies is still required when we allow argument on the basis of both left and right context. It just cannot be lexical ordering.

A convenient ordering is one that separates assertions justified by right context alone and those justified by both left and right context as described above. The accessibility rule for this proof organization distinguishes positions as being either right positions or left positions depending upon whether they are to the right or left of a colon, respectively.

An assertion A (starting) at position p with free variables $x_1,\ldots,x_n$ is accessible from right position q in an argument iff p is a right position and lexically follows q, conditions 2 and 3 above hold, and 5: none of $x_1, \ldots, x_n$ is changed on any execution path from q to p.

An assertion A (starting) at position p with free variables $x_1,\ldots,x_n$ is accessible from left position q in an argument iff p is a right position and lexically follows q and condition 2, 3, and 5 hold, or p lexically precedes q, and conditions 2 through 4 above hold.

A proof in this system is, just as in PL/CV, an argument in which every occurrence of an assertion, A, is either an assumption or follows from assertions accessible from the position of A by a rule of inference.

The first rule for processes is one that allows conclusion of a property of its communications language from properties of the history variable at each point in the program. To do this use the process elimination rule:

```
process PR:
    stmt                Q: accessible from everywhere in process PR
endprocess                 (including after the endprocess)
```

$$\frac{}{\forall h \in L(PR).\ Q(h/\underline{h})}$$

5.1.1. Proving Safety Properties of Processes

PL/CV rules for the versions of the usual sequential programming constructs used in SPPL are summarized in Appendix 1. PL/CV itself requires proofs that all loops terminate, but previous discussion has indicated that this is undesirable in some processes and systems. Termination is replaced by liveness, which is dealt with in the next section.

Before a process has done any communication, $\underline{h}$ is empty. This is captured in the process initialization rule:

process $\underline{h} = \epsilon$: . . . endprocess

Accessibility rules insure that the assertion $\underline{h} = \epsilon$ is available anywhere syntactically before the first comm statement.

The value of h can only be changed by the comm statement and process termination and then only in a particular way: h is lengthened by exactly one communication symbol each time a comm statement is executed and when the process terminates. This is very similar to the assignment statement, and the rule reflects this in its form. It also takes into account the boolean expressions in the guards and the fact that there are several branches of the comm statement. This rule is called guard elimination, and is expressed in the following schema:

$\bigwedge_{i=1}^{n} b_i \Rightarrow P_i(hc_i/h)$:
comm
  $b_1 : c_1 \rightarrow$
    $\uparrow$
  $P_1 \wedge b_1$:   conclusion
  .
  .
□ $b_n : c_n \rightarrow$
    $\uparrow$
  $P_n \wedge b_n$:   conclusion
endcomm

The termination of a process appends a ∇ to its local history. This is expressed in the termination rule:

$$\frac{\text{process} \; \ldots \; P(h\nabla/h)\text{: endprocess}}{P\text{:}}$$

Finally, there is a rule analogous to the conditional elimination rule, that allows a conclusion about the state following a comm statement. Let us call it comm elimination. If Q is true after executing any branch of the comm statement, and is also true if none of the branches can be executed, then it is true after the comm statement.

```
  n
( ∧ ¬b_i) ⇒ Q:
 i=1
comm
    b_1:c_1 → stmt_1 Q:
        .
        .
 □  b_1:c_1 → stmt_1 Q:
endcomm
_____________________________
            Q:
```

The binary semaphore specification and an implementation of the binary semaphore serve as an example of how these rules can be used to prove safety properties of processes.

```
process B
c := 1;
(c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
while true do
    (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
    comm
        c=1:P_1 → c := 0
            (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
     □  c=0:V_1 → c := 1
            (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
            .
            .
     □  c=1:P_n → c := 0
            (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
     □  c=0:V_n → c := 1
            (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
    endcomm
    (c = 0 ∨ c = 1) ∧ c = 1 - (|h↓P| - |h↓V|):
od
endprocess
```

We can therefore conclude $\forall h \in L(B).\ 0 \leq (|h\downarrow P| - |h\downarrow V|) \leq 1$ as a simple arithmetic consequence of the invariant and the process elimination rule.

5.1.2. Proving Liveness Properties of Processes

To prove liveness properties we argue on the basis of the right context of each position, taking advantage of the properties discussed in Section 5.1. The rules are unusual enough that a soundness argument about them is presented in Section 5.1.2.2.

