INFORMATION TO USERS

This reproduction was made from a copy of a manuscript sent to us for publication and microfilming. While the most advanced technology has been used to photograph and reproduce this manuscript, the quality of the reproduction is heavily dependent upon the quality of the material submitted. Pages in any manuscript may have indistinct print. In all cases the best available copy has been filmed.

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

1. Manuscripts may not always be complete. When it is not possible to obtain missing pages, a note appears to indicate this.

2. When copyrighted materials are removed from the manuscript, a note appears to indicate this.

3. Oversize materials (maps, drawings, and charts) are photographed by sectioning the original, beginning at the upper left hand corner and continuing from left to right in equal sections with small overlaps. Each oversize page is also filmed as one exposure and is available, for an additional charge, as a standard 35mm slide or in black and white paper format.*

4. Most photographs reproduce acceptably on positive microfilm or microfiche but lack clarity on xerographic copies made from the microfilm. For an additional charge, all photographs are available in black and white standard 35mm slide format.*

*For more information about black and white slides or enlarged paper reproductions, please contact the Dissertations Customer Services Department.

UMI Dissertation Information Service

University Microfilms International
A Bell & Howell Information Company
300 N. Zeeb Road, Ann Arbor, Michigan 48106
Sobel, Ann E. Kelley

MODULAR VERIFICATION OF CONCURRENT SYSTEMS

The Ohio State University

University Microfilms International

300 N. Zeib Road, Ann Arbor, MI 48106
Modular Verification of Concurrent Systems

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of the Ohio State University

By

Ann E. Kelley Sobel, B.S., B.S., M.S.

§ § § § §

The Ohio State University
1986

Dissertation Committee:
N. Soundararajan
W. F. Ogden
J. S. Gourlay

Approved by

Department of Computer & Information Science

Advisor
DEDICATION

To my Aunt, Mildred F. Bonebrake (1913–1984), whose determination in life was an inspiration to all whom she touched and who instilled in me the confidence to achieve my goals.
ACKNOWLEDGEMENTS

Many people at Ohio State University contributed in one way or another to this thesis. Bill Ogden and John Gourlay, as members of my reading committee, provided me with many useful comments and suggestions along with the insight to the depth of my research. The inhabitants of the fourth floor supported me in various ways during my graduate career. To these people and to the faculty, staff, and students of the Ohio State Computer and Information Science Department, I express my gratitude.

I want to give special thanks to my dearest friend, Michael Kaelbling. He provided me with professional and emotional support. He contributed greatly by reading various drafts and providing numerous comments. I also wish to especially thank Lawrence Sobel whose support and understanding helped me write this thesis.

There are no words to express my immense gratitude to Neelam Soundararajan, my advisor. He introduced me to program verification. His advice, suggestions, and ideas form an integral part of this thesis. I feel very privileged to have known and worked with him.

My research was supported by a number of agencies and I gratefully acknowledge their generosity. My graduate education was funded primarily by teaching assistantships in the Ohio State Computer and Information Science Department. I also received additional support by fellowships from the National Science Foundation (ECS-8404725) and the Apple Computer Company.
VITA

September 9, 1958 ............................... Born – Akron, Ohio

1980 .................................................. B.S., Statistics, Cum Laude
Certificate in Computer Science
University of Akron
Akron, Ohio

1981 .................................................. B.S., Applied Mathematics
University of Akron

1980-1984 ........................................... Teaching Associate
Dept. Computer & Information Science
The Ohio State University
Columbus, Ohio

1982 .................................................. M.S., The Ohio State University

1985-1986 ........................................... Research Associate
Dept. Computer & Information Science
The Ohio State University

PUBLICATIONS

FIELDS OF STUDY

Major Field: Computer and Information Science

- Studies in Theory of Programming Languages.  
  Dr. Neelam Soundararajan

- Studies in Computer Architecture.  
  Dr. Bruce W. Weide

  Dr. Edwin R. Tripp
# TABLE OF CONTENTS

DEDICATION .......................................................................................................................... ii

ACKNOWLEDGMENTS ........................................................................................................ ii

VITA ........................................................................................................................................ iv

NOTATION ................................................................................................................................ x

CHAPTER  | PAGE
--- | ---
I. INTRODUCTION .................................................................................................................. 1
   1. History of Verification ........................................................................................................ 2
      1. Floyd .............................................................................................................................. 3
      2. Hoare ............................................................................................................................ 3
   2. Verification and Concurrency ............................................................................................. 6
   3. Related Works ..................................................................................................................... 7
      1. Owicki & Gries ................................................................................................................. 7
      2. Apt, Francez, & de Roever ............................................................................................. 8
      3. The Issue ....................................................................................................................... 9
   4. Main Contributions of the Thesis ....................................................................................... 10
   5. Overview and Organization of the Thesis ......................................................................... 13

II. CSP ...................................................................................................................................... 22
   1. Introduction ....................................................................................................................... 22
   2. Axioms and Rules .............................................................................................................. 23
      1. Output Command ............................................................................................................ 23
      2. Input Command .............................................................................................................. 24
      3. Guarded Selection .......................................................................................................... 24
      4. Guarded Repetition ....................................................................................................... 25
      5. Parallel Composition ..................................................................................................... 25
   3. Set Partition Example ........................................................................................................ 26
   4. Conclusion ......................................................................................................................... 29

vi
## III. MULTIPLE EXIT STATEMENTS

1. EXIT Statement .......................................................... 31
2. Axiomatic Notation .................................................. 31
3. Axioms and Rules .................................................. 32
   1. Unconditional EXIT .......................................... 32
   2. Conditional EXIT ............................................. 32
   3. Alternation .................................................. 34
   4. LOOP ...................................................... 35
   5. WHILE .................................................. 35
4. Lookup Example .................................................. 36
5. Exception Handler Statements ............................... 37
6. Exception Handler Rules ........................................ 39
   1. RAISE Statement ........................................... 40
   2. Exception Handler .......................................... 40
   3. Block Statement ........................................... 41
   4. Procedure Call ........................................... 42
   5. Procedure .................................................. 43
7. Search Example .................................................. 43
8. Summary ............................................................. 44

## IV. TASKS .......................................................... 47

1. Ada ................................................................. 47
2. Axiomatic Notation ................................................ 54
3. TERMINATE Statement ........................................ 55
4. Entry Call Statement ............................................ 55
5. ACCEPT Statement .............................................. 56
6. Selective Wait ..................................................... 57
   1. Selective Wait Without ELSE ....................... 57
   2. Selective Wait With ELSE .............................. 57
   3. Conditional Entry Call .................................. 58
   4. Timed Entry Call .......................................... 58
7. Task Without Exception Handler .............................. 59
8. Task With Exception Handler .................................. 59
9. Parallel Composition ............................................ 60
10. Buffer Example .................................................. 60
11. Summary ............................................................ 64

## V. DISTRIBUTED PROCESSES LANGUAGE .................................................. 66

1. DP ................................................................. 66
## NOTATION

<table>
<thead>
<tr>
<th>Symbols</th>
<th>Sample Usage</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \vdash )</td>
<td>( \vdash S )</td>
<td>It is true that the statement S is provable</td>
</tr>
<tr>
<td>( \models )</td>
<td>( \models S )</td>
<td>It is true that the statement S is operationally valid</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>( A \Rightarrow B )</td>
<td>A implies B; A( \Rightarrow )B</td>
</tr>
<tr>
<td>( \forall )</td>
<td>( \forall N )</td>
<td>Universal quantifier</td>
</tr>
<tr>
<td>( \exists )</td>
<td>( \exists h )</td>
<td>Existential quantifier</td>
</tr>
<tr>
<td>( x )</td>
<td>( p^x )</td>
<td>Replace each occurrence of x with e in the assertion p</td>
</tr>
<tr>
<td>( h )</td>
<td>( h_i )</td>
<td>A sequence</td>
</tr>
<tr>
<td>( \langle \rangle )</td>
<td>( \langle \text{call},i,k,\overline{u},\overline{v} \rangle )</td>
<td>An element of a sequence</td>
</tr>
<tr>
<td>( \circ )</td>
<td>( h \circ \langle \text{call},i,k,\overline{u},\overline{v} \rangle )</td>
<td>Concatenate the element to the right end of h</td>
</tr>
<tr>
<td>( [ ] )</td>
<td>( h[i : k] )</td>
<td>The sequence consisting of the ( i^{th} ) through ( k^{th} ) elements of h</td>
</tr>
<tr>
<td>#</td>
<td># h</td>
<td>The length of h</td>
</tr>
<tr>
<td>( \subseteq )</td>
<td>( h_1 \subseteq h_2 )</td>
<td>( h_1 ) is an initial subsequence of ( h_2 )</td>
</tr>
</tbody>
</table>
CHAPTER I

Introduction

There are two views of programming. In the old view, the purpose of our programs is to instruct our machines; in the new one, it is the purpose of our machines to execute our programs.

Edsger W. Dijkstra

From the earliest days of computer science, programmers have been faced with the task of demonstrating that their programs achieve some intended purpose. This task has come to be known as program verification. In order to verify that a program is correct, one must prove that the instructions of the program meet the specification of that program.

Such a formal approach is essential when working with distributed programs, since human intuition has been demonstrated as unreliable when dealing with several processes which are executing simultaneously and interacting extensively. The problem stems from the fact that concurrent processes can exhibit extremely complicated interaction patterns and neither informal reasoning nor testing is reliable enough to establish their correctness. In fact, the inherent complexity of even comparatively simple distributed programs makes the need for a formal theory of such programs obvious.

During the last ten years, a number of authors have proposed systems that can be used for proving correctness of distributed programs; Owicki & Gries[32] and Apt, Francez, and de Roever[4] among others. These verification techniques allow one to prove properties of individual processes using global assumptions about the
behavior of the remaining processes in the program. As a result, their proof rule for parallel composition requires one to justify these global assumptions before drawing any conclusions regarding the correctness of the entire program. Often, establishing the correctness of these global assumptions is the most difficult part of the proof and presents a serious obstacle to hierarchical program development.

In this thesis, we will develop a new approach to the verification of concurrent systems. Our approach is modular and supports compositional development of programs, since the proofs of each individual process of a program are completely isolated from all others. We will illustrate the generality of this approach by applying it to a representative set of contemporary concurrent programming languages. In addition, we will also show how the approach may be used to deal with a number of other constructs that have been proposed for inclusion in concurrent languages. These results allow us to argue that our approach is universal and can be used to design proof systems for any concurrent language.

Since the intuition underlying our approach is not solely axiomatic, it is not surprising that we can use it to define the operational semantics of concurrent languages. We explain how this may be done for one of the languages we consider. We then indicate how we can demonstrate the soundness and completeness of our proof system for this language with respect to this operational model. A similar development may be used to show soundness and completeness of other systems designed using this approach.

1.1 History of Verification

Few people had worked in the area of program verification prior to the late 1960's. However, two important articles written in the 1960's had a profound impact on the field.
1.1.1 Floyd

A significant piece of work was presented by Robert Floyd[18] at a meeting of the American Mathematical Society in 1967. Floyd set the cornerstone for verification with the idea of attaching assertions to program statements to describe the behavior of these statements. He modeled programs with flowcharts: the nodes represent program statements and the arcs represent the flow of control. The approach attached assertions to the edges of a flow chart with the intent that each assertion would be true whenever execution reached that edge during the execution of the program. When the statement was a loop, Floyd placed an assertion \( P \) on an arbitrary edge of the cycle and called it a cut point. He would then prove that if the execution of the loop began at the cut point with \( P \) being true and during its execution the cut point was reached again, \( P \) would still be true at that point. Thus was born the concept of a loop invariant.

1.1.2 Hoare

C.A.R. Hoare took to heart Floyd's suggestion that a specification of proof techniques could provide an adequate definition of a programming language[24]. In this article, Hoare defined a small programming language in terms of a logical system of axioms and inference rules for proving the partial correctness of programs.

Hoare uses the notation

\[
\{P\} \text{ } S \{Q\}
\]

where \( P \) and \( Q \) are assertions, and \( S \) is a programming statement. The intended meaning is the following: if \( P \) is true before \( S \) is executed and if \( S \) terminates, then \( Q \) will be true afterwards. The assertions \( P \) and \( Q \) are both truth valued expressions over the domains of the program variables. \( P \) is called a precondition of \( S \) and \( Q \) a postcondition. A precondition may also be called the "input" assertion
and a postcondition the "output" assertion.

If the execution of S fails to terminate, then \{P\} S \{Q\} is trivially valid for any assertions P and Q. Thus, a proof of correctness of a program using this method is actually only a proof of partial correctness in that its correctness is subject to the assumption of termination. If we modify the definition of \{P\} S \{Q\} to insist that S must terminate and that Q be true when it does, then we would be verifying the total correctness of S. In general, it is easier to prove partial correctness than total correctness since the latter must include a proof that the program will indeed terminate.

The approach proposed by Hoare to the formal definition of the semantics of programming language constructs has come to be known as the axiomatic approach. The idea is to specify the semantics of a programming language construct by providing a precise rule that can be used to derive the postcondition of any occurrence of the construct, given its precondition. Verification of a given construct in a program is performed by deriving output assertions using information from the input assertion and the rule corresponding to the construct. The output assertion may be used as the input assertion to a subsequent programming construct. Hence, verification proceeds by induction on the structure of a program.

Hoare presents four rules for deriving the pre- and postconditions of program statements (one corresponding to each of the three statements that he considers and a general "rule of consequence"). They are as follows:

**Assignment**

\[ \{P \} x := e \{P\} \]

**Consequence**

\[ \{P\} S \{Q\}, \; P \supset P', \; Q' \supset Q \]

\[ \{P\} S \{Q\} \]
Composition

\[
\begin{align*}
\{P\} S_1 \{P'\}, \{P'\} S_2 \{Q\} \\
\{P\} S_1; S_2 \{Q\}
\end{align*}
\]

Iteration

\[
\begin{align*}
\{P \land B\} S \{P\} \\
\{P\} \text{While B Do}\ S \{P \land \neg B\}
\end{align*}
\]

The formula \(P^x\) represents the assertion \(P\) with all free occurrences of \(x\) being replaced by the expression \(e\). Three of these rules are inference rules and they take the form

\[
\begin{align*}
a_1, a_2, \ldots, a_n \\
b
\end{align*}
\]

The notation means that if \(a_1, a_2, \ldots, a_n\) are provable, then we may conclude \(b\).

In order to define a formal semantics for a program, we need to be able to establish all the correct formulae about that program through the use of an appropriately selected set of axioms and proof rules. This means we expect of such a proof system that whenever a formula is operationally valid, it is provable using this system of axioms and rules of inference. A system satisfying this requirement is called \(\text{complete}\). More importantly, we expect in such a proof system that no operationally incorrect formula is provable. In this case, the system is called \(\text{sound}\).

Hoare's article attempts to deal directly with the programming problem. It suggests that defining a programming language in terms of how to prove a program correct, instead of how to execute it, might lead to simpler designs. It is this comprehensive evaluation of the possible benefits of adopting the axiomatic approach to language definition, both from a formal foundations of computing perspective and from a software engineering perspective, that made this article so significant.
1.2 Verification and Concurrency

In the early seventies, research concentrated on the development and application of techniques for formal verification of sequential programs. Now, due to the dramatic rise in the production of distributed computing systems, it is imperative that similar formal methods also be employed in the construction and verification of concurrent programs.