5.1.2.1. Liveness Rules

Most of the liveness rules are nearly identical to their safety counterparts in presentation. The only differences are which assertions in the rule are the hypotheses and which are the conclusions. In these rules, the hypotheses appear _below_ a dashed line, the conclusion above, representing the fact that the conclusion appears to the left of (above) the hypotheses in the argument. The first right context rule is the axiom expressing the fact that _h_ is always in the language of the process. We might call it the language introduction rule since it is the only rule that introduces a reference to the language of a process (although other rules will propagate the reference). The language-intro rule is:

process P: . . . :$h \in L(P)$ . . . **endprocess**

The process termination rule is:

process P: . . . :$h\nabla \in L(P)$ **endprocess**

The assignment rule is:

```
   :P(e/x)
----------
    x:=e
     :P
```

The skip rule is:

:P <u>skip</u> :P

The conditional rules are:

$$\frac{:\ b \Rightarrow P \wedge \neg b \Rightarrow Q}{\underline{\text{if}}\ b\ \underline{\text{then}}\ :P\ \ldots\ \underline{\text{else}}\ :Q\ \ldots\ \underline{\text{fi}}}$$

and

$$\frac{\underline{\text{if}}\ b\ \underline{\text{then}}\ \ldots\ :P\ \underline{\text{else}}\ \ldots\ :P\ \underline{\text{fi}}}{:P}$$

The communication rules are:

$$:\ \bigwedge_{i=1}^{n} b_i \Rightarrow P_i(\underline{h}c_i/\underline{h})$$

<u>comm</u>
    $b_1 : c_1 \rightarrow$
        $:P_1$ . . .
        .
        .
 □ $b_n : c_n \rightarrow$
        $:P_n$ . . .
<u>endcomm</u>

and

$$:(\bigwedge_{i=1}^{n} \neg b_i) \Rightarrow Q$$

<u>comm</u>
    $b_1 : c_1 \rightarrow stmt_1$:Q
        .
        .
 □ $b_n : c_1 \rightarrow stmt_n$:Q
<u>endcomm</u>

:Q

Finally, we come to the difficult rule: the rule for inferring something at a point with a loop in the nearby right context.

```
:b ∧ ∀i≥0. Q(i/n) ∨  b ⇒ R
------------------------------------------------------------
Q(0/n)                    (Q(0/n) independent of variable values)
while b do
    :Q(i+1/n)
        stmt
    :arbitrary i≥0 where b ∧ Q(i/n) ∨ ¬b ∧ R
    od
:R
```

In addition to these rules for reasoning about programs, we need rules for doing purely logical reasoning about assertions in right positions. The PL/CV immediate rules are adequate when used with the modified accessibility rules, but the PL/CV rules that require explicit proofs are not. The proofs have to proceed from lexically later positions to lexically earlier ones and the existing proof syntax does not allow this. The first of the required proof rules is implication in a right argument:

```
:A ⇒ B by intro.
    :qed
        :B
        ...
        [:assume A]
    :proof
```

The square brackets mean it is allowable to write the "assume A" but not necessary. The proof means that B follows from A by accessing assertions in only the right context of the argument. The tokens :qed and :proof indicate that the proof is proceeding from the bottom to the top, and this is why they are written in the rather peculiar order.

The right or elimination rule also requires an explicit proof and again it looks peculiar. Just remember that inferences are proceeding from the bottom of the rule to the top.

```
:Q by cases. A1 v ... v An.
    :qed
            :Q
            ...
        :case An
            .
            .
            .
            :Q
            ...
        :case A1
    :proof
------------------------------------------------
:A1 v ...An
```

Right context not introduction is again similar.

```
:¬A by intro.
    :qed
        :false
        ...
        [:assume A]
    :proof
```

Finally, the universal quantifier can be introduced similarly:

```
:∀x. P by intro.
    :qed
        :P
        ...
        :arbitrary x
    :proof
```

The example termination proof in Section 5.1.2.3 illustrates the use of these rules together with the right context program rules.

5.1.2.2. Soundness of the Liveness Rules

This section argues the soundness of the inference rules in the previous section. That is, they do not allow erroneous conclusions about processes. The central notion in these arguments is that of the _states possible_ at a point in an argument. By making an assertion at a point in an argument we are saying that the assertion holds for all states possible at that point.

The assignment rule serves as an example of the reasoning necessary to show soundness of the rules for the non-iterative statements in the language. Recall that the rule is

```
:P(x/e)
--------
 x := e
   :P
```

In words, this says "given that P is true of all states possible after the assignment x:=e, infer that P(x/e) is true of all states possible before x:=e."

The proof of soundness is by contradiction. (P(s) is the assertion of P about the particular state s.) Suppose the rule is unsound. Then there is an assignment statement x:=e and an assertion P such that P is true of all states possible after P, and there is a state s_0 possible before the assignment where P(x/e)(s_0) is false. But P(x/e) is the set of all states such that execution of x:=e begun in one of them results in a state satisfying P. Executing x:=e beginning in state s_0 must therefore result in a state s_1 where P is false. But we hypothesized that s_0 was possible before x:=e, so s_1 must be possible after, contradicting the requirement that P hold for all states possible after x:=e.

The soundness of the conditional and communication rules is even easier to see. Suppose that P and Q are true of all states possible at the beginning of the *then* and *else* branches respectively, and there is a state s_0 possible before the conditional such that $[b \wedge P \vee \neg b \wedge Q](s_0)$ is not true. Either $b(s_0)$ is true or $\neg b(s_0)$ is true. Suppose $b(s_0)$ is true. Then $P(s_0)$ is not true, but then s_0 is possible at the beginning of the *then* branch, contradicting the assumption that all states possible there satisfied P. A similar argument works for the other conditional rule and the two communication rules.

The interesting soundness argument is that for the loop rule. Recall the loop rule is

```
:b ∧ ∀i≥0. Q(i/n) ∨  b ⇒ R
------------------------------------------------------------
Q(0/n)                   (Q(0/n) independent of variable values)
while b do
    :Q(i+1/n)
        stmt
    :arbitrary i≥0 where b ∧ Q(i/n) ∨ ¬b ∧ R
    od
:R
```

Let

$PRE \equiv b \wedge \forall i \geq 0.\ Q(i) \vee \neg b \wedge R.$

Suppose the conditions required by the loop rule are satisfied and there is a state s_0 possible before the loop where $\neg PRE(s_0)$. There are two cases: $b(s_0)$ and $\neg b(s_0)$. Suppose $\neg b(s_0)$. Then $\neg R(s_0)$. But s_0 is possible at the end of the loop where R is true of all possible states, giving us a contradiction. Suppose $b(s_0)$. Then there is an i_0 such that $Q(i_0)(s_0)$ is false. Consider any execution of the loop begun with state s_0. The body of the loop is executed either more than i_0 times or no more than i_0 times. If it is executed more than i_0 times consider the sequence of states arising at the end of each of the executions of the body. The sequence is $<s_1, s_2, \cdots, s_{i_0}, s_{i_0+1}, \ldots>$. We know $b(s_{i_0}) \wedge Q(0)(s_{i_0})$. The argument in the body of the loop tells us $Q(1)(s_{i_0-1})$. Repeatedly applying the argument we discover $Q(i_0)(s_0)$, a contradiction.

If the loop body is executed no more than i_0 times the sequence of states at the bottom of the body of the loop is $<s_1, s_2, \cdots, s_k>$ where $k \leq i_0$. Thus $[b \Rightarrow Q(i_0-k) \wedge \neg b \Rightarrow R](s_k)$, so again by the argument in the body of the loop $Q(i_0-k+1)(s_{k-1})$. Repeatedly applying the above argument we find $Q(i_0-k+k)(s_{k-k})$. That is, $Q(i_0)(s_0)$, a contradiction.

Thus the rule for loops is sound provided the rules for the loop body are sound. Now we can do an induction on the level of nesting of loops to show that the rules for the body are sound. Let the <u>nesting depth</u> of a loop be zero if the loop contains no loops in its body, and the maximum nesting depth of the contained loops plus one otherwise. The base case of the induction is that the loop rule is sound for loops with nesting depth zero because they contain no other loops and the inference rules for non-loop statements are sound. The induction step says that if the loop rule is sound for all loops of nesting depth i, $0 \leq i \leq n$ then it is sound for all loops of nesting depth n+1. A loop with nesting depth n+1 contains loops of nesting depth at most n in its body, hence the inferences in the body are sound, and by the argument above the loop rule is sound for loops of nesting depth n+1.

5.1.2.3. Completeness Issues

The other side of the soundness coin is completeness. Of course we could not expect our logic to be absolutely complete in the sense that if P(S) is true of system S then P(S) is provable. The halting of a sequential program that does no communication is expressible in our logic by

$$\forall h \in L(S).\ |h \downarrow \nabla| = 0$$

Such programs can encode arbitrary Turing machine computations and the recursive enumerability of proofs in the system would then imply the recursive enumerability of the indices of Turing machines that do not halt on blank tape.

Therefore, if any completeness at all is to be found it will be relativized. We would at least like to show that reasoning from both left and right contexts is as powerful as the loop termination rule in PL/CV, since that rule has been replaced by the right context rules. Given an arbitrary program and a proof of its correctness in PL/CV, including termination, can we find a proof in this logic? There is some indication that the answer is yes, but the result is not established.