Concurrent programs differ from sequential programs in that they contain a number of distinct processes which can conceptually operate simultaneously. In reality, these processes can either be executed by permitting them either to share one or just a few processors or each to have its own individual processor. The former approach is referred to as multiprogramming and is supported by an operating system kernel that multiplexes the processes on the processor(s). The latter approach is referred to as multiprocessing if the processors share a common memory as in a multiprocessor, or as distributed processing if the processors are connected only by a communications network.

In order to cooperate, concurrently executing processes must be able to both communicate information and synchronize their activity. Communication permits the execution of one process to influence the execution of another. Interprocess communication is based either on using shared variables (variables that can be referenced by more than one process) or on passing messages. The goal of synchronization is to constrain the ordering of events in a system or to prevent processes from interfering with each other. The programmer employs a synchronization mechanism to delay execution of a process in order to satisfy these objectives.

Despite the wide variety of concurrent programming languages which have been proposed, each can be viewed as belonging to one of three classes: procedure oriented, message oriented, or operation oriented.[1]

The transformation of verification techniques from sequential to parallel pro-
grams is far from straightforward. When concurrency was introduced to the computing community, it caused a significant amount of confusion. Part of this confusion was generated by the fact that concurrency simultaneously introduced the concept of nondeterminacy. Therefore, the verification of parallel programs is complicated by the nondeterministic manner in which parallel programs can affect each other. The correctness of parallel programs depends not only on the functionality of the individual processes but also on the interactions, or the lack of interactions, among those processes. When verifying parallel programs, we may not make assumptions about the way in which the parallel execution is implemented. In particular, nothing can be assumed about the rates at which execution of the processes progress. Therefore, it is not possible to identify process interruption or process interference points caused by the rates of progress of the processes.

1.3 Related Works

Over the past few years, a number of attempts have been made to define the semantics of various concurrent programming languages [see references]. The most well known definitions of the axiomatic semantics of concurrent languages seem to be those given by Owicki & Gries[32] and Apt et al.[4].

1.3.1 Owicki & Gries

Susan Owicki’s work on the correctness of parallel programs has formed the basis for most of the later verification techniques for parallel programs. She extended Tony Hoare’s axiomatic system for proving the partial correctness of sequential programs to include the parallelism of a shared variable language[32].

The key idea behind Owicki’s approach was the notion of “interference freedom”: treat each process independently, but weaken the assertions in the proof of
each process so that the assertions cannot be falsified by the actions of the other processes. Suppose, for example, we have the assignment "x := 1" in one process, and the assignment "x := 2" in another process. When considering the first process, we would use the assertion \( \{ x = 1 \lor x = 2 \} \) (rather than \( \{ x = 1 \} \)) as the postcondition of the assignment \( x := 1 \). Thus, although in a formal sense we consider each process in isolation, in reality the proof of each process is obtained by taking into account how the actions of the other processes may affect the proof of the first process.

We must then verify that the proof of each process is indeed free of interference from the other actions of the other processes. Thus the parallel composition rule of the Owicki & Gries system is:

\[
\text{proofs of } \{p_i\} S_i \{q_i\}, i = 1, \ldots, n \text{ "do not interfere with each other"}
\]

\[
\{p_1 \land \ldots \land p_n\} \text{ Cobegin } S_1 || \ldots || S_n \text{ coend } \{q_1 \land \ldots \land q_n\}
\]

I.3.2 Apt, Francez, and de Roever

The Apt, Francez, and de Roever verification technique for Hoare's language CSP[^4] is similar in spirit to the Owicki & Gries method. Initially, proof outlines for each process are made. If one can prove that the possible interactions (due to parallel execution) do not invalidate the sequential proofs, one can conclude that the postcondition of the parallel execution of the processes is the conjunction of the postconditions of each individual process.

The major difference between the Apt system and the Owicki & Gries system is due to the difference in the way that CSP processes interact with each other compared to the way processes interact with each other in the Owicki & Gries language: the primary means of process communication in CSP is an input/output command in which a process \( P_1 \) receives a value from a process \( P_2 \) when \( P_1 \) executes an input command and \( P_2 \) executes a matching output command. The subsequent
behavior of $P_i$ may (and usually will) depend on the particular value which $P_j$ transmitted to $P_i$. Consequently, in a proof of $P_i$, using the Apt system, one would make assumptions about the value that $P_j$ would send to $P_i$ in response to $P_i$'s input request. These assumptions must then be validated in the parallel composition rule using what are called "cooperation" checks:

$$\text{proofs of } \left\{ P_i \mid P_j \{q_i\} \right\} \text{ cooperate with each other}$$

$$\left\{ P_1 \land \ldots \land P_n \right\} \left[ P_1 \| \ldots \| P_n \right\} \left\{ q_1 \land \ldots \land q_n \right\}$$

1.3.3 The Issue

In all these systems, the proofs of the individual processes are not isolated from each other. The properties of each individual process are proved using assumptions about the behavior of the remaining processes in the program. In order to arrive at a postcondition for the whole program, one must verify that these proofs of the individual processes do not invalidate each other ("do not interfere with each other" in the work of [32] and "cooperate with each other" in the work of [4]). The proof of non-interference is often the most difficult part of the proof. Moreover, the non-interference test requires the use of the complete proof outlines for all of the processes within the program. Therefore, a modification to any process could potentially require the modification of the proofs of all the other processes defined within the program. Clearly, a system in which individual processes can be proved correct in isolation and then combined in a parallel construct without separate non-interference proofs would have many advantages.

Another problem is that the non-interference proof method does not seem to lend itself to hierarchical program development. Processes are proven in isolation, but using assumptions about the behavior of other processes with which they may run concurrently. A demonstration of non-interference can only be given after all
process proofs are completed. For large programs requiring hierarchical development, postponing the check for non-interference until the end of development can obviously lead to very costly problems if the non-interference proof does not ultimately succeed. Moreover, the number of non-interference tests which are required grows rapidly with the number of processes and I/O statements. The proof method of Apt et al. suffers from similar problems when used in hierarchical program development. One can develop the individual processes, providing that details of the communication between processes are known. Assumptions about the behavior of the other processes are made but not recorded in a specification. Validation of the assumptions by performing a cooperation proof is delayed until after development.

The approaches given in Owicki & Gries[32] and Apt et al.[4] have been extended to other concurrent programming languages by Gerth et al.[19,20] and Barpringer & Mearns[8]. Although this seems to argue for the universality of the Owicki & Gries approach, the transformation of the approach from one concurrent language to another is not straightforward and has resulted in rather complex systems since it required the introduction of multiple control point assertions and additional interference tests.

1.4 Main Contributions of the Thesis

The main aim of this thesis is to develop a modular axiomatic semantics that can be used to define the semantics of various concurrent systems. The approach is modular or compositional: it is possible to infer a specification for the whole program from the specifications of its component processes alone, without additional knowledge of the internal structure of these components. This goal was also the motivation for the development of the axiomatic semantics of CSP defined by Soundararajan in [35,38,39] and for a shared variable language defined in [37]. In order to examine the universality of this modular approach to verification, the se-
mantics of the concurrent programming languages Ada and Distributed Processes will be defined. These languages were selected because the former has practical merit while the latter is a theoretically interesting language. These languages were also selected so that this approach would be applied to at least one language from each of the three classes of concurrent languages defined by Andrews & Schneider[1].

We argue that this modular approach could and should be used to define the semantics of any concurrent programming language. To provide further justification for this claim of universality, we examine a number of other constructs (not complete languages) that have been proposed for inclusion in concurrent languages and explain how our approach can deal with them. Lastly, we perform the important task of demonstrating the soundness and completeness of our axiomatics.

The modularity of our approach means that the proofs of the individual processes of a program are completely independent of each other. This allows us to consider an individual process without making assumptions about the behavior of the other processes. In fact, the most important idea underlying our work is that the individual processes of a concurrent program must be dealt with in isolation if we are to avoid getting lost in the complex interactions between the processes. Since the semantics of a process is independent of the rest of the program, we have the flexibility to change any one process without causing a change in the verification of the other processes in the program. Once we obtain the individual process proofs, we only need to combine the postconditions of these processes in order to describe the behavior of the entire program.

Another important aspect of our approach is that it will allow us to give semi-formal proofs of program correctness by omitting most of the intermediate assertions and other tedious details of a strictly formal proof. This aspect of the proposed approach is likely to increase its appeal to programmers who might have little or no background in formal logic and who may be unwilling to supply complete formal
proofs of their programs. From this point of view, the proposed system resembles the axiomatic systems for sequential programs where semi-formal proofs are often given by specifying only the assertions (i.e. loop invariants) that hold at key points in the program. (Such semi-formal proofs are harder to give in the Owicki & Gries and Apt, Francez, and de Roever systems since the interference freedom/cooperation check requires complete proofs with all intermediate assertions.)

In order to define the semantics of a concurrent program, we need to define the semantics of the individual processes and then specify how to combine these individual semantics to obtain the semantics of the entire program. Intuitively, the semantics of an individual process will be the behavior of the process which would be seen by an external observer of the process. An external observer of a process will be able to see not only the final values of the local variables of the process but also a sequence comprised of all communications with other processes of the program in which the process participates. Given the i\textsuperscript{th} process \( P_i \) of a program, we will associate with \( P_i \) a sequence valued variable \( h_i \) that will record the communications in which \( P_i \) participates. Thus, the semantics of the process \( P_i \) will be a relation that maps the initial state of \( P_i \) to the final state of \( P_i \) and to the sequence \( h_i \) of communications that \( P_i \) participated in during its execution.

In summary, there are two important ideas underlying our modular approach to concurrency:

1. The semantics of a process of a concurrent program is essentially the behavior of the process as seen by an external observer of the program.
2. When obtaining the semantics of a process \( P_i \) in a program, we should make no assumptions about the behavior of the other processes.

A comparison between the proof systems given in \([4,8,19,20,32]\) and the contents of this thesis will demonstrate that our modular approach is simpler to use than
these other systems since it allows us to examine the individual processes of a program independently of each other rather than forcing us to look at all of them simultaneously. Moreover, this approach is easily transformed to the two concurrent languages in this thesis and the languages given in [35,37]. Also, it can easily be extended to the concurrent language constructs that have not been examined in this thesis. Therefore, it is universally applicable to any concurrent language construct. Since the majority of existing concurrent languages combine a small number of constructs, we believe that the semantics of the individual constructs defined by our approach can be combined to define the semantics of future concurrent programming languages.

1.5 Overview and Organization of the Thesis

One of the most important aspects of Hoare’s work was the modularity of the approach. Thus, for instance, the semantics of the statement “$S_1; S_2$” is arrived at by establishing the external behavior of $S_1$ and of $S_2$ (in terms of input/output assertions) and combining these two behaviors to derive the external behavior of “$S_1; S_2$”. Notice that in performing this last step, we do not examine the details of the individual statements $S_1$ and $S_2$. Therefore, the key to good axiomatics for any language is to decide first, for each of the constructs in the language, what the external behavior of the construct is and then to capture this behavior in an appropriate axiom. Next, for each method of composing constructs to form larger constructs, one must decide how the external behavior of the larger construct can be obtained from that of its components and then capture this composition of behaviors in an appropriate proof rule.

When we try to extend this modular approach to concurrent programs, we face the following problem: the behavior of any one process of a concurrent program depends upon the behavior of the other processes in the program since they are all
running concurrently and will, in general, affect each other's behavior. It was this problem that forced Owicki & Gries to abandon modularity (at least in an informal sense). The Owicki & Gries system tries to obtain the behavior of each process in a concurrent program by simultaneously considering how the other processes will affect the behavior of this process.

We illustrate how this modular approach can be extended to concurrent programs by specifying appropriate axioms and rules of inference for CSP[35]. We chose to begin our presentation with CSP since it is a simple concurrent programming language. The fundamental idea underlying our approach is, of course, that each process must be considered in isolation from the other processes in the program. In order to express the externally observed behavior of a process we associate, with each process, a sequence that will record all the communications between this process and other concurrently executing processes. Therefore, every input and output command that this process executes will be recorded in this sequence. Consider, for instance, an output command in the process $P_i$ that names the receiving process $P_j$ and sends the value $t$. This command is recorded on $P_i$'s sequence, $h_i$, by concatenating the element

$$(i, j, t)$$

The execution of the corresponding input command in the process $P_j$ will be recorded on $h_j$ by concatenating the element

$$(i, j, X)$$

where $X$ is the value sent by $P_i$. However, since we are considering $P_j$ in isolation, we don't know what value $P_j$ will receive. Hence, the axiom corresponding to the input command requires us to consider all possible values for $X$.

When we combine (using the parallel composition rule) the postconditions of $P_j$ and all other processes, the parallel composition rule will allow us to restrict the
value received by the input command in \( P_f \) to be exactly the value sent by \( P_i \). It is the predicate \( COMPAT' \) in the postcondition of the parallel composition rule that performs this restriction. \( COMPAT' \) essentially requires that the communications between the various processes be recorded in the sequences associated with the individual processes in a mutually consistent fashion.

We have omitted important details in the previous discussion (and we will, of course, go into them in Chapter II), but the key points have been made:

1. The externally visible behavior of CSP processes, i.e. the sequence of values output from a process to another process and the sequence of values accepted by a process from another process, are all recorded in the sequence associated with the process (this sequence is our way of specifying the external behavior of the process).

2. When obtaining the externally visible behavior of a CSP process, we consider it in isolation from the others, and only at the parallel composition stage do we put these behaviors together to obtain the behavior of the entire program.

Chapters IV and V form the heart of the thesis. In these chapters, we deal with the problem of defining a modular axiomatics of two different types of concurrency: in Chapter IV, we consider Ada "tasks" and present an appropriate set of axioms and rules of inference corresponding to the various aspects of Ada tasks. However, before we can discuss the semantics of an Ada task, we must first examine the sequential statements contained within a task which include a variety of multiple exit statements. This raises the problem of maintaining modularity in our axiomatics when it is well known that the easiest way to destroy modularity is the use of control transfers. We can then ask whether a simple, modular, axiomatic approach can be used to define the semantics of multiple exit statements.

We answer this question positively in Chapter III by specifying appropriate
axioms/rules of inference for Ada's *EXIT* and related statements. The key to this definition is the realization that the behavior (as seen by an external observer) of a construct $S$ that contains an *EXIT* (that transfers control out of the construct) is the following: $S$ starts execution in some initial state (which will be visible to the external observer) and after a number of internal actions (all of which will be invisible to the external observer), $S$ will finish execution in one of two ways:

1. the "normal" completion (without execution of an *EXIT*)
2. an "exit" completion, when $S$ executes an *EXIT* statement that transfers control out of $S$.

In either case, the external observer will see the state in which $S$ finishes its execution. The external observer will also know whether $S$ finished "normally" or by executing an *EXIT* statement. Thus, an axiomatic description of $S$ should be given as

$$\{p\} S \{q\} \Sigma$$

where $p$ is the usual precondition satisfied by the initial state and $q$, the postcondition, will be satisfied by the final state if $S$ finishes normally. Lastly, $\Sigma$, the exit-condition, will have the following form:

$$\{L : r\}$$

and the interpretation that on any exit from $S$ via an *EXIT* $L$ statement, the exit-condition $r$ will be satisfied. Note that only *EXIT* statements that transfer control out of $S$ will be represented in $\Sigma$. If there is an *EXIT* statement in $S$ which transfers control to another part of $S$ (as might happen if $S$ contains a loop and a statement in the loop exits to the end of the loop after which execution continues with the rest of $S$), this will be an internal action and will, of course, be invisible to the external
observer. Hence, it should not, and will not, be represented in Σ.