Consider the simplest possible terminating loop and its proof of termination in PL/CV.

```
assume k≥0
j := k;
∃i≥0. j=i;
while j>0 do
    j ≠ 0;
    arbitrary i where j=i;
    j := j-1;
    j=i-1;
    od
```

In the new logic the proof is quite a bit more complicated but it still can be done. Furthermore the predicate Q(i) required by the loop liveness rule in the new logic is derived from the termination predicate j=i of the PL/CV rule in a straightforward way.

```
assume k≥0:;
h = ε:;
∇◀L: by cases k>0 | k=0:.   (*the program terminates*)
    proof;
        case k>0:;
            ∀i≥0.(((i≥1 ∧ k=i) ⇒ h∇◀L)):;
            (k≥1 ∧ k=k) ⇒ ∇◀L):;
            ∇◀L:;
        case k=0;
            ∇◀L:;
    qed;
:k>0 ∧ ∀i≥0.((i≥1 ∧ k=i) ⇒ h∇◀L) ∨ k=0 ∧ h∇◀L;
j := k;
:j>0 ∧ ∀i≥0.((i≥1 ∧ j=i) ⇒ h∇◀L) ∨ j=0 ∧ h∇◀L;
(0≥1 ∧ j=i) ⇒ h∇◀L;
```

```
while j≠0 do
    :(i+1≥1 ∧ j=i+1) ⇒ h∇∈L by intro.
    :qed;
        :h∇∈L by cases
            :(j≠1 ∧ ((i≥1) ∧ j=i+1) ⇒ h∇∈L) ∨ (j=1 ∧ h∇∈L).
        :qed;
                :h∇∈L;
            case :j=1 ∧ h∇∈L;
                :h∇∈L;
                :i≥1;
                :0≠i;
                :i+1≥1 ∧ j=i+1;
                :(i≥1 ∧ j=i+1) ⇒ h∇∈L);
                :j≠1;
            case :(j≠1 ∧ ((i≥1) ∧ j=i+1) ⇒ h∇∈L)
        :proof;
    :proof;
    :(j≠1 ∧ ((i≥1) ∧ j=i+1) ⇒ h∇∈L) ∨ (j=1 ∧ h∇∈L);
    j := j-1;
    :arbitrary i where (j≠0 ∧ ((i≥1 ∧ j=i) ⇒ h∇∈L ∨ (j=0 ∧ h∇∈L);
    od
:h∇∈L
```

5.2. Rules for Systems

Given a program comprising processes P_1 // P_2 // ... // P_n we need a rule that allows us to prove from the specifications of its component processes that it meets its specifications. The proof rule that we use is just a restatement of the definition of the language of a system in terms of the language of the component processes. The specification of the system would then have to be proven using string and set theory.

$$\frac{P_1 \;//\; P_2 \;//\; \dots \;//\; P_n}{s \in L \Leftrightarrow (s \downarrow \Sigma_1 \in L_1) \wedge (s \downarrow \Sigma_2 \in L_2) \wedge \dots (s \downarrow \Sigma_n \in L_n)}$$

The important point here is that the specifications of the processes are proved outside the processes and hence are available at the end of the program for use in proving the specifications of the program. The program elimination rule provides the necessary additional information.

The subsystems discussed in Chapter Four are not strictly part of SPPL. If we did include them the language, the proof rule for $((P_1 \mathbin{//} P_2 \ldots P_n)\downarrow\Omega)$ would require that it be proven apparently deterministic as well as that it satisfied its safety and liveness specifications.

5.3. Conclusion

The logic presented here is useful for reasoning about processes and systems, but relative completeness has not been proved for the system. It seems that PL/CV termination proofs are easily mapped into this logic. Another remaining problem is the need for an elegant string theory, the foundation of our logic of processes. PL/CV automates much arithmetic reasoning in a very powerful proof rule. Can string theory be formulated in a way that is as elegant? Finally, how well does this style of reasoning capture the concepts of Pratt's process logic and Pnueli's temporal logic? Can proofs in these logics be formulated using rules similar to those proposed for liveness in SPPL? Observe that the reasoning we do for specifications $\forall h \in L(P).\ Q(h)$ is closely related to that which must be done to prove the statement "throughout P, R" in process logic. How well can the other modalities of process logic, such as "insures" and "preserves" be modeled in this one? What is their utility in specifying processes, perhaps with shared variables?

These techniques for proving liveness should be explored further on both the practical and theoretical levels. They also suggest that fruitful study of programming methods for parallel systems might be done in this area. The weakest precondition calculus relating specifications and program text for sequential programs might be expanded to include parallel programs.

Chapter Six: Conclusion and Analysis

This thesis has presented techniques for thinking and reasoning about parallel processes as meaningful entities, independent of the system in which they run. The processes are written in a simple parallel programming language that allows them to communicate with one another using message names. The standard semantics of the language is given using feasible traces.

We argue that traces are inappropriate for reasoning about and describing processes at a higher level and concentrate instead on the communication language of a process--the set of strings of messages that can be communicated by executions of the process. We distinguish a class of processes in SPPL, called the apparently deterministic processes, whose behavior is adequately characterized by their communication languages.

Finally, a PL/CV-like proof system is developed for process reasoning. The proof system structures the inductive arguments one could make in string theory using the definition of the communication language. The major advantage of this is that it presents a process and its proof simultaneously, making it easier for a reader to integrate the one and the other. The key to proving liveness specifications using asserted programs is the ability to justify an assertion on the basis of assertions in both it left and right contexts in the program. Contrast this with PL/CV and other asserted program techniques where inference can be made only from the assertions in the left context.

6.1. Extension of SPPL's Communication Mechanism

The programming language used here had messages consisting of only message names. In Hoare's Communicating Sequential Processes proposal messages carry both a name and data. The name provides synchronization and the data transfers information between processes. In SPPL data can be moved from one process to another by pre-defining message names to mean that a particular piece of information has moved from one process to another, or by encoding a piece of information as a sequence of synchronizations. Such encodings are unappealing because they obscure the intent of the programmer. Since we are really interested in specifying processes to improve their quality it seems that we defeat ourselves when we resort to such tricks.

The tricks, however, suggest that we extend our techniques to deal directly with data transfer between processes. Without formalizing data communication let us look at an example of a parallel system where data communication is useful.

Given monotonic, increasing functions f, g, and h from positive integers to positive integers, and given that $\exists i,j,k.\ f(i)=g(j)=h(k)$, write a program to print the least i, j, k such that $f(i)=g(j)=h(k)$.

Let us name the points we are looking for $\bar{i}$, $\bar{j}$, $\bar{k}$, the values of which are unknown. Then a loop that has invariant $i\le\bar{i} \wedge j\le\bar{j} \wedge k\le\bar{k}$ and terminates with $f(i)=g(j)=h(k)$ finds the correct answer and is a sequential program that solves the problem. See Figure 6.1.

```
i, j, k := 1, 1, 1;
do  f(i) < g(j)  →  i := i+1
 □  g(j) < h(k)  →  j := j+1
 □  h(k) < f(i)  →  k := k+1
od
print(i, j, k)
```

Figure 6.1 Sequential solution to f, g, h problem

Now suppose we wish to give a distributed solution to the same problem in order to do function evaluations in parallel. One process is used to compute f, another for g, and third for h. A fourth process will print the result. Let us call the four processes F, G, H, and print, respectively. Refer to Figure 6.2.

The specification for the problem remains that print should write the least i, j, k such that f(i)=g(j)=h(k). This implies that the other processes must compute the proper values without deadlocking, without running forever in a local loop, and without communicating forever. Let us focus on process F and its relationship to the other processes. The specifications for G and H are symmetric. Termination implies that the length of the communications histories of the system are finite. In fact we will bound their length with the system invariant and the process specifications of F, G, H. The system invariant is: all values communicated between the processes are less than or equal to $f(\bar{i})=g(\bar{j})=h(\bar{k})$. Process F is required to communicate an increasing sequence of values with G and an increasing sequence with H. The system invariant bounds each of these sequences above, implying they are bounded in length. Therefore, eventually the processes will stop communicating.