In Chapter III, we also consider another set of Ada statements related to the\textit{EXIT}: the statements used to raise "exceptions", and the exception handlers. It is these statements that play an important role in defining the semantics of Ada tasks. The concepts used to define the semantics of the \textit{EXIT} statement will form the basis of capturing the axiomatic description of exceptions. We explain how proper axioms and rules of inference may be written for these statements. Chapter III also includes some simple examples to demonstrate the application of our axioms/rules for \textit{EXIT}s and exception handling.

In Chapter IV, we present an appropriate set of axioms and rules of inference corresponding to the various aspects of Ada tasks. As before with CSP, we associate, with each task, a sequence that will record all the communications between this task and other concurrently executing tasks. Thus every entry call that the task issues (to other tasks), as well as entry calls (from other tasks) that this task accepts, will be recorded on this sequence. Consider for instance a call to $T_j.A_k$ issued by $T_i$ (i.e. the task $T_i$ issues an entry call to the \textit{ACCEPT} statement $A_k$ in the task $T_j$) passing the input parameters $\vec{X}$. This call will be recorded by concatenating the element

$$(\text{call.to},i,j,k,\vec{X})$$

to $h_i$, the interaction sequence associated with the task $T_i$. Again, the return from $T_j.A_k$ to $T_i$ will be recorded by concatenating the element

$$(\text{return.from},i,j,k,\vec{Y})$$

with $\vec{Y}$ being the results returned by $T_j.A_k$.

The execution of the \textit{ACCEPT} statement in $T_j$,

$$\text{Accept } T_j.A_k(\vec{Z}:\text{in})$$
will be recorded by concatenating the element

$$(\text{call} j, k, Z)$$

where $Z$ are the values of the input parameter(s). Since we are considering $T_j.A_k$ in isolation, there is no way of knowing exactly what values $T_j.A_k$ will receive as its input parameters. Hence, the axiom corresponding to the $\text{ACCEPT}$ will require us to consider all possible values for $Z$ (the element concatenated to $h_i$ will, of course, have to match the value in $Z$). A similar situation occurs corresponding to a return from an entry call to $T_j.A_k$ issued by $T_i$: when considering $T_i$ in isolation, there is no way of knowing what values $T_j.A_k$ will return to $T_i$. Correspondingly, the axiom for entry calls will require us to consider all such possible values that can be returned by the called task.

The parallel composition rule will allow us to restrict the values for the parameters received by the $\text{ACCEPT}$ in $T_j$ to be exactly the values sent by the calling tasks. Similarly, this rule will allow us to restrict the values received by $T_j$ when its entry call to $T_i.A_k$ returns to be exactly the value(s) (as recorded in $h_i$) that $T_i.A_k$ will return to its caller. The predicate that does this is the $\text{COMPAT}$ predicate in the postcondition of the parallel composition of the tasks which is similar to the $\text{COMPAT}'$ of Chapter II.

One aspect of Ada that we do not deal with in our axiomatics is shared variables. We decided to ignore shared variables mainly because there doesn’t seem to be any (informal) description of exactly what these variables are, and exactly how they are shared between the various tasks.

In Chapter V, we show how our approach may be used to axiomatize a rather different type of concurrency than the one in Ada. Distributed Processes (DP), although some of its ideas were used in the design of Ada, exhibits a more complex form of concurrency than does Ada due, mainly, to the coroutine-like structure
present in individual DP processes. Thus we had to worry about two types of interactions when dealing with DP – the one between the processes of a program (and this type of communication is not unlike that in Ada programs) and the communication between the procedures of an individual DP process. Despite this added complexity, we are able to use our approach to define the axiomatics of DP. Specifically, the behavior of a procedure $Q_k$ of a DP process $P_i$ is characterized in terms of the sequence of communications (such as “suspensions” of $Q_k$ when the execution of $Q_k$ is suspended and control in $P_i$ transfers from $Q_k$ to another procedure of $P_i$, and the subsequent “resumption” of $Q_k$ when control returns to $Q_k$) between $Q_k$ and the other procedures of $P_i$, as well as between $Q_k$ and procedures of other processes (corresponding to procedure calls that $Q_k$ makes and their corresponding returns).

Corresponding to the two levels of concurrency, we have two rules of composition: the first one allows us to obtain the behavior of a process from the behaviors of its procedures; the second one allows us to obtain the behavior of a DP program given the behaviors of its processes. A natural observation we can make at this point is that there is no reason why we should consider only two levels of concurrency: our approach can easily be applied to deal with “hierarchical parallelism”. We comment further on this fact in Chapter VII. Chapters IV and V also contain proofs of correctness of some nontrivial Ada and DP programs using our axioms and rules of inference. The DP example is a parallel sort program taken from Gerth et al.[20] who also prove its correctness using their proof system. A comparison of our proof and that in [20] illustrates the basic differences between the two approaches.

The results of Chapters II, IV and V and the work in [37] bring us to the fundamental thesis of this dissertation: the approach of specifying the behavior of an individual process in terms of the sequence of interactions between this process and other processes external to this process, obtaining this behavior by considering the process in isolation from the other processes and then combining the behaviors of
the individual processes to obtain the behavior of the entire program, is universal and can be used to define the (partial correctness) axiomatics of any type of concurrency. We provide further justification for this claim in Chapter VI. We begin our argument with a quick summary of [37] that illustrates how the axiomatics of a simple concurrent language with shared variables may be defined using our approach. We then consider, in turn, each of the following constructs and briefly explain how each of them could be handled using our approach: FORK and JOIN primitives, nested monitor calls, path expressions, atomic transactions, and asynchronous message passing. We believe Chapters II, IV and V including the discussion in Chapter VI provide ample evidence to support the claim of universality of our approach.

In Chapter VII we turn to the important task of showing the soundness and (relative) completeness of our axiomatics. We argue that the approach used in [36] to demonstrate the soundness and completeness of the CSP proof system of [35] can be adapted to demonstrate the soundness and completeness of proof systems defined using our approach for any concurrent language. Briefly, the procedure of [36] for proving soundness and completeness proceeds as follows: define a ("non-standard") operational model that is similar in spirit to the axiomatic definition; thus, in this model one would specify the operational semantics of each process in isolation and then specify how the semantics of the individual processes may be combined to obtain the operational semantics of the whole program. (In the case of DP, there would be three steps: first define the operational semantics of the individual procedures of a process, then specify how these semantics may be combined to obtain the operational semantics of the process, and lastly specify how the semantics of the processes may be combined to obtain the operational semantics of the entire DP program.) The operational semantics of a process would, of course, be very non-standard, since it is not possible to "execute" a process of a concurrent program in isolation from the other processes. Therefore, the next step in this proof
of soundness and completeness demonstrates that the operational semantics defined above is equivalent to a standard operational semantics where all the processes of the program would be considered at the same time, as far as the final states that can be reached are concerned (or, in the case of DP, as far as the sequences of possible interactions between the various processes is concerned). The final step is to demonstrate that the proof system is both sound and complete with respect to the non-standard operational model. This part of the proof uses the approach of Apt et al.[3]. In fact, even our (non-standard) operational model is somewhat similar to that of [3], although they, of course, deal with a sequential language. In Chapter VII, we briefly show how this demonstration of soundness and completeness may be performed using the language DP. We do not deal with Ada (or any of the constructs considered in Chapter VI) since the method is precisely the same in each case.

Finally, we reach our concluding chapter where we briefly summarize the contributions of our work, reiterate the claim of universality of our approach, perform a comparison to related works, and mention further work that needs to be investigated.
CHAPTER II

CSP

We begin by illustrating how the modular approach created by Hoare for sequential programs can be extended to concurrent programs by specifying appropriate axioms and rules of inference for Communicating Sequential Processes [35]. An example of a set partitioning program will illustrate the use of this proof system.

II.1 Introduction

Using Communicating Sequential Processes (CSP), input and output commands specify the communication between two concurrently operating sequential processes. Consider a CSP program \([P_1||\ldots||P_n]\) that consists of the communicating sequential processes \(P_1,\ldots,P_n\). In order to prove the (partial) correctness of such a program using the modular approach proposed, we proceed as follows: first, prove the appropriate properties of the individual processes \(P_1,\ldots,P_n\) using the axioms and rules of inference applicable to the statements in the individual processes and then use the rule for parallel composition to combine the properties of the individual processes in order to prove the correctness of the entire program. With each process \(P_i\), we associate a communication sequence \(h_i\) which represents the sequence of all communications in which \(P_i\) has participated. Thus, \(h_i\) is a sequence of elements of the form \((i,j,u)\) which corresponds to the number \(u\) being sent from \(P_i\) to \(P_j\) by an output statement in \(P_i\), and \((j,i,v)\) which corresponds to the number \(v\) being
received by \( P_i \) from \( P_j \) in response to an input request in \( P_i \). There exists one other type of element, \((i,R,r)\), where \( R \) is a subset of \( \{1, \ldots, i-1, i+1, \ldots, n\} \). Such an element will indicate that a loop in \( P_i \) terminated because all the processes whose indices appear in \( R \) had terminated. The functions and predicates used in the following axioms/rules of inference can be found in the Appendix.

II.2 Axioms and Rules

**Skip**

\[
\{p\} \text{Skip} \{p\}
\]

**Assignment**

\[
\{p^x\} x := e \{p\}
\]

**Sequential Composition**

\[
\begin{align*}
\{p\} S_1 \{r\}; \{r\} S_2 \{q\} & \implies \\
\{p\} S_1; S_2 \{q\}
\end{align*}
\]

**Rule of Consequence**

\[
p \Rightarrow p', \{p'\} S \{q'\}, q' \Rightarrow q \implies \\
\{p\} S \{q\}
\]

II.2.1 Output Command

\[
\{p_{k_i}^{\lambda_j y_j}\} P_j \{y\} \{p\}
\]
An output command in the process \( P_i \) specifies as its destination the name of the receiving process \( P_j \). As far as \( P_i \) is concerned, the only effect of the output statement is to extend the communication sequence \( h_i \) by the element \((i,j,y)\).

**II.2.2 Input Command**

\[
\{\forall u. \rho_{u,h_i}(i,j,u)\} P_j x \{p\}
\]

An input command in the process \( P_i \) specifies as its source the sending process' name, \( P_j \). The effect of the input statement is to extend the sequence \( h_i \) by the element \((j,i,u)\) and assign the value \( u \) to the variable \( x \). The universal quantifier is needed since we have no idea what number \( P_j \) will send.

**II.2.3 Guarded Selection**

\[
\{p \wedge B(g_k)\} C(g_k); S_k \{q\}, k = 1, \ldots, m \\
\{p\}[g_1 \rightarrow S_1 | g_2 \rightarrow S_2 | \ldots | g_m \rightarrow S_m ] \{q\}
\]

Guards, \( g_k \), can be purely Boolean, an I/O command or both. Initially, the Boolean portion of the guard is evaluated. If it denotes false then the guard fails. If \( g_k \) does not contain a Boolean portion, then it is assumed to be true. An I/O command is executed only if and when a corresponding I/O command is executed. Guarded selection specifies the execution of exactly one of its constituent statements \( S_i \) by arbitrarily selecting a successfully executable guard. The notation \( B(g_k) \) and \( C(g_k) \) refer to the Boolean portion and the I/O portion of the guard \( g_k \) respectively.
II.2.4 Guarded Repetition

\[
\{ p \land \bigwedge_{k \in PB} \neg B(g_k) \} \Rightarrow q^k_{\eta_i} \quad \text{for } \eta_i = \{p_i \mid i \in IO : \exists k \in PB : B(g_k) \land D(g_k) = j \}, r
\]

\[
\{ p \land B(g_k) \} \quad C(g_k) ; S_k \{ p \}, \quad k = 1, \ldots, m
\]

\[
\{ p \} \ast [ g_1 \rightarrow S_1 | g_2 \rightarrow S_2 | \ldots | g_m \rightarrow S_m ] \{ q \}
\]

Guarded repetition specifies as many iterations as possible of its constituent selection command. Consequently, when all the guards fail, the repetitive command terminates. A distributed termination condition requires the termination of the repetitive command when all the sources named by the I/O portion of the guards have terminated. PB and IO are the set of indices of the purely Boolean guards and the I/O guards respectively. D(g_k) is the index of the process addressed in the I/O portion of g_k. In the first line, we assume that when the loop in P_i terminates, h_i will be extended by the element (i, T, r) where T is the set of indices of those processes with which P_i was "willing" to communicate. The second line ensures that p is the loop invariant.

II.2.5 Parallel Composition

\[
\{ p_i \land h_i = \varepsilon \} \quad P_i \{ q_i \}, \quad i = 1, \ldots, n
\]

\[
\{ p_1 \land \ldots \land p_n \} \quad [ P_1 || \ldots || P_n ] \{ q_1 \land \ldots \land q_n \land COMPAT'(h_1, \ldots, h_n) \}
\]

When the process P_i begins execution, we shall have h_i = \varepsilon. When all the processes finish, the sequences h_1, \ldots, h_n will be mutually "compatible". Therefore, if h_i contains the element (i, j, u) then h_j must contain the same element at an appropriate point in h_j.
II.3 Set Partitioning Example

The following is a program of Dijkstra for distributed partitioning of sets[35]. S and T are disjoint sets of integers with initial values \(S_0\) and \(T_0\) respectively. When the program finishes, the following predicate is satisfied:

\[
\{ |\text{MAX}(S) < \text{MIN}(T)| \land |S \cap T = S_0 \cap T_0| \land |S| = |S_0| \land |T| = |T_0| \}
\]

where \(|S|\) is the cardinality of \(S\).

The program \(P::[P_1||P_2]\) is as follows:

\(P_1::\)

\[
\begin{align*}
\text{mx} &= \text{MAX}(S); \\
P_2!\text{mx}; \\
S &= S - \{\text{mx}\}; \\
P_2?x; \\
S &= S \cup \{x\}; \\
\text{mx} &= \text{MAX}(S); \\
* [\text{mx} > x \rightarrow P_2!\text{mx}; \\
\quad S &= S - \{\text{mx}\}; \\
P_2?x; \\
S &= S \cup \{x\}; \\
\text{mx} &= \text{MAX}(S)]
\end{align*}
\]

\(P_2::\)

\[
* [P_1?y \rightarrow T := T \cup \{y\}; \\
\quad \text{mn} &= \text{MIN}(T); \\
P_1!\text{mn}; \\
T &= T - \{\text{mn}\}]
\]

It is assumed that \(S_0\) is not empty. During each iteration, the current maximum
of $S$ is exchanged with the current minimum of $T$. The loop terminates when the current maximum of $S$ is the same as the value last received from $P_j$.