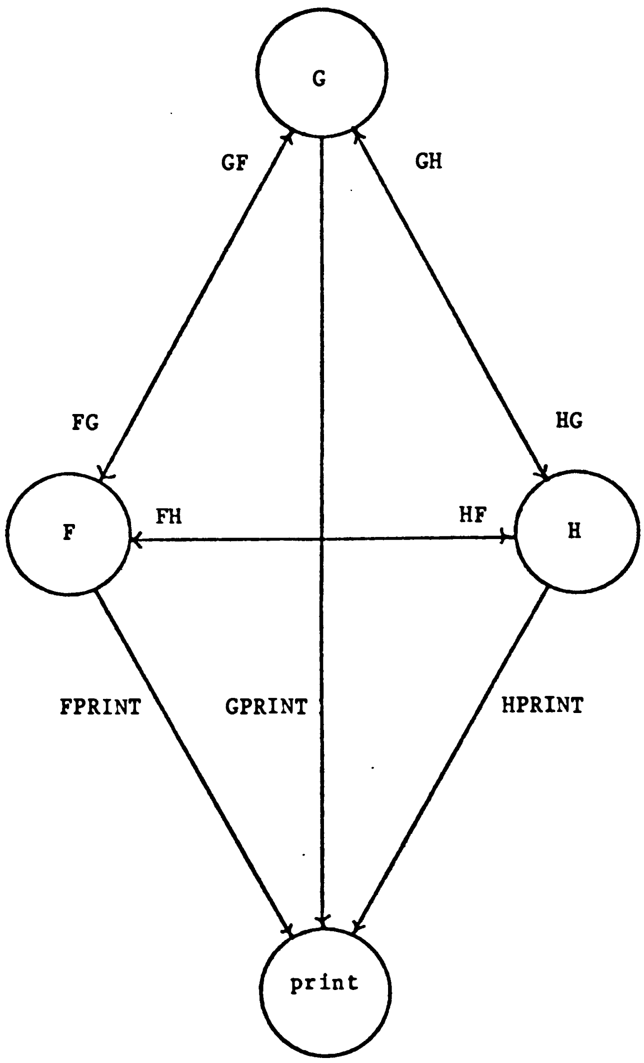

Figure 6.2 f, g, h problem

To avoid deadlock, it is necessary that F be able to send a value whenever it is required by G or H. That is, subject to monotonicity of the sequences, at any point in its execution, F will be able to send a value of f that is no less than the last value it received from G Preserving the system invariant requires that this value be the least such value of f. Furthermore, as long as F has not determined that the values it has received and computed are all equal it must be prepared to receive new values from G and H. When F has computed a value equal to the last two values it has received it must send the index it has computed to the print process. Then there can be no deadlock, because that would require all the processes to be waiting to receive, having sent the last value they computed. Then the last value sent by F to H is no less than the last value sent to the last value sent by G to F is no less than the last value sent by H to G, implying they are all equal. But then the specifications require the processes to send to the print process.

These informal specifications are implemented by the program in Figure 6.3. The notation for value communication follows Hoare[26] with <id>!<expression> representing synchronization on identifier id in the sense defined in Chapter 3, and communication of the value of the expression. <id>?<var> places the value of the expression in the matching <id>!<expression> guard into var.

```
F:: process
    i, sentG, sentH, fi, gj, hk := 1, false, false, f(i), 0, 0;
    whilefi≠gj ∨ fi≠hk ∨ ¬sentG ∨ ¬sentH do
        comm  ¬sentG:FG!fi → sentG := true
         □    ¬sentH:FH!fi → sentH := true
         □    GF?gj → skip
         □    HF?hk → skip
        endcomm;
        while fi<gj do
            i:=i+1; fi, sentG, sentH := f(i), false, false
        od
    od;
    comm FPRINT!i endcomm → skip
    endprocess
```

Figure 6.3 One process of the system

Process G is obtained by systematic substitution of G for F, g for f, H for G, h for g, F for H, and f for h in Figure 6.3. A similar substitution produces process H. Process print is shown in Figure 6.4.

```
print:: process
    comm FPRINT?i → skip endcomm;
    comm GPRINT?j → skip endcomm;
    comm HPRINT?k → skip endcomm;
    print( i, j, k )
    endprocess
```

Figure 6.4 The print process

The interesting aspect of this little exercise is that the specifications were suggested by standard sequential programming techniques (termination by showing existence of a bounded decreasing function of the state variables), not by the usual sequential solution to the problem. Together, the process specifications imply the system specification, which guarantees termination with the correct answers.

6.2. Directions for Further Research

In addition to the theoretical questions of Chapter Five, the details of this approach to communication of arbitrary data await further development. The answers to the theoretical questions will lead to a better understanding of parallelism and liveness, while the latter are important to practical application of the techniques. A practically adequate specification technique for systems of processes that share variables is another research direction to be explored. Finally, exploration of these techniques in the presence of bounded buffering of messages would add another dimension to the power of the method.

Bibliography

[1] Andler, S. "Predicate Path Expressions," Proceedings of the Sixth Annual ACM Symposium on the Principles of Programming Languages, 1979.

[2] Andrews, G. "Message Classes: an Approach to Process Synchronization," Computer Science Technical Report TR76-275, Cornell University, 1976.

[3] Andrews, G. R. "Synchronizing Resources," Computer Science Technical Report TR78-360, Cornell University, 1978.

[4] Apt, K. et al. "A Proof System for Communicating Sequential Processes," University of Rotterdam, 1979.

[5] Atkinson, R. and Hewitt, C. "Synchronization in Actor Systems," Proceedings of the Third Annual ACM Symposium on the Principles of Programming Languages, 1976.

[6] Bates, J. "A Logic for Correct Program Development," Ph.D. Thesis, Computer Science Technical Report TR79-388, Cornell University, 1979.

[7] Birtwistle, G. M. et al. Simula Begin, New York: Petrocelli/Charter, 1973.

[8] Chandy, K. M. and Misra, J. "An Axiomatic Proof Technique for Networks of Communicating Processes," Technical Report, University of Texas at Austin, 1979.

[9] Clarke, E. M. Jr. "Program Invariants as Fixed Points," Computer Science Technical Report CS-1977-5, Duke University, 1977.

[10] Constable, R. L. "On the Theory of Programming Logics," Proceedings of the Ninth Annual ACM Symposium on the Theory of Computing, 1977.

[11] ___________ and O'Donnell, M. J. A Programming Logic, Cambridge, Mass.: Winthrop Publishers, Inc., 1978.

[12] deBakker, J. W. "The Fixed Point Approach in Semantics: Theory and Applications," Mathematical Centre Tracts 63, Mathematical Centre, Amsterdam, 1975.

[13] Dijkstra, E. W. A Discipline of Programming. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1976.

[14] ____________. "More on A. J. Martin's Design." Unpublished report EWD 672, 1978.

[15] ____________. "On the Correctness of a Design by A. J. Martin." Unpublished report EWD 668, 1978.

[16] Donahue, J. E. "Complementary Definitions of Programming Language Semantics." Lecture Notes in Computer Science, Volume 42, 1976.

[17] Flon, L. and Suzuki N. "Consistent and Complete Proof Rules for the Total Correctness of Parallel Programs." Xerox PARC Report CSL-78-6, 1978.

[18] Francez, N. et al. "Semantics of Nondeterminism, Concurrency and Communication." Unpublished memorandum, 1978.

[19] Good, D. I. and Cohen, R. M. "Principles of Proving Concurrent Programs in Gypsy." Proceedings of the Sixth Annual ACM Symposium on the Principles of Programming Languages, 1979.

[20] Grief, I. " A Language for Formal Problem Specification." CACM, 20, 12, December, 1977.

[21] Gordon, M. et al. "Edinburgh LCF: A Mechanised Logic of Computation." Lecture Notes in Computer Science, to appear.

[22] Habermann, A. N. "Path Expressions." Carnegie-Mellon University, 1975.

[23] Hansen, Per Brinch. "Distributed Processes: A Concurrent Programming Concept." CACM, 21, 11, November, 1978.

[24] ____________. "The Programming Language Concurrent Pascal." IEEE Transactions on Software Engineering, SE-1, 2, June, 1975.

[25] ____________ and Staunstrup, J. "Specification and Implementation of Mutual Exclusion." Computer Science Department, University of Southern California, 1977.

[26] Harel, D. "Logics of Programs: Axiomatics and Descriptive Power." Ph.D. thesis, M.I.T. Laboratory for Computer Science, 1978.

[27] Hoare, C. A. R. "Communicating Sequential Processes." CACM, 21, 8, August, 1978.

[28] ____________. "A Model for Communicating Sequential Processes." Oxford University Computing Laboratory, PRG, December 1978.

[29] ___________. "A Model for Communicating Sequential Processes," Oxford University Computing Laboratory, PRG, July 1979.

[30] ___________. "Monitors: an Operating System Structuring Concept," CACM, 17, 10, October, 1974.

[31] ___________. "Some Properties of Predicate Transformers," JACM, 25, 3, July, 1978.

[32] ___________. "Towards a Theory of Communicating Sequential Processes," Oxford University Computing Laboratory, PRG, February 1979.

[33] ___________ and Wirth, N. "An Axiomatic Definition of the Programming Language PASCAL," Acta Informatica, 2, 1975.

[34] Howard, J. H. "Proving Monitors," CACM, 19, 5, May, 1976.

[35] Igarishi, S. et al. "Automatic Program Verification I: A Logical Basis and its Implementation," Acta Informatica, 4, 1975.

[36] Kahn, G. "The Semantics of A Simple Language for Parallel Processing," Information Processing 74, North-Holland Publishing Company, 1974.

[37] ___________, [ed.]. "Semantics of Concurrent Computation: Proceedings of the International Symposium, Evian, France, 1979" Lecture Notes in Computer Science, Volume 70, 1979.

[38] ___________ and MacQueen, D. B. "Coroutines and Networks of Parallel Processes," IRIA-Laboria Report No. 202, 1977.

[39] Keller, R. M. "Semantics of Parallel Program Graphs," Department of Computer Science Technical Report UUCS-77-110, University of Utah, 1977.

[40] King, J. C. "Program Correctness: On Inductive Assertion Methods," Computer Science Department, IBM San Jose Research Report RJ2525(32955), 1979.

[41] Lamport, L. "Proving the Correctness of Multiprocess Programs," IEEE Transactions and Software Engineering, SE-3, 2, March, 1977.