We now annotate the process $P_1$; only assertions at key points in $P_1$ are stated with the assertions at the remaining points being left to the reader.

$$P_1:: \quad \{ h_1 = \epsilon \land S = S_0 \land S_0 \cap T_0 = \emptyset \}$$

$$m_x := \text{MAX}(S);$$

$$P_2!m_x;$$

$$S := S \setminus \{m_x\};$$

$$P_2?x;$$

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

$$m_x := \text{MAX}(S);$$

$$\ast\{p_1\}$$

$$[m_x > x \to P_2!m_x;$$

$$\quad S := S \setminus \{m_x\};$$

$$\quad P_2?x;$$

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

$$\quad m_x := \text{MAX}(S)]$$

$$\{p_1 \land m_x \leq x\}$$

where the loop invariant $p_1$ is

$$p_1 \equiv \{ m_x = \text{MAX}(S) \land \#T(h_1) \geq 2 \land \text{EVEN}(\#T(h_1))$$

$$\land x = \text{ELEM}B(T(h_1),1) \land S_0 \cap T_0 = \emptyset \land S = f(S_0, T(h_1))$$

$$\land |S| \leq |S_0| \land \forall m[1 \leq m \leq \#T(h_1) \land \text{ODD}(m) \Rightarrow$$

$$\text{ELEM}(T(h_1), m) = \text{MAX}(f(S_0, T(h_1)[1 : m - 1]))]\}$$

It is easy to see that $p_1$ is indeed a loop invariant. The clause $S = f(S_0, T(h_1))$ expresses the fact that during each iteration, the element sent to $P_2$ is removed
from $S$ and the element received from $P_2$ is added to $S$. The clause $|S| \leq |S_0|$ expresses the fact that an element (the maximum of the current $S$) is removed from $S$ and the element received from $P_2$ is added to $S$ during each iteration. (We use $|S| \leq |S_0|$ instead of $|S| = |S_0|$ since the element received from $P_2$ may already be in $S$; thus, during each iteration, the cardinality of $S$ either decreases by 1 or remains unchanged.) The final clause represents that each element sent to $P_2$ (the odd numbered elements of $Tr(h_1)$) is equal to the maximum value in the current $S$.

Next, we have $P_2$:

$$\{h_2 = \varepsilon \land T = T_0 \land S_0 \cap T_0 = \emptyset\}$$

$$\ast\{p_2\}$$

$$[P_1 ? y \rightarrow T := T \cup \{y\};$$

$$\quad m_{n} := \text{MIN}(T);$$

$$\quad P_1 ! m_{n};$$

$$\quad T := T - \{m_{n}\}]$$

$$\{p_2\}$$

where

$$p_2 \equiv \{\text{EVEN}(\#Tr(h_2)) \land T = g(T_0, Tr(h_2)) \land |T| \leq |T_0|$$

$$\land \forall \,[1 \leq m \leq \#Tr(h_2) \land ELEM(h, m)] \Rightarrow$$

$$ELEM(Tr(h_2), m) = \text{MIN}(g(T_0, Tr(h_2)[1 : m - 1]))\}$$

The clause $T = g(T_0, Tr(h_2))$ expresses the fact that during each iteration, the number received from $P_1$ is added to $T$ and the number sent to $P_1$ is removed from $T$. The last clause represents that the number sent to $P_1$ is the minimum of the current $T$. The postcondition is identical to $p_2$; recall that $Tr(h_2)$ has no elements of the kind $(2, T, r)$; thus the element $(2, (1), r)$ added to $h_2$ when the loop in $P_2$ terminates has no effect on $Tr(h_2)$ and hence $p_2$ is a valid postcondition.
Using the rule for parallel composition, we then have

\[ \{ S = S_0 \land T = T_0 \land S_0 \cap T_o = \emptyset \} \]

\[ |P_1||P_2| \]

\[ \{ p_1 \land mx \leq x \land p_2 \land COMPAT'(h_1, h_2) \} \]

It is easy to show that \( COMPAT'(h_1, h_2) \Rightarrow Tr(h_1) = Tr(h_2) \) and this proof can be found in [35].

II.4 Conclusion

We have presented an axiomatic semantics for CSP. An important aspect of our approach is that in proving a property of one of the processes, say \( P_i \), we must consider it in isolation. Therefore, no knowledge of the "expected" behavior of the remaining processes in the program may be used when we are dealing with the proof of \( P_i \). This aspect shows up most clearly in our "input axiom"; if there is an input statement \( P_j?x \) in \( P_i \), the input axiom prevents us from putting any restrictions on the number that \( P_j \) could send in response to this input request. Since we are dealing with \( P_i \) in isolation, nothing in \( P_i \) will give us any indication as to what number \( P_j \) will actually send. The advantage of dealing with the processes in isolation in this fashion is that the property of \( P_i \) thus proved will necessarily be valid irrespective of what the remaining processes in the system may do. As a result, our rule for parallel composition is rather simple.
CHAPTER III

Multiple Exit Statements

In this chapter, we will present a simple modular axiomatic approach that can be used to define the semantics of the multiple-exit statements found in Ada. The key to this approach is in capturing the behavior of a construct S that contains a multiple-exit statement as seen by an external observer of S. These statements include the EXIT and exception handlers. A number of examples are given to illustrate the use of our axioms/rules of inference.

Ada[17] has been defined as a complete language whose multiprogramming or tasking features are intended for implementation either on a single processor or on a set of distributed processors. The United States Department of Defense (DoD) sponsored the development of Ada to provide a standard machine independent high order language for software which is embedded in or procured as part of major defense systems. The designers of Ada acknowledge the strong influence of DP on the creation of its tasking features. The Ada language supports modern programming practices and integrates features from the experimental languages Alphard and CLU along with those which have evolved from ALGOL, PASCAL, and PL/1. Early termination of loop constructs occur with the use of the EXIT statement. Definition and control of concurrent processes is provided through the use of tasks. Tasks may be defined and synchronized using a rendezvous model. Exceptions are raised when constraint violations occur. When an exception is raised, the execution of the corresponding handler replaces the execution of the block where the exception
III.1 EXIT Statement

An exit statement is used to complete the execution of an enclosing loop statement.

```
exit_statement ::= 
Exit [loop._name] [When condition];
```

Execution of an *EXIT* statement begins with the evaluation of the condition, if present. The named loop is exited if the condition evaluated to true or if there is no condition. It is possible to exit several nested loops by an *EXIT* statement that names the outer loop. Ada requires that all loop names be unique.

III.2 Axiomatic Notation

Our proof rules incorporate the notion that multiple-exit statements should be described with multiple postconditions. In order to represent multiple-exit statements, we modified the standard operational model of Apt et al.[3]. This new model will then be used to prove the soundness and relative completeness of the proof rules.

Using the *EXIT* statement destroys the convention that each statement has only one exit. Therefore, we must consider two possible exits. This requires us to use a notation that includes two postconditions - one for the normal exit, called the postcondition; and one for all exits via an *EXIT Loop._name* statement that transfer control outside of the loop labelled Loop._name, called the exit-condition. The notation \{(p)S(q) \{L : r\}\} represents the following:
Given the precondition \( p \), on normal exit from \( S \) the postcondition \( q \) will be satisfied; while on any exit from \( S \) via an EXIT \( L \) statement, the exit-condition \( r \) will be satisfied.

The assertion \( r \) is then used to adjust the postcondition of the named loop \( L \) since termination of this loop can also occur from the execution of the statement EXIT \( L \).

In general, there may be more than one label-assertion pair associated with the statement \( S \). The notation \( \Sigma \) will be used to represent a set of label-assertion pairs and \( \text{LABELS}(\Sigma) \) will represent the set of labels in \( \Sigma \). Thus, \( L \in \text{LABELS}(\Sigma) \) if for some \( p \), \( (L:p) \in \Sigma \). Therefore, \( (p) S (q) \Sigma \) will represent that on any exit from \( S \) via an EXIT \( L \) statement with \( L \in \text{LABELS}(\Sigma) \), the exit-condition \( r \) will be satisfied given that \( (L:r) \in \Sigma \).

\section*{III.3 Axioms and Rules}

\subsection*{III.3.1 Unconditional EXIT}

\[
\{p\} \text{Exit } L \{\text{false}\} \{L:p\}
\]

The normal postcondition reflects that normal exit from the unconditional EXIT is impossible. The exit-condition records that \( p \) will hold on the exit from the named loop \( L \).

\subsection*{III.3.2 Conditional EXIT}

\[
\{p\} \text{Exit } L \text{ When } b \{p \land \neg b\} \{L:p \land b\}
\]

In this case, normal exit from the conditional EXIT can only occur when the
condition b is false. Also, the exit-condition records that \( p \land b \) holds on the exit from the named loop L.

\[ \text{SKIP} \]

\[
\{p\} \text{Skip} \{p\} \Phi
\]

Assignment

\[
\{p_x\} x := e \{p\} \Phi
\]

Sequential Composition

\[
\{p\} S_1 \{q\} \Sigma_1, \{q\} S_2 \{r\} \Sigma_2 \\
\hline
\{p\} S_1;S_2 \{r\} \Sigma_1 \cup \Sigma_2
\]

where \( \cup \) is defined by:

\[
\Sigma_1 \cup \Sigma_2 \overset{\text{def}}{=} \{ (L:p) \mid \forall (L:p) \in \Sigma_1, L \notin \text{LABELS}(\Sigma_1) \} \\
\vee \forall (L:p) \in \Sigma_2, L \notin \text{LABELS}(\Sigma_1) \} \\
\vee [\forall L \in [(\text{LABELS}(\Sigma_1) \cap \text{LABELS}(\Sigma_2))] \\
\wedge [\text{ASSERT}(\Sigma_1,L) \vee \text{ASSERT}(\Sigma_2,L) \equiv p]]
\]

The postcondition of \( S_1;S_2 \) is the usual one. The exit-condition, \( \Sigma_1 \cup \Sigma_2 \), represents that an exit may occur from a loop labeled L either by executing an \textit{EXIT} L statement in \( S_1 \) or by executing an \textit{EXIT} L statement in \( S_2 \). The last
clause in the definition of $\Sigma_1 \cup \Sigma_2$ ensures that any label $L$ occurs at most once in the set of label-assertion pairs. Notice that the function $ASSERT(\Sigma, L)$ returns the assertion paired with the label $L$ that is contained in $\Sigma$.

III.3.3 Alternation

\[
\{ p \land b \} S_1 \{ q \} \Sigma_1, \{ p \land \neg b \} S_2 \{ q \} \Sigma_2
\]

\[
\{ p \}\text{ If } b \text{ Then } S_1 \text{ Else } S_2 \{ q \} \Sigma_1 \cup \Sigma_2
\]

The justification for this rule is similar to that for sequential composition.

Rule of Consequence

\[
p \Rightarrow p', q' \Rightarrow q, \{ p' \} S \{ q' \} \Sigma_1, \Sigma_1 \leftarrow \Sigma_2
\]

\[
\{ p \} S \{ q \} \Sigma_2
\]

\[
\Sigma_1 \leftarrow \Sigma_2 \overset{def}{=} \{ \forall (L : p) \in \Sigma_1. \exists (L : p') \in \Sigma_2. p \Rightarrow p' \} \land PROPER(\Sigma_2)
\]

The rule of consequence allows us to strengthen the precondition and weaken the postcondition (as usual). It also allows us to weaken the exit-condition by weakening the assertion associated with the label $L$, and by adding additional label-assertion pairs. The predicate $PROPER(\Sigma_2)$ ensures that every label $L$ in the set $\Sigma_2$ can only occur once and can be formalized as follows:

\[
\forall L. \forall p, q \ [(L : p) \in \Sigma \land (L : q) \in \Sigma | \Rightarrow p \equiv q]
\]
III.3.4 LOOP

\[
\{p\} S' \{p\} \Sigma, (L:r) \in \Sigma \\
\overset{(p) L:\text{Loop } S' \text{ end}L \{r\} \Sigma - (L:r)}{}
\]

With the \textit{LOOP} statement, normal exit is impossible. Therefore, the \textit{LOOP} can only terminate by means of an \textit{EXIT} statement. Hence, the postcondition of the loop is the exit-condition associated with the \textit{EXIT} \textit{L} statement contained within the body of the loop \textit{S'}. Notice that we represent the situation where \textit{S'} does not contain an \textit{EXIT} \textit{L} statement, using the rule of consequence, by the label-assertion pair \((L:\text{false})\). The assertion \(r\) becomes the postcondition of the loop \textit{L} and therefore must be eliminated from the exit-condition of the loop.

III.3.5 WHILE

\[
\{p \land b\} S' \{p\} \Sigma, (L:r) \in \Sigma \\
\overset{(p) L:\text{While } b \text{ Do } S' \text{ end}L \{(p \land -b) \lor r\} \Sigma - (L:r)}{}
\]

Termination of the \textit{WHILE} can occur from the normal exit of the loop \((p \land -b)\) or the execution of an \textit{EXIT} \textit{L} statement that is contained within its body \((r)\). Therefore, the postcondition of the \textit{WHILE} loop includes the assertions associated with both of these possibilities. Again, we represent the situation where \textit{S'} does not contain an \textit{EXIT} \textit{L} statement by the label-assertion pair \((L:\text{false})\). We then eliminate the corresponding label-assertion pair from the loop exit-condition.
III.4 Lookup Example

We compare our proof system with that of Arbib and Alagic[5] by examining a version of the function Lookup. We modified the program of [5] in order to replace their GOTO with an EXIT statement. Lookup performs a binary search of a sorted array A to discover that element in the array whose value is equal to a given number X. It is assumed that X satisfies the relation A[1] ≤ X < A[N]. The function returns either the index of the array element that equals X or an error flag (-1) which indicates that X is not an element of A.

Function Lookup(var A: array[1..N] of integer; X:integer)
var i,m,n:1..N;
Begin
  m:=1; n:=N; Lookup:=-1;
{((m<n) ∧ sorted(A) ∧ (A[m] ≤ X < A[n]) ∧ (Lookup=-1))}
L:While m+1 < n Do
Begin
  i:=(m+n) div 2;
  If X < A[i] Then n:=i
  Else If A[i] < X Then m:=i
  Else Begin
    m:=i
    {A[m]=X}
    Exit L {false} {L: A[m]=X}
  end
{((m+1≤n) ∧ sorted(A) ∧ (A[m] ≤ X < A[n]) ∧ (Lookup=-1))}{L: A[m]=X}
end L
Notice that the WHILE loop could have terminated from locating the element X in the array A (A[m]=X) or from examining all the elements in the array (m+1=n). It is the following IF statement that determines which of these two situations occurred.

III.5 Exception Handler Statements

During program execution, errors or other exceptional situations can arise. To raise an exception is to abandon normal program execution so as to draw attention to the fact that the corresponding situation has arisen. An exception can be raised by the raise statement. Executing some actions, in response to the arising of an exception, is called handling the exception.

An exception declaration declares a name for an exception.

```
exception.declaration ::= identifier.list : exception;
```

An exception handler occurs in a construct that is either a block statement, the body of a subprogram, or a task unit.
Begin

sequence.of.statements

Exception

test.exception.handler

{test.exception.handler}

diagnostics

test.exception.handler ::= 

When test.exception.choice

{| test.exception.choice} =>

sequence.of.statements

test.exception.choice ::= test.exception.name

| others

The exceptions denoted by the exception names given as exception choices of a
blocked section of code must all be distinct. The exception choice OTHERS is only
allowed for the last exception handler of a blocked section of code. It stands for all
exceptions not listed in previous handlers of the blocked section of code including
exceptions whose names are not visible at the place of the exception handler.

A raise statement raises an exception.

raise.statement ::= Raise [test.exception.name];

If an exception is raised in the sequence of statements of a blocked section of
code, then execution of this sequence of statements is abandoned. If this blocked
section of code does contain a handler for the raised exception, then the execu-
tion of the sequence of statements of the handler completes the execution of the
blocked section of code. Conversely, the next action depends on the context of the surrounding code:

- For a subprogram body, the same exception is raised again at the point of call of the subprogram, unless the subprogram is the main program itself, in which case the execution of the main program is abandoned.

- For a block statement, the same exception is raised again immediately after the block statement.

- For a task body, the task becomes completed.

An exception that is raised again is said to be propagated. Notice that propagation does not occur in the case of an exception raised within a task body. If an exception is raised in the sequence of statements of an exception handler, the exception will be propagated to the next highest level depending on the nature of the blocked section of code.

III.6 Exception Handler Rules

Given the complete discussion of the *EXIT* L statement and its related statements, we can extend the concept of multiple postconditions to the *RAISE* statement and exception handlers. In this case, label-assertion pairs will be referred to as exception-assertion pairs. Therefore, we will associate with each exception an assertion which captures the state of the system at the time that the given exception was raised. When using the previous axioms in the context of exception handling, we will refer to the label as E instead of the previous notation L.
III.6.1 RAISE Statement

\{p\} Raise E \{false\} \{E : p\}

The normal postcondition reflects that the normal exit from the RAISE statement is impossible. The exit-condition records that \(p\) will hold on the jump that occurs from the execution of the RAISE statement.

III.6.2 Exception Handler

\{\textit{ASSERT}(\Sigma, E_i)\} S_i, \{q\} \Sigma', i = 1, \ldots, n

\{\Sigma\} \textit{H}: Exception When \(E_1 \Rightarrow S_1 \ldots\)

When \(E_n \Rightarrow S_n\) end \{q\} \Sigma' \uplus REST(\Sigma, EXCPT(H))

The set of exception-assertion pairs is used as the precondition to the handler since each assertion associated with each exception named in a clause of the handler \(H\) will be used as the precondition to that clause. \textit{ASSERT}(\Sigma, E_i)\) returns the assertion paired with the exception \(E_i\) in the set \(\Sigma\). A new set of exception-assertion pairs, \(\Sigma'\), will be generated from any exceptions raised during the execution of the exception handler \(H\). The set of exception-assertion pairs defined by \(REST(\Sigma, EXCPT(H))\) will contain all exception-assertion pairs from \(\Sigma\) for which there did not exist a clause within the exception handler \(H\) for their exceptions. These assertions will be used to modify the postcondition of the surrounding construct.
Exception Handler With OTHERS

\[
\{ \text{ASSERT}(\Sigma, E_i) \} S_i \{q\} \Sigma', \ i = 1, \ldots, n-1
\]
\[
\{ \bigvee_j \text{ASSERT}(\Sigma, \text{REST}(\Sigma, \text{EXCEPT}(H))) \} S_n \{q\} \Sigma'
\]

(\Sigma) H: Exception When \( E_1 \Rightarrow S_1 \) … When \( E_{n-1} \Rightarrow S_{n-1} \)

When others \( \Rightarrow S_n \) end \( \{q\} \Sigma' \)

Since all exceptions contained within \( \Sigma \) will correspond to a clause within the handler \( H \), the precondition of the OTHERS clause will be the disjunction of the assertions associated with the exceptions contained in \( \Sigma \) that are not explicitly named within \( H \). Notice that the first line deals with all the explicitly named exceptions in the handler \( H \). Again, \( \Sigma' \) will contain all exceptions raised during the execution of \( H \) with their corresponding assertions.

III.6.3 Block Statement

\[
\{p\} \begin{array}{c} S \{q\} \Sigma \\ \{p\} \text{Begin } S \text{ end } \{q\} \Sigma \end{array}
\]

If the block statement does not contain an exception handler, then any exceptions that were raised during its execution will be propagated to the surrounding construct. Therefore, the set \( \Sigma \) will also be propagated. Theoretically, the postcondition of the block statement is written as \( \{q \lor \text{false} \} \) since the entire block statement may not be executed due to the raising of an exception.
Block Statement With Exception Handler

\[
\{p\} S \{q\} \Sigma, \{\Sigma\} H (r) \Sigma'
\]

\[
\{p\} \text{Begin } S; H \text{ end } \{q \lor r\} \Sigma'
\]

Termination of a block statement that contains an exception handler can occur in three different situations. If the execution completes normally, then the postcondition of the block statement is \(q\). If an exception is raised during its execution and a corresponding clause is executed in the exception handler \(H\), then the postcondition of the block statement is \(r\). Lastly, if an exception is raised and no handler exists for the exception then the postcondition is false. In this case, the assertion associated with the exception is contained in \(\Sigma'\) which is propagated outside of the block statement. Notice that \(\Sigma'\) will also contain any exceptions that were raised during the execution of the exception handler \(H\).

III.6.4 Procedure Call

\[
\{p \overline{X}, \overline{Y}, \overline{Z}\} \text{Proc } P_i(\overline{U} : \text{in}; \overline{V} : \text{inout}; \overline{W} : \text{out}) \text{ begin } S \text{ end } \{q \overline{V}, \overline{W}\} \Sigma \overline{V}, \overline{W}
\]

\[
\{p\} \text{Call } P_i(\overline{X} : \text{in}; \overline{V} : \text{inout}; \overline{Z} : \text{out}) \{q\} \Sigma
\]

where \(S\) is the body of the procedure \(P_i\).

If an exception is raised during the execution of the procedure body and that procedure does not contain a handler for the exception, then that exception is propagated to the construct that contain the procedure call. Therefore, the \(\Sigma\) associated with the procedure must also be associated with the procedure call. Since the postcondition of the procedure and \(\Sigma\) contains references to the parameters of the procedure, they must be replaced by their corresponding arguments given in the \textit{CALL}. Notice that neither \(q\) nor \(\Sigma\) may refer to the \textit{in} parameters.
III.6.5 Procedure

\{p\} \text{Begin} \ S \ \text{end} \ \{q\} \ \Sigma

\{p\} \ \text{Proc} \ P_4(U: \text{in}; \ Y: \text{inout}; \ Z: \text{out}) \ \text{Begin} \ S \ \text{end} \ \{q\} \ \Sigma

This rule simply states that the block statement can be a procedure body and therefore can be extended to a procedure.

III.7 Search Example

The following annotated example is given in Luckham and Polak[30]. It considers the procedure \textit{Search} with \textit{in} parameters \(A, N,\) and \(X\) and \textit{out} parameter \(I.\)

In the case when the value \(X\) is not found within the array \(A,\) \textit{Search} will raise the exception \textit{Notfound}.

\textbf{Type} \textit{array} is array(1...N) of Integer;


\{true\}

Begin

...........

\{\forall j. \ (1 \leq j \leq N \Rightarrow X \neq A[j])\}

\textbf{Raise} notfound \{false\} \ \{\text{notfound: } \forall j. \ (1 \leq j \leq N \Rightarrow X \neq A[j])\}

...........

end

\{1 \leq I \leq N \land X = A[I]\} \ \{\text{notfound: } \forall j. \ (1 \leq j \leq N \Rightarrow X \neq A[j])\}

The procedure \textit{Search} could be used in the following context with \(M, Y,\) and \(B\) declared in some global scope.
Begin

Search(B,M,Y,K);
\{1 \leq K \leq M \land Y=B[K]\} \{\text{notfound} : \forall j. (1 \leq j \leq M \Rightarrow Y\neq B[j])\}

Exception
\{\forall j. (1 \leq j \leq M \Rightarrow Y\neq B[j])\}

When notfound ⇒

k:=0;
\{(\forall j. (1 \leq j \leq M \Rightarrow Y\neq B[j]) \land k=0)\}

end

end

\{(Y=B[K]) \lor (\forall j. (1 \leq j \leq M \Rightarrow Y\neq B[j] \land k=0)\}

III.8 Summary

We have presented a relatively simple set of proof rules that capture the semantics of the \textit{EXIT} statement. We modified the standard Hoare-logic to include an exit-condition which consisted of a set of label-assertion pairs that represent what assertion should hold when a jump to the corresponding label occurs. This new notation did not significantly add to the complexity of the other statements within the language. From the \textit{Lookup} example, one can see how these label-assertion pairs are used to modify the normal postconditions of the looping constructs and that these proof rules will result in fairly simple proofs.

The proof rules for the \textit{EXIT} are simpler than the rules for the \textit{GOTO} \cite{5,15,40} and, surprisingly, they do not add additional complexity to the other proof rules of the proof system. Conversely, the rule for the \textit{GOTO} is actually a collection of general rules that depend upon the context in which the \textit{GOTO} statement appears.
and therefore their proof system is rather complex. Using the system of Arbib
and Alagic[5], the proof of the Lookup example is constructed using a set of general
proof rules that can be combined to prove an equivalent set of statements. Because
of this fact, we feel that our specific set of proof rules allow one to verify the
given problem directly. The work of de Bruijn[13] on the GOTO statement is quite
analogous to the system presented in sections 2 through 10. His notation records
labels with their corresponding label-assertions (the assertion that should hold at
that label). Since the EXIT is a restricted form of the GOTO, we were able to
directly modify the postconditions of the looping constructs.

Luckham and Polak[30] require documentation for exceptions that are propa­
gated by procedures or functions. Using this documentation, it is then possible to
modify the procedure call rule to describe the dynamic association of handlers with
exceptions at call time. They introduce an exception propagation declaration and
require that each exception propagated by a subprogram be specified in the sub­
program's propagation declaration. A propagation declaration has to provide an
assertion that is to be true whenever one of the exceptions in the propagate clause
is propagated. However, their system does not handle exceptions raised during a
rendezvous and in the next chapter, we will illustrate how our notation captures
these exceptions. Also, documentation is used to record which assertions should
hold when the corresponding exception is raised, whereas our notation generates
and maintains these assertions using the proof rules.

Yemini[43] proposes a “replacement model” which supports all the handler re­
sponses, including resumptions. Her notation includes an input assertion, an output
assertion, an exception condition which corresponds to a particular exception, and
a resumption condition which is required to be satisfied by a handler before resump­
tion. She then uses procedure proof rules to push assertions through a signalling in
the process of verification, even though the handler is not known within the signaller
body. Our notation can easily be extended to handle resumption by generating another set of label-assertion pairs that contain the exceptions paired with the output assertion of the corresponding exception handler clause.
CHAPTER IV

Tasks

We begin the major portion of this thesis by considering Ada tasks and presenting an appropriate set of axioms and rules of inference corresponding to the various aspects of Ada tasks. The fundamental idea underlying our approach to capture the semantics of Ada tasks is that each task must be considered in isolation from the other tasks. Therefore, we will present a means of capturing the externally visible behavior of each task and demonstrate how these behaviors can be combined to determine the behavior of the entire program. An example is presented to illustrate how one could use our axioms/rules of inference to determine the behavior of a bounded buffer.

IV.1 Ada

Tasks are entities whose executions proceed in parallel in that each task can be considered to be executed by a logical processor of its own. An entry of a task can be called by other tasks. A task accepts a call of one of its entries by executing an accept statement for the entry. Synchronization is achieved by a rendezvous between a task issuing an entry call and a task accepting the call.

The properties of each task are defined by a corresponding task unit which consists of a task specification and a task body.
task_specification ::= 
    Task [type] identifier [is 
        {entry.declaration} 
        {representation.clause} 
    end [task.name]]

task.body ::= 
    Task body task.name is 
        [declarative.part] 
        Begin 
            sequence.of.statements 
            [Exception 
                exception.handler 
                {exception.handler}] 
        end [task.name];

The task automatically becomes active when the task elaborating its declaration reaches the following BEGIN. Entry calls and ACCEPT statements are the primary means of synchronization of tasks and communicating values between tasks. An entry declaration is similar to a subprogram declaration and is only allowed in a task specification. The parameters may be of three modes: in, out, and inout which are input, output, and a combination of the two types respectively. The actions to be performed when an entry is called are specified by corresponding accept statements. A task may contain more than one ACCEPT statement for the same entry.
entry_call_statement ::= 
    entry_name [parameter_part]

accept_statement ::= 
    Accept entry_name [parameter_part] | do
    sequence_of_statements
    end [entry_name]

If a given entry is called by only one task, there are two possibilities:

* If the calling task issues an entry call statement before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is suspended.

* If a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call is received.

When an entry has been called and a corresponding accept statement has been reached, a rendezvous occurs and the sequence of statements associated with the accept statement is executed by the called task while the calling task remains suspended. Thereafter, the calling task and the task owning the entry continue their execution in parallel. If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued where one queue is associated with each entry.

A task body can also contain a select statement. There exist three forms of the select statement. A selective wait statement allows a combination of waiting for, and selecting from, one or more alternatives.
selective.wait ::= 
   Select 
   select.alternative 
   {Or 
    select.alternative} 
   [Else 
    sequence.of.statements] 
   endselect;

select.alternative ::= 
   [When condition \(\Rightarrow\)] 
   accept.statement [sequence.of.statements] 
   | Delay simple.expression [sequence.of.statements] 
   | Terminate

A selective wait must contain at least one accept alternative. In addition, a selective wait can contain a terminate alternative, an else part, or one or more delay alternatives but these three possibilities are mutually exclusive. A select alternative is said to be open if it does not start with a when or if the condition of the when is true.

For the execution of a selective wait, open alternatives are determined. Selection of one such alternative takes place immediately if a corresponding rendezvous is possible. If several alternatives can thus be selected, one of them will be selected arbitrarily. When such an alternative is selected, the corresponding ACCEPT statement and possible subsequent statements are executed. If no rendezvous is immediately possible and there is no else part, the task waits until an open selective wait alternative can be selected.
Selection of the other forms of alternative or of an else part is performed as follows:

* An open delay alternative will be selected if no accept alternative can be selected before the specified delay has elapsed.
* The else part is selected if no accept alternative can be immediately selected.
* An open terminate alternative cannot be selected while there is a queued entry call for any entry of the task.

Execution of the \textit{DELAY} statement evaluates the simple expression and suspends further execution of the task containing the \textit{DELAY} statement for at least the duration specified by the resulting value. The execution of the \textit{TERMINATE} statement causes the completion of the execution of the task which contains it.

A conditional entry call issues an entry call that is then canceled if a rendezvous is not immediately possible.

\begin{verbatim}
conditional.entry.call ::= 
    Select 
    entry.call.statement 
    [sequence.of.statements] 
    Else 
    sequence.of.statements 
    end select;
\end{verbatim}

The entry call is canceled if either the execution of the called task has not reached a point where it is ready to accept the call or if there are prior queued entry calls for this entry. If the called task has reached a \textit{SELECT} statement, the entry call is canceled if an accept alternative for this entry is not accepted. If the
entry call is canceled, the statements of the else part are executed.

A timed entry call issues an entry call that is canceled if a rendezvous is not started within the given delay.

    timed_entry_call ::= Select
                        entry_call_statement [sequence.of.statements]
                        Or delay_statement [sequence.of.statements]
                    end select;

If a rendezvous can be started within the specified duration, it is performed and the optional sequence of statements after the entry call is then executed. Otherwise, the entry call is canceled when the specified duration has expired and the optional sequence of statements of the delay alternative is executed.

A rendezvous can be completed abnormally when an exception is raised within an ACCEPT statement whose body does not contain a handler for that exception. In this case, the execution of the ACCEPT statement is abandoned and the same exception is raised again immediately after the ACCEPT statement within the called task. Also, the exception is propagated to the calling task at the point of the entry call.

If a task has no dependent task, its termination takes place when it has completed its execution. If a task has dependent tasks, its termination takes place when the execution of the task is completed and all dependent tasks are terminated. A block statement or subprogram body whose execution is completed is not left until
all of its dependent tasks are terminated.

Termination of a task otherwise takes place if and only if its execution has reached an open \textit{TERMINATE} alternative in a select statement and the following conditions are satisfied:

* The task depends on some master whose execution is completed.

* Each task that depends on the master considered is either already terminated or similarly waiting on an open \textit{TERMINATE} alternative of a select statement.

When both conditions are satisfied, the task considered becomes terminated, together with all tasks that depend on the master considered. The rules given for termination imply that all tasks that depend on a given master and that are not already terminated, can be collectively terminated if and only if each of them is waiting on an open terminate alternative of a select statement and the execution of the given master is completed.