[42] ___________. "'Sometime' is Sometimes 'Not Never'," Report CSL-86, SRI International, 1979.

[43] Lauer, P. E. "COSY: A System Specification Language Based on Paths and Processes," Computing Laboratory Report Number 136, University of Newcastle upon Tyne, 1979.

[44] Levin, G. M. "A Proof Technique for Communicating Sequential Processes," Computer Science Department Technical Report TR79-401, Cornell University, 1979.

[45] Lynch, N. A. and Fischer M. J. "On Describing the Behavior and Implementation of Distributed Systems," in Kahn, G. [ed.], Semantics of Concurrent Computation, 1979.

[46] Manna, Z. "Mathematical Theory of Partial Correctness," JCSS, 5, 3, June 1971.

[47] Martin, A. J. "An Axiomatic Definition of Synchronization Primitives," Philips Research Laboratory, Eindhoven, Netherlands, 1979.

[48] Milne, G. and Milner, R. "Concurrent Processes and their Syntax," University of Edinburgh, 1977.

[49] Milner, R. "Flowgraphs and Flow Algebras," Department of Computer Science Internal Report CSR-5-1977, University of Edinburgh, 1977.

[50] Milner, R. "Processes: a Mathematical Model of Computing Agents," Logic Colloquiam 1973, North-Holland Publishing Company, 1973.

[51] Milner, R., Gordon M. J. and Wadsworth C. P. "Edinburgh LCF", Lecture Notes in Computer Science, Volume 78, 1979.

[52] Mizell, D. "Verification and Design Aspects of 'True Concurrency'," Proceedings of the Fifth Annual ACM Symposium on the Principles of Programming Languages, 1978.

[53] Morris, J. H. and Wegbreit, B. "Subgoal Induction," CACM, 20, 4, April, 1977.

[54] Nauer, P., [ed.], "Revised Report on the Algorithmic Language ALGOL-60," CACM, 6, 1, January, 1963.

[55] Ogden, W. F. et al. "Complexity of Expressions Allowing Concurrency," Proceedings of the Fifth Annual ACM Symposium on the Principles of Programming Languages, 1978.

[56] Owicki, S. "Axiomatic Proof Techniques for Parallel Programs," Ph.D. Thesis, Computer Science Technical Report TR75-251, Cornell University, 1975.

[57] ___________, "Specifications and Proofs for Abstract Data Types in Concurrent Programs," Departments of Electrical Engineering and Computer Science, Technical Report 133, Stanford University, 1977.

[58] ___________ and Gries, D. "An Axiomatic Proof Technique for Parallel Programs I," Acta Informatica, 6, 1976.

[59] ____________ and ____________. "Verifying Properties of Parallel Programs: An Axiomatic Approach," CACM, 19, 5, May, 1976.

[60] Pnueli, A. "A Simple Model of Concurrent Programs," in Kahn, G, [ed.], Semantics of Concurrent Computation, 1979.

[61] ____________. "The Temporal Logic of Programs," 18th IEEE Symposium on Foundations of Computer Science, 1977.

[62] Pratt, V. R. "Dynamic Logic," M.I.T., 1978.

[63] ____________. "A Logic of Action," M.I.T., 1978.

[64] ____________. "A Near-Optimal Method for Reasoning About Action," M.I.T. 1978.

[65] ____________. "A Practical Decision Method for Propositional Dynamic Logic," Proceedings of the Tenth Annual ACM Symposium on the Theory of Computing, 1978.

[66] ____________. "Process Logic," M.I.T., 1978.

[67] Prawitz, D. Natural Deduction, Stockholm: Almquist and Wiksell, 1965.

[68] Rabin, M. "Probabalistic Algorithms," in Traub, J, [ed.], Algorithms and Complexity, New York: Academic Press, 1977.

[69] Riddle, W. E. "Behavior Modelling During Software Design," IEEE Transactions on Software Engineering, SE-4, 4, July, 1978.

[70] ____________. "An Introduction to the DREAM Software Design System," Software Engineering Notes, 2, 4, July 1977.

[71] Schneider, F. B. "Synchronization in a Distributed Environment," Computer Science Department Technical Report TR79-391, Cornell University, 1979.

[72] Shaw, A. C. "Software Descriptions with Flow Expressions," IEEE Transactions on Software Engineering, SE-4, 3, May, 1978.

[73] ____________. "Software Specification Languages Based on Regular Expressions," ETH Institut fur Informatik Report 31, 1979.

[74] van Lamsweerde, A. and Sintzoff M. "Formal Derivation of Strongly Correct Parallel Programs," Report R338, M.B.L.E. Research Laboratory, 1976.