A rendezvous may be modeled by the \textit{FORK}, \textit{JOIN}, and \textit{RESUME} commands between calling and called tasks, and an operating system\cite{41}. The calling of a task entry as well as the arrival at an \textit{ACCEPT} statement may be modelled by \textit{RESUME} commands to the operating system. Rendezvous may be viewed as a concurrent form of coroutine control that requires the operating system to receive \textit{resume} calls from both a calling and a called task, results in temporary merging (\textit{join}) of the calling and called tasks in order to execute the body of the \textit{ACCEPT} statement, and requires a \textit{FORK} with a \textit{RESUME} call in each branch to resume separate concurrent execution of the calling and the called task when the rendezvous has been completed. Lastly, the entry call interface of Ada tasks cannot support coroutine-like suspension and resumption of subprocesses within a task.
IV.2 Axiomatic Notation

Our proof rules are centered around the interactions between tasks. These interactions can be used to characterize the externally visible behavior of any task. The semantics of a task is a simple formalization of what an external observer of that task would see during its execution. Our assertions will refer to this sequence of interaction of the task that we are verifying. To easily express this sequence of interactions, we will associate a sequence $h_i$ with each task $T_i$.

The use of the $\text{EXIT}$, $\text{RAISE}$, and $\text{TERMINATE}$ statements require us to use the multiple exit notation $\{p\} S \{q\} \{E:r\}$ which represents the following:

Given the precondition $p$, on normal exit form $S$ the postcondition $q$ will be satisfied; while on any jump from $S$ via an $\text{EXIT}$ or $\text{RAISE}$ statement, the exit-condition $r$ will be satisfied.

The assertion $r$ will then be used to adjust the postcondition of the task body. In general, there may be more than one exception-assertion pair associated with the statements; thus $\Sigma$ will be used to represent a set of exception-assertion pairs. Notice, Ada requires that all loop names and exception names be unique. Moreover, we may also have the pair $(\text{Terminate}:r)$ in $\Sigma$ corresponding to the termination of the task by the execution of a $\text{TERMINATE}$ statement. Often we want to refer to the exceptions in a set of exception-assertion pairs. Therefore, $\text{EXCEPTION}(\Sigma)$ will denote the set of exceptions in $\Sigma$.

The elements that comprise the interaction sequences will be of varying type and length. The first component will specify the type of the element (e.g. call, return, etc.). If the element is associated with an entry call then the next three components record the identity of the task in which this statement appears and the identity of the corresponding $\text{ACCEPT}$ statement in the called task (e.g. $i,j,k$). Therefore, if the task $T_i$ issues an entry call to the $\text{ACCEPT}$ statement $A_k$ located in the task $T_j$, the corresponding element is written as $(\text{call to},i,j,k,...)$. If the
element is associated with an ACCEPT statement then the next two components record the identity of the given ACCEPT within the task. Therefore, if the task $T_j$ accepts an entry call to the ACCEPT statement $A_k$ then the corresponding element is written as $(\text{call}, i, j, k, \ldots)$. The last two components record the parameter values (e.g., $X, Y, Z$). Completly the example, the element $(\text{call}.\text{to}, i, j, k, X)$ if the task $T_i$ issues an entry call to $T_j.A_k$ with $X$ being the arguments of the call.

We will consider the axioms and rules corresponding to the statements in the task $T_i$. A majority of the axioms/rules of inference pertaining to tasks were presented in the previous chapter.

IV.3 TERMINATE Statement

$$\{p\} \text{Terminate} \{\text{false}\} \{\text{Terminate} : p\}$$

Normal exit from the TERMINATE statement is also impossible. We use the special exception name Terminate to label the assertion associated with this statement.

IV.4 Entry Call Statement

$$p \Rightarrow \{\forall \bar{U}, \bar{V}, q^\bar{V}, \bar{h}, (\text{call}.\text{to}, i, j, k, X, Y)^\sim (\text{return}.\text{from}, i, j, k, \bar{U}, \bar{V})\}$$

$$p \Rightarrow \{\forall \bar{U}, \bar{V}, r_l^\bar{V}, \bar{h}, (\text{call}.\text{to}, i, j, k, X, Y)^\sim (\text{exception}, i, j, k, e, \bar{U}, \bar{V})\}, l = 1, \ldots, m$$

$$\{p\} T_j.A_k (\bar{X} : \text{in}; \bar{V} : \text{inout}; \bar{Z} : \text{out}) \{q\} \text{EXCEPT}(T_j.A_k)$$

With the exception of task synchronization, an entry call is similar to an ordinary procedure call. The effect of the statement $T_j.A_k$ within the task $T_i$ is to
extend the sequence $h_i$ by two elements: one corresponding to the call and the other to the return. The element that corresponds to the return can either represent a normal return or the abnormal return that occurs from an unhandled exception.

The concatenation operation, $\cdot$, represents the addition of an element to the right end of a sequence. In the first line, an element is added to the sequence to represent that control leaves $T_i$ to call the $ACCEPT$ statement $T_j.A_k$ and that the values of the variables $X$ and $Y$ are the arguments associated with this call. The second element is added to the sequence to indicate that when control returns to $T_i$, the parameter values that are returned by the $ACCEPT$ statement are assigned to the corresponding argument variables $Y$ and $Z$. The universal quantifier ensures that we consider all possible values that $T_j.A_k$ could return. In the case of the abnormal return, the second line ensures that an element of type exception is added to the sequence for each exception $e_j$ in the set $EXCEPT(T_j.A_k)$. $EXCEPT$ is a syntactic function that generates a set of exception-assertion pairs that contains all exceptions that could be raised during the execution of the $ACCEPT$ statement $T_j.A_k$. Each exception $e_j$ will be paired with the corresponding assertion $r_j$.

IV.5 $ACCEPT$

$$^h_i (call,i,k,X,Y) \Rightarrow p$$

$$(p) \{q'\} \Sigma_1$$

$$q' \Rightarrow q^h_i (return,i,k,Y,Z)$$

$$\Sigma_1 \Rightarrow \Sigma_2 h^j_i (except,i,k,e_j,Y,Z), j = 1, \ldots, m$$

$$(r) Accept \ T_i.A_k \ (X: in; \ Y: inout; \ Z: out) \ do \ S \ end \ \{q\} \Sigma_2$$
The first line ensures that when the execution of $T_i.A_k$ begins, the appropriate element corresponding to the entry call has been recorded on $h_i$. All possible values of $X$ and $Y$, which are the in and inout parameters respectively, are allowed since their actual values are unknown. The third line ensures that the element recording the return from $T_i.A_k$ to the caller is concatenated to the sequence. The element includes the values of the inout and out parameters, $Y$ and $Z$ respectively. The fourth line examines the case of an exception raised within the body of an ACCEPT statement that does not contain a handler for that exception. It ensures that the assertions contained within the exception-assertion pairs associated with the body of the ACCEPT reflect the possible abnormal termination of the ACCEPT due to the raising of the corresponding exception. Notice that $m$ is the cardinality of the set $EXCEPT(T_i.A_k)$.

IV.6 Selective Wait

IV.6.1 Selective Wait Without ELSE

$$\{p \land b_j\} S_j \{q\} \Sigma, \ j = 1, \ldots, n$$

$(p)$ Select When $b_1 \Rightarrow S_1$ Or $\ldots$ Or When $b_n \Rightarrow S_n$ end $(q)$ $\Sigma$

IV.6.2 Selective Wait With ELSE

$$\{p \land b_j\} S_j \{q\} \Sigma, \ j = 1, \ldots, n - 1$$

$$(p) S_n \{q\} \Sigma$$

$(p)$ Select When $b_1 \Rightarrow S_1$ Or $\ldots$ When $b_{n-1} \Rightarrow S_{n-1}$ Else $S_n$ end $(q)$ $\Sigma$
If the Selective Wait contains an *ELSE* clause then no other branches may contain a *DELAY* or a *TERMINATE* statement. If no rendezvous is immediately possible, then the *ELSE* clause may be selected. This situation occurs when either all the branches are closed or no rendezvous is immediately possible for the open branches. Therefore, the *ELSE* clause can be selected when any or all of the branches are open or closed.

**IV.6.3 Conditional Entry Call**

\[
\{p\} S_1 (q) \Sigma, \{p\} S_2 (q) \Sigma \\
\{p\} \text{Select } S_1 \text{ Else } S_2 \text{ end } (q) \Sigma
\]

In the case of the Conditional Entry Call, if a rendezvous is immediately possible then the first clause is selected; otherwise the *ELSE* clause is selected. Since the selection is dependent upon the order of execution of the tasks and no assumptions are permitted as to the behaviors of the other tasks then no indication is given as to which clause will be selected.

**IV.6.4 Timed Entry Call**

\[
\{p\} S_1 (q) \Sigma, \{p\} S_2 (q) \Sigma \\
\{p\} \text{Select } S_1 \text{ Or Delay } D; S_2 \text{ end } (q) \Sigma
\]

If a rendezvous is possible before \(D\) seconds have transpired then the first clause is selected; otherwise the second clause is selected. Again, since no assumptions are permitted as to the behaviors of the other tasks then no information is given as to which clause will be executed.
IV.7 Task Without Exception Handler

\[
\{p\} S \{q\} \Sigma
\]

\[
\{p\} \text{Task } T_i \text{ Begin } S \text{ end } \{q \lor \bigvee_j \text{ASSERT}(\Sigma, \text{EXCEPTION}(\Sigma))\}
\]

A task can terminate after the normal completion of its execution (\(q\)). It can also terminate after the execution of a \textit{TERMINATE} statement or the raising of an exception. Therefore, the postcondition of a task without an exception handler will reflect these possibilities by the addition of the disjunction of all the assertions contained within the exception-assertion set \(\Sigma\). Form the previous chapter it is known that \textit{ASSERT}(\Sigma, \text{EXCEPTION}(\Sigma)) is the set of assertions that correspond to the exceptions given by \textit{EXCEPTION}(\Sigma).

IV.8 Task With Exception Handler

\[
\{p\} S \{q\} \Sigma_1, \{\Sigma_1\} H \{q'\} \Sigma_2, (\text{Terminate: } r) \in \Sigma_1
\]

\[
\{p\} \text{Task } T_i \text{ Begin } S; \text{H end } \{q \lor q' \lor r \lor \bigvee_m \text{ASSERT}(\Sigma_2, \text{EXCEPTION}(\Sigma_1))\}
\]

In this case, termination can occur from normal completion of the execution of the task (\(q\)), execution of a \textit{TERMINATE} statement within the body of the task (\(r\)), execution of the exception handler (\(q'\)), or a raised exception that does not have a corresponding clause in the exception handler. Therefore, the last disjunction of the postcondition will contain the assertions that correspond to all exceptions raised within the task body that do not correspond to a clause in the handler
along with all exceptions raised and \textit{TERMINATE} statements executed within the exception handler itself. Notice that if $\Sigma_i$ does not contain the exception-assertion pair labeled \textit{Terminate}, the pair (\textit{Terminate}:false) can be added to $\Sigma_i$ using the rule of consequence.

\section*{IV.9 Parallel Composition}

\begin{align*}
\{p_i \land h_i = \epsilon\} &\quad T_i \quad \{q_i\}, \quad i = 1, \ldots, n \\
\{p_1 \land \ldots \land p_n\} &\quad [T_1|| \ldots ||T_n] \quad \{q_1 \land \ldots \land q_n \land \text{COMPAT}(h_1, \ldots, h_n)\}
\end{align*}

Lastly, we will combine the individual task postconditions and add a compatibility requirement, using the predicate \textit{COMPAT}, on the task interaction sequences. The compatibility check ensures that any action in which two tasks participate is recorded on the two task sequences in a mutually consistent fashion.

\section*{IV.10 Buffer Example}

The following example of a Bounded Buffer was originally given in Barringer and Mearns\cite{Barringer}. This version of the standard bounded buffer problem smooths out the variations between the speed of output of a producing task and the speed of input of some consuming task. The \textit{PRODUCER} and \textit{CONSUMER} tasks copy values from an array $A(0..M)$ to an array $D(0..M)$. The buffer \textit{POOL} is an array with bounds $0$ up to $N-1$ where $1 \leq N \leq M$. The three tasks, \textit{PRODUCER}, \textit{CONSUMER}, and \textit{BUFFER} are suitably annotated by their respective local proofs. The following functions will be used to simplify the notation used in the proof.

$\textit{ODD}(h,k)$ – The $k^{th}$ element of the sequence obtained by removing all the even numbered elements from $h$. 
\(EVEN(h, k)\) – The \(k^{th}\) element of the sequence obtained by removing all the odd numbered elements from \(h\).

\(#BUFFER.PUT\) – The number of elements of type "return" from the \(ACCEPT\) statement \(PUT\) in the process sequence \(h_B\).

\(#BUFFER.GET\) – The number of elements of type "return" from the \(ACCEPT\) statement \(GET\) in the process sequence \(h_B\).

\(h_{PUT}\) – The sequence generated from \(h_B\) by concatenating the values of the elements \((\text{call}, B, P, \text{POOL}(\text{inm mod N}))\).

\(h_{GET}\) – The sequence generated from \(h_B\) by concatenating the values of the elements \((\text{return}, B, G, \text{POOL}(\text{outm mod N}))\).

\(h_{ODD}\) – The sequence generated by concatenating the values of the elements of \(ODD(h_P, k)\).

\(h_{EVEN}\) – The sequence generated by concatenating the values of the elements of \(EVEN(h_O, k)\).

**Task Body** \(PRODUCER\) is

\[A: \text{Array}(0..M) \text{ of } X;\]

\[i: \text{integer};\]

\[\text{Begin}\]

\[i := 0;\]

\[\{i \geq 0 \land i \leq M + 1 \land \#h_P = 2 \ast i \land \forall k \lt i \ [ODD(H_P, k) = (\text{call.to}, P, B, P, A(k))]\}\]

\[\text{While } i \leq M \text{ do}\]

\[\text{BUFFER.PUT (A(i):ln);}\]

\[i := i + 1;\]

\[\text{end While}\]

\[\text{end}\]
{i=M+1 ∧ #h_P=2M+2 ∧ ∀k<i [ODD(h_P,k)=(call.to,P,B,P,A(k))]
 ∧ #ODD(h_P,i)=M+1}

**Task Body CONSUMER is**

D: Array(0..M) of X:
j: integer;

Begin
j:=0;
{j≥0 ∧ j≤M+1 ∧ #h_C = 2*j ∧
 ∀k<j [EVEN(h_C,k)=(return.from,C,B,G,D(k))]}]
While j≤ M do
   BUFFER.GET(D(j):out);
   j:=j+1;
end While
end

{j=M+1 ∧ #h_C=2M+2 ∧ ∀k<j [EVEN(h_C,k)=(return.from,C,B,G,D(k))]
 ∧ #EVEN(h_C,j)=M+1}

**Task Body BUFFER is**

POOL: Array(0..N-1) of X;
count, inm, outm: integer;

Begin
count:=0; inm:=0; outm:=0;
Loop
{0≤count≤N ∧ count=#BUFFER.PUT−#BUFFER.GET
 ∧ h_GET ⊆ h_PUT}
Select
When count < N ⇒

Accept \(PUT(c:in X)\) do

\(POOL(\text{inm mod N}) := c;\)

end;

\(\text{inm} := \text{inm} + 1;\)

\(\text{count} := \text{count} + 1;\)

Or When count > 0 ⇒

Accept \(GET(c:out X)\) do

\(c := POOL(\text{outm mod N});\)

end

\(\text{outm} := \text{outm} + 1;\)

\(\text{count} := \text{count} - 1;\)

Or

Terminate

{false} \{(\text{Terminate} : \text{count}=\#BUFFER.PUT-\#BUFFER.GET \land \text{hGET} \subseteq \text{hPUT})\}

end Select;

end Loop; \{false\}

end

{false \lor \{\text{count}=\#BUFFER.PUT-\#BUFFER.GET \land \text{hGET} \subseteq \text{hPUT}\}}

Using the parallel composition rule, one can obtain:

\[\forall k \ [0 \leq k < M + 1] \Rightarrow D(k) = A(k)\]

by demonstrating the correspondence between the array \(A\) and the array \(POOL\) and the correspondence between the array \(POOL\) and the array \(D\). The predicate \(COMPAT\) will pair the elements (call.to,P,B,P,A(k)) in \(h_P\) to the elements
(call,B,P,POOL(k)) in h_{B} and the elements (return,B,G,POOL(k)) in h_{B} to the elements (return,from,C,B,G,D(k)) in h_{C}. Therefore, we can conclude h_{ODD} = h_{PUT} and h_{EVEN} = h_{GET} which allows us to obtain the program behavior.

IV.11 Summary

In this chapter we have presented a set of proof rules for Ada tasks which examines individual tasks in complete isolation by banning any assumptions about the behavior of the other tasks in the program. We feel that this methodology usually results in fairly simple proofs which can be envisioned given the example of the bounded buffer.

This methodology can be contrasted with the proof systems of Barringer & Mearns[8] and Gerth & de Roever[19]. In their systems, the properties of individual tasks are proved using assumptions about the behavior of the remaining tasks within the program. Their parallel composition rule contains a "cooperation test", since this test is needed to substantiate the validity of these assumptions. To express this test, a global invariant is introduced to tie the local reasonings within each process globally together. In particular, the global invariant is used to distinguish among all communication possibilities (the syntactically matching ones), and the ones which may actually occur (the semantically matching ones). Also, auxiliary variables have to be introduced to express the necessary assertions and invariants. As the variables appearing free in the global invariant have to be updated when communication occurs to model synchronization, the global invariant can not hold throughout the whole program. Hence the introduction of bracketed sections (each associated with a unique communication-action) to which the updating of the global invariant-variables is confined and inside of which the global invariant need not hold.

Both Gerth & de Roever[19] and Barringer & Mearns[8] experienced problems with the nesting of communications which caused the nesting of bracketed
sections. Barringer & Mearns required the rewriting of the program to eliminate the nested communication, whereas Gerth & de Roever introduced the concept of "open" bracketed sections so that each communication will be enclosed by a single bracketed section.

The proof system of Gerth & de Roever\[19\] does not include exception handling, whereas the proof system of Barringer & Mearns\[8\] treat a task that has raised an exception as a 'failure'. The \textit{TERMINATE} statement is discussed in both proof systems, but neither presents a means of explicitly maintaining the assertion associated with termination. Also, they require the use of a predicate \textit{ALLEND} which assures that all the other tasks have terminated or are capable of executing a \textit{TERMINATE} statement.
CHAPTER V

Distributed Processes Language

Our approach will now be used to axiomatize a rather different type of concurrency than the one found in Ada. Distributed Processes exhibits a more complex form of concurrency due, mainly, to the coroutine-like structure present in an individual process. Despite this added complexity, we are able to apply our approach to define the axiomatics of Distributed Processes. Corresponding to the two levels of concurrency, we have two rules of composition: the first one allows us to obtain the behavior of a process from the behaviors of its procedures; the second one allows us to obtain the behavior of a program given the behaviors of its processes. Our proof system will be used to verify a parallel sort program taken from Gerth et al.[20], who prove its correctness using their proof system. A comparison of our proof and that in [20] illustrate the basic differences between the two approaches.

V.1 DP

Distributed Processes[11], henceforth called DP, was proposed as a basis for a language for real-time applications controlled by microcomputer networks with distributed storage by Brinch Hansen. A DP program consists of a fixed number of sequential processes that are executed concurrently and exist forever. Each process can only access its own variables - therefore, there are no shared variables. A process may call the common procedures defined within the other processes. Processes are
synchronized by means of nondeterministic statements called guarded regions.

If \( P_1, P_2, \ldots, P_n \) denote \( N \) processes, a program is denoted as:

\[
[ P_1 \| P_2 \| \cdots \| P_n ]
\]

V.1.1 Processes

A process is defined as follows:

- Process name
- Variables local to the process
- Common Procedures
- Variables local to Process Initialization
- Process Initialization

Each process contains a number of variables that are accessible to all the procedures of the process. A procedure in one process may call a procedure in another process, but not a procedure within the same process. Each process also includes a process initialization section and the execution of a process begins with this section.

V.1.2 Procedures

A procedure is defined as follows:

- Proc name(input parameters#output parameters)
- Variables local to procedure
- Procedure body

There are two types of parameters: in and out. A procedure can be called by the process initialization of the process within which the procedure is defined (i.e. internal calls). It can also be called by any other procedures or process initializations
that are defined within any of the other processes (i.e. *external* calls).

A process P can call the procedure R, defined within process Q, by the following statement.

```plaintext
CALL Q.R (expressions#variables)
```

When a procedure call is accepted, a new incarnation of the called procedure is created for the calling process. Previous to the execution of the procedure R, the expression values of the call are assigned to the in parameter. When the execution of the procedure terminates, the out parameters are assigned to the variables listed within the call. An external request is regarded as an atomic action within the calling process. In the example above, the process P will be suspended until the process Q has completed the request. Therefore, recursion will not be realized in the language. In the case of internal requests, the process initialization will be suspended until the called procedure has completed the request.

A process is executed by executing the process initialization until it finishes, or until it reaches a synchronization statement where no synchronization condition is satisfied. In the latter case, the initialization is suspended and the process begins executing one of its procedures that has been called. If no call has yet been received, or the process initialization has finished, then the process waits for a procedure call. Therefore, the process never terminates. The process will execute a procedure call until the procedure finishes or until it reaches a synchronization statement where no synchronization condition is satisfied. The process will then either begin the execution of another procedure that has been called, or if one of the conditions of a previously suspended synchronization statement is now satisfied, resume one of those statements. This selection is made nondeterministically. The interleaving of the process initialization and procedures continues forever.

Apart from procedure calls and synchronization statements, procedures (and process initialization sections) include the usual statements: *SKIP*, assignment,
sequential composition, and Dijkstra's selection ($IF$) and repetition ($DO$) [16].

V.1.3 Nondeterminism

Nondeterminism can arise from the following statements which are called *guarded commands*. A guarded command causes the process to make an arbitrary choice between several statements based on the current state of the variables.

\[
\text{If } b_1 : S_1 | b_2 : S_2 | \ldots | b_n : S_n \text{ end}
\]

An arbitrary selection is made among those $b_i$ that evaluate to true and the corresponding $S_i$ is executed. If all the $b_i$'s evaluate to false, a program exception will occur.

\[
\text{Do } b_1 : S_1 | b_2 : S_2 | \ldots | b_n : S_n \text{ end}
\]

While any of the $b_i$ evaluate to true, arbitrarily select one of them and execute the corresponding $S_i$. The execution of the $DO$ statement terminates when all $b_i$ evaluate to false.

V.1.4 Synchronization

Synchronization is established by means of *guarded regions*. A guarded region causes a process to wait until the state of the variables allows an arbitrary choice among several statements to be made. If the state of the variables makes this selection impossible, then the process will postpone the execution of the guarded region.

\[
\text{When } b_1 : S_1 | b_2 : S_2 | \ldots | b_n : S_n \text{ end}
\]

If one or more of the $b_i$ evaluates to true, pick one of these $b_i$ and execute the corresponding $S_i$. If none of the $b_i$ evaluates to true, the procedure (or process initialization section) in which the $WHEN$ appears is suspended and will be resumed at some later time when one or more of
the $b_i$ evaluates to true.

$$\text{Cycle } b_1 : S_1 \mid b_2 : S_2 \mid \ldots \mid b_n : S_n \text{ end}$$

This is equivalent to the endless repetition of the *WHEN* statement.

We may compare the *WHEN* and *CYCLE* statements to the *IF* and *DO* statements: if all the guards are false, the *IF* statement raises an exception, the *DO* statement terminates, and the *WHEN* and *CYCLE* statements suspend their execution. The use of the *CYCLE* statement is restricted to the process initialization section since procedures are expected to terminate.

A DP process maybe sub-divided into three classes[42]:

1. A DP process which consists of a set of procedures and an initialization section which terminates before any procedures calls are allowed is a monitor in the conventional sense but with explicit wait and implicit signal operations as provided by the synchronization commands.

2. A DP process which contains no procedures and consists solely of an initialization section is a process in the conventional sense whose only communication is by synchronized procedure calls to other (monitor-like) processes.

3. By bringing together the process and monitor as a single “distributed process”, the sequential execution of the process body is nondeterministically interleaved with the execution of calls to its monitor-like procedures.

It is instructive to compare the effect of nondeterminism in a *WHEN* statement within a DP procedure and in a *SELECT* statement within an Ada task[41]. Both use a similar mechanism for selecting among viable alternatives when they are immediately executable and suspend themselves when no alternative can be executed. However, DP processes can activate a waiting procedure on an internal
procedure queue when the concurrent thread of control is suspended while tasks
must wait until an external event allows an alternative to be executed. The ability
to suspend and subsequently reactivate DP procedures cannot be easily simulated
in Ada. Tasks have no internal mechanism for initiating a secondary activity when
a primary activity has been interrupted.

In Ada, the entry call and the WHEN statement that are used as primitives for
inter process communication are taken directly from Brinch Hansen[10]. However,
Ada's ACCEPT statement is quite different from DP procedures. Brinch Hansen's
procedures are called out-of-line and perform a certain unit of work for the calling
process. Therefore, to impose a deterministic sequence on the accepted procedure
calls in DP, the programmer must use guards composed of state variables which are
reset at appropriate points in the code. Since Ada ACCEPT statements occur in
the middle of executable code (in-line procedures), the task defines how, when, and
how many times they are callable.

Within this thesis, integer data types have been used exclusively although other
data types can be added with minimal difficulty.

V.2 Bounded Buffer Example

Because of the difficulty of envisioning the mixture of synchronization com-
mmands and nonterminating processes, an illustrative example of a DP program is
given. The following example is the standard Bounded Buffer problem.

**BOUNDDED BUFFER**

Process BUFFER

\[ S: \text{Seq}[N] \text{ char} \]
\[ \text{Count: Int} \]
Proc SEND(c:char)
When Count < N:
    Count + = 1
    S[Count]:= c
end

Proc RECEIVE(#v:char)
When Count > 0:
    v:= S[Count]
    Count - = 1
end

Begin
    S:= []
    Count:=0
end

Process PRODUCER: ...... Call BUFFER.SEND(item) ......

Process CONSUMER:
    ...... Proc Qk ...... Call BUFFER.RECEIVE(#item) ......

The local variables of the process BUFFER are S and Count. Execution of the process BUFFER begins with the process initialization section. Once execution of the initialization section terminates, the process BUFFER waits for calls to its procedures SEND and RECEIVE. If the process PRODUCER calls the procedure BUFFER.SEND then the execution of the process PRODUCER is suspended until the procedure call is completed and control returns to PRODUCER. Notice that if the process CONSUMER calls BUFFER.RECEIVE when the buffer is empty, then
the execution of the procedure is suspended until an item is placed in the buffer by the process \textit{PRODUCER} calling the procedure \textit{BUFFER.SEND}. Similarly, if the buffer is full when \textit{PRODUCER} calls \textit{BUFFER.RECEIVE}, the procedure will be suspended until the \textit{CONSUMER} process removes an item from the buffer by calling the procedure \textit{BUFFER.SEND}.

\textbf{V.3 Introduction}

Our proof system is centered around the interactions made between procedures, the process initializations, and the processes defined within a program. These interactions are used to characterize the \textit{externally visible behavior} of any construct. The semantics of any construct $S$ is a simple formalization of what an \textit{external} observer of $S$ would see during the execution of $S$.

In order to easily express the sequence of actions that are visible to an external observer, we will associate a sequence $h_{i,k}$ with each procedure $Q_k$ of the process $P_i$. This sequence will record all interactions between $P_i, Q_k$ and agents external to $P_i, Q_k$: the calls to and returns from $P_i, Q_k$, all calls that $P_i, Q_k$ makes with their corresponding returns, and the suspensions and resumptions of $P_i, Q_k$. A similar sequence $h_{i,\text{init}}$ will be associated with the initialization section of $P_i$. We also associate a sequence $h_i$ with each process $P_i$ that will record all interactions between $P_i$ and other processes: calls and corresponding returns made by the procedures of $P_i$ to the procedures of the other processes, and the calls and corresponding returns made by the other processes to $P_i$'s procedures. Thus $h_i$ will essentially be a "merge" of all the $h_{i,k}$'s (and $h_{i,\text{init}}$).

Because of the nature of DP, we had to modify the standard Hoare-logic of $(p) S (q)$. Since processes never terminate, we could trivially conclude $(p) P_i (\text{false})$ \cite{33}. Therefore, we chose to use an invariant, instead of a postcondition, with the restriction that the invariant will be defined in terms of the interaction sequence $h_i$. 

associated with the process \( P_i \). The notation \((r) P_i\) will represent the following:

Given that the interaction sequence \( h_i \) associated with \( P_i \) satisfies the invariant \( r \) at the start, \( h_i \) will satisfy \( r \) during the execution of \( P_i \).

The invariant \( r \) will hold at all times during the execution of \( P_i \). Therefore, \( r \) must be chosen so that it will remain valid when elements are added to the corresponding interaction sequence. The invariant notation is used when verifying the \textit{CYCLE} statement, processes, and programs since these constructs do not terminate. When using this notation to verify the \textit{CYCLE} statement, we need to modify it by adding a precondition, i.e. \((r) \{p\} S\). Therefore, the meaning of the notation will be extended to include that the precondition \( p \) must be satisfied at the beginning of the execution of the \textit{CYCLE} statement.

We will use the standard Hoare-logic that has been extended to include an invariant to verify simple statements and procedures since the execution of these constructs may not necessarily terminate. The notation \((r) \{p\} S \{q\}\) will represent the following:

Given the precondition \( p \) and that the interaction sequence \( h \) associated with \( S \) satisfies the invariant \( r \) initially, when the execution of \( S \) terminates the postcondition \( q \) will be satisfied and \( h \) will satisfy \( r \) during the execution of \( S \).

The invariants of the procedures and process initialization will be combined to obtain the behavior of the process, and the invariants of the processes will be combined to obtain the behavior of the program.

The proof system was constructed by examining what an observer would witness as he stood at different locations within the operating program. At the lowest level, the observer standing within a procedure would witness a single execution of the procedure. This single execution would include any suspensions or resumptions that occurred during this execution. The observer will witness the suspensions and
resumptions since he (and the system) must be able to distinguish between a procedure's suspension or its completion of execution. If that observer were to move directly outside the procedure but still only observe this procedure, he would witness the numerous procedure calls (both external and internal) to this procedure, as well as its suspensions and resumptions.

If the observer moved away from that procedure but still remained within the process, he would witness all the calls made to the procedures of this process and the order of execution of the procedures and process initialization. This order of execution will be dictated by the suspensions and resumptions of the procedures, the execution of the process initialization, and the accepted procedure calls. Lastly, if the observer moved directly outside the process, he would witness the external procedure calls and returns move from the requesting process to the called process and back. He would not witness the suspensions and resumptions of the procedures and process initialization defined within this process since these actions do not involve any external agents.

Using this methodology, rules for the "higher level" constructs will specify how the behaviors of the lower level constructs may be combined to obtain those of the higher level constructs. A number of useful functions and predicates are defined in the Appendix.

V.4 Sequence Notation

The elements that comprise the interaction sequences will be of varying type and length. The first component will specify the type of the element (e.g. call, return, etc.). The second and third components of the element record the identity of the process and procedure (e.g. i,j,l,k). Therefore, an element associated with the call to the procedure Qj in the process Pi is represented as \((\text{call}, i,j, \ldots)\). If the element represents a procedure call then the next two components record the identity of the
process and procedure of the called procedure. If the element is associated with a procedure or procedure call, the next component records the parameter values (e.g. \( \overline{X}, \overline{V}, \overline{Z} \)). Expanding the previous example, the element \((\text{call}, i, j, \overline{X}, \ldots)\) represents a procedure call to \(P_i, Q_j\) with \(\overline{X}\) being the values of the \(\text{in}\) parameters. Lastly, for elements corresponding to all statements other than the external call statement, the current values of the process variables are recorded (e.g. \(R, T\)). Completing the example, the element \((\text{call}, i, j, \overline{X}, \overline{R})\) represents a procedure call to \(P_i, Q_j\) with \(\text{in}\) parameter values \(\overline{X}\) and process variable values \(\overline{R}\). If the element is associated with a statement that is contained within a procedure, then an incarnation number is added to the element by the system since a number of procedure calls can be incomplete at the same time.

Initially, we will consider the axioms and rules corresponding to the statements in the procedure \(P_i, Q_k\).

V.5 Axioms and Rules

**SKIP Statement**

\[
\begin{align*}
\quad & p \Rightarrow r \\
\Rightarrow & (r) \{ p \} \text{ Skip } \{ p \}
\end{align*}
\]

**Assignment Statement**

\[
\begin{align*}
\quad & p \Rightarrow r, \ p \Rightarrow q_i^X \\
\Rightarrow & (r) \{ p \} \ x := e \{ q \}
\end{align*}
\]
Sequential Composition

\[
(r) \ (p) \ S_1 \ {q'}, \ (r) \ {q'} \ S_2 \ {q} \\
\hline \\
(r) \ (p) \ S_1; S_2 \ {q}
\]

Rule of Consequence

\[
r' \Rightarrow r, \ p \Rightarrow p', \ (r') \ {p'} \ S \ {q'}, \ q' \Rightarrow q \\
\hline \\
(r) \ (p) \ S \ {q}
\]

Conjunction

\[
(r) \ (p) \ S \ {q}, \ (r') \ (p) \ S \ {q'} \\
\hline \\
(r \land r') \ (p) \ S \ {q \land q'}
\]

Disjunction

\[
(r) \ (p) \ S \ {q}, \ (r) \ {p'} \ S \ {q} \\
\hline \\
(r) \ (p \lor p') \ S \ {q}
\]

Body

\[
(r) \ (p) \ S \ {q} \\
\hline \\
(r) \ (p) \ \text{Begin S end} \ {q}
\]

V.5.1 IF Statement

\[
(r) \ (p \land b_j) \ S_j \ {q}, \ j = 1, \ldots, n \\
\hline \\
(r) \ (p) \ \text{If} \ b_1 : S_1 \ | \ldots | b_n : S_n \ \text{end} \ {q}
\]
V.5.2 DO Statement

\[(r) \{ p \land b_j \} S_j \{ p \}, \ j = 1, \ldots, n\]

\[(r) \{ p \} \text{Do} \ b_1 : S_1 | \ldots | b_n : S_n \ \text{end} \{ p \land \lnot b_1 \land \ldots \land \lnot b_n \}\]

V.6 CALL Statement (in \( P_i.Q_k \))

\[
p \Rightarrow r, \ p \Rightarrow r \mathcal{h}_{i,k}^{\text{call.to},i,k,l,j,X} \land q \Rightarrow r
\]

\[
p \Rightarrow \{ \forall \mathcal{Y}. q \mathcal{X}, \mathcal{K}, \mathcal{L} \mathcal{h}_{i,k}^{\text{call.from},i,k,l,j,Y,\mathcal{Y}} \} \land \text{return.from},i,k,l,j,Y
\]

\[
(r) \{ p \} \ \text{Call} \ P_i.Q_j(\mathcal{X}\#\mathcal{Z}) \{ q \}
\]

The effect of the statement \( \text{CALL} \ P_i.Q_j \) within the procedure \( P_i.Q_k \) is to extend the sequence \( h_{i,k} \) by two elements: one corresponding to the call and the other to the return. In addition, the values of the variables returned by \( P_i.Q_j \) will be assigned to the corresponding variables of \( P_i.Q_k \).

The first line ensures that the invariant \( r \) is satisfied before the execution of the external call statement begins and that \( r \) will remain satisfied after its execution terminates. The concatenation operation, \( \circ \), represents the addition of an element to the right end of a sequence. We add the element \( \text{(call.to},i,k,l,j,X) \) to the sequence in order to indicate that control leaves \( P_i.Q_k \) to call the procedure \( P_i.Q_j \), and that the values of the variables \( X \) are the arguments associated with this call. The other element, \( \text{(return.from},i,k,l,j,Y) \), is added to the sequence to indicate that control returns to \( P_i.Q_k \). The parameter values, \( Y \), that are returned by the procedure are assigned to the corresponding argument variables \( Z \). Since we are considering \( P_i.Q_k \) in isolation, we don't have any information on what values \( P_i.Q_j \) will return. The universal quantifier ensures that we consider all possible parameter values that \( P_i.Q_j \) could return. Note that a \( \text{CALL} \) statement may appear within the initialization
section of the process \( P_i \). For convenience, we assume that the initialization section is the 0th "procedure" of \( P_i \).

Using the Bounded Buffer example, the external procedure call issued by the process \( \text{CONSUMER} \) is recorded on the sequence \( h_{\text{Consumer}, k} \) as the elements

\[
\text{(call.to, Consumer, k, Buffer, Receive, \( q \))}
\]

and

\[
\text{(return.from, Consumer, k, Buffer, Receive, \( Y \))}
\]

with \( Y \) being the value returned by the procedure \( \text{BUFFER.RECEIVE} \).

The above rule applies only to external calls. The rule corresponding to an internal call (a call to \( P_i.Q_k \) from the initialization section of \( P_i \)) is different. We need to record the values left in the local variables of the process by the initialization section at the time of the call and the values left in the same variables by \( P_i.Q_k \) when control returns to the initialization section.

\[ p' \Rightarrow r, p \Rightarrow r, q \Rightarrow r \]

\[ p' \Rightarrow p_{k_i, \text{init}} \sim (\text{call.to}, i, 0, i, k, X, R) \]

\[ p \Rightarrow \forall \overline{Z}, \overline{R}. q_{k_i, \text{init}} \sim (\text{return.from}, i, 0, i, k, \overline{Z}, \overline{R}) \]

\[
(r) \quad \{p'\} \text{ Call } P_i.Q_k(\overline{X} \# \overline{Z}) \{q\}
\]

Let us suppose that the initialization section contains an internal procedure call to the procedure \( P_i.Q_k \). The first line ensures that the invariant \( r \) is satisfied at the start of the execution of the internal call statement and that \( r \) remains satisfied after its execution. In the second line, the first element records the values of the in parameters, \( \overline{X} \), and the current values of the process variables, \( \overline{R} \). When control returns to the process initialization, the second element records all possible
values of the out parameters, \( Z \), and process variable values, \( \overline{R} \). In the third line, the universal quantifier reflects the fact that we do not know what values will be returned by the procedure or the current process variable values.

V.7 WHEN Statement

\[
p \Rightarrow r, \{p \land \bigwedge_{j=1}^{n} \neg b_j \Rightarrow r_{h_{i,k}}^{k} \sim (\text{suspend},i,k,\overline{R})
\]

\[
(r) \{p \land b_j \} S_j \{q\}, j = 1, \ldots , n
\]

\[
(p_1 \land \bigwedge_{j=1}^{n} \neg b_j) \Rightarrow \{\forall t. \overline{p}_{h_{i,k}}^{k} \sim (\text{suspend},i,k,\overline{R}) \sim (\text{resume},i,k,\overline{R})\}
\]

\[
(r) \{p\} \text{ When } b_1 : S_1 | \ldots | b_n : S_n \text{ end } \{q\}
\]

The first line ensures that the invariant \( r \) is satisfied at the start of the execution of the WHEN statement. The second line considers the case when at least one of the \( b_j \) is satisfied; therefore, execution of the WHEN is not suspended. The third line considers the case when a suspension of the execution of \( P_s.Q_k \) occurs. This line ensures that the proper elements have been concatenated to the interaction sequence \( h_{i,k} \), with \( \overline{R} \) being the current values of the process variables of \( P_s \), and \( \overline{T} \) being the possible values of the process variables after the execution of the WHEN is resumed. The universal quantifier ensures that we consider all possible values that the process variables may have when the execution of \( P_s.Q_k \) resumes.

Again, returning to the Bounded Buffer example, let us suppose that the execution of the procedure \( BUFFER.RECEIVE \) becomes suspended due to the buffer being empty. Therefore, the elements added to the interaction sequence are as follows:

\[
(\text{suspend},Buffer,Receive,\{\},0) \text{ and } (\text{resume},Buffer,Receive,S',COUNT')
\]
where \( S' \) and \( COUNT' \) are the current values of the local variables of the process \( BUFFER \) at the time that \( BUFFER.RECEIVE \) resumes execution.

**V.8 Procedure Rule**

\[
(r) \{ h_{i,k} = (\langle \text{call},i,k,\overline{x},\overline{r} \rangle) \} S \{ p \}
\]

\[
\forall T. P^T \Rightarrow q^{h_{i,k}} \langle \text{return},i,k,\overline{y},\overline{r} \rangle
\]

\[
(r) \{ h_{i,k} = \varepsilon \} \text{Proc } Q_k(\overline{x}\#\overline{y}) \text{Begin } S \text{ end } \{ q \}
\]

Even though a procedure may be executed numerous times, we only need to verify the procedure once. Therefore, the sequence \( h_{i,k} \) will only record elements from a single execution of the procedure.

The precondition of \( S \) ensures that when the execution of \( S \) begins, the appropriate element corresponding to the procedure call has been recorded as the first element in \( h_{i,k} \). This element includes the in parameters, \( \overline{x} \), and the process variables, \( \overline{R} \). The second line ensures that the element recording the return from \( P_i.Q_k \) to the caller is added to the sequence \( h_{i,k} \). This element includes the out parameters and the current values of the process variables of \( P_i \). We use the universal quantifier to eliminate any explicit references to the process variables \( \overline{R} \) since \( q \) may only refer to the sequence \( h_{i,k} \).

**V.9 Externalize Procedure**

The next rule corresponds to our "observer" moving just outside the procedure \( P_i.Q_k \). He will then observe all the calls to (and returns from) \( P_i.Q_k \). We use the notation \( [P_i.Q_k] \) to represent this "externalized" view of \( P_i.Q_k \). In this view,
\[ [P_i, Q_k] \text{ never "terminates". Once a call to } P_i, Q_k \text{ finishes, } [P_i, Q_k] \text{ gets ready for the next call. Hence we need to use the invariant notation } (r) [P_i, Q_k] \text{ to deal with } [P_i, Q_k]. \text{ We will use the name } h^r_{i,k} \text{ for the sequence associated with } [P_i, Q_k]. \text{ The elements in } h^r_{i,k} \text{ will be somewhat different from the elements in } h_{i,k}. \text{ Since } h^r_{i,k} \text{ deals with several calls to } P_i, Q_k \text{ at the same time, the elements of } h^r_{i,k} \text{ will have an extra component, the incarnation number, corresponding to the particular call to } P_i, Q_k \text{ to which the given element is associated.}

\begin{align*}
(r) \{ h^r_{i,k} = \epsilon \} \ P_i, Q_k \ (g) \\
( \text{CHECK}(h^r_{i,k}) \land \forall m \leq \text{NO.CALLS}(h^r_{i,k}), r^{h^r_{i,k}}_{\text{SUBSEQ}(m, h^r_{i,k})}) \end{align*}

\text{CHECK}(h^r_{i,k}) \text{ checks the syntax of } h^r_{i,k}; \text{ in particular it ensures that } P_i, Q_k \text{ doesn't accept a call unless the previous call has completed or suspended. A precise definition of } \text{CHECK} \text{ is given in the Appendix. } \text{NO.CALLS}(h^r_{i,k}) \text{ is the number of calls, some of which may not be complete, to } P_i, Q_k \text{ in } h^r_{i,k}. \text{ SUBSEQ}(m, h^r_{i,k}) \text{ picks out the elements of } h^r_{i,k} \text{ that correspond to the } m^{th} \text{ call and makes up the sequence of these elements with their incarnation numbers removed. Thus, the invariant of } [P_i, Q_k] \text{ does the following:}

1) \text{ It ensures that } P_i, Q_k \text{ does not accept a call unless the previous call has completed or has been suspended.}

2) \text{ For each incarnation of } P_i, Q_k \text{ in } h^r_{i,k}, \text{ the portion } h^m_{i,k} \text{ of } h^r_{i,k} \text{ corresponding to the } m^{th} \text{ incarnation of } P_i, Q_k \text{ will satisfy the invariant of } P_i, Q_k. \text{ (Note that } h^m_{i,k} \text{ is the same as } \text{SUBSEQ}(m, h^r_{i,k}).\)
V.10 CYCLE Statement

\[ \bigwedge_j \neg b_j \Rightarrow r_{h_{i,k}}^{\text{suspend},i,0,\overline{R}} \]

\[(r) \{ p \land b_j \} S_j \{ p \}, \ j = 1, \ldots, n \]

\[ \{ p \land \bigwedge_j \neg b_j \} \Rightarrow \{ \forall^T, p_{T h_{i init}}^{\text{resume},i,0,\overline{R}} \} \]

\[ (r) \{ p \} \text{ Cycle } b_1 : S_1 \mid \ldots \mid b_n : S_n \text{ end} \]

The rule for the CYCLE statement is virtually identical to the WHEN rule with the exception that it establishes an invariant rather than a postcondition.

V.11 Process Initialization

The processes initialization statement may contain any of the statements that may appear in the procedures with the addition of the CYCLE statement. Therefore, the presence of the CYCLE statement will determine whether the process initialization section is expected to terminate or not.

\[ (r) \{ h_{i init} = \epsilon \} PI, \{ q' \} \]

\[ q' \Rightarrow q_{h_{i init}}^{\text{end},i,0,\overline{R}} \]

\[ (r \lor q) PI \]

If the processes initialization section terminates, we must record on the interaction sequence \( h_{i init} \) an element which represents its termination and contains the current values of the process variables. The element \( \text{end},i,0,\overline{R} \) is this element with \( \overline{R} \) representing the values of the process variables.
The verifications of the externalized view of the procedures and the process initialization statement of the process \( P_1 \) will be combined using this rule in order to verify \( P_1 \).

\[
\begin{align*}
( r_{i,k} ) & \mid P_i, Q_k, k = 1, \ldots, m \\
\frac{( r_{i, \text{init} } ) P_i}{( r_{i, \text{init} } \land \{ \forall j \leq m. r'_{i,j} \} \land \text{CONSIST}(h_i) ) P_i}
\end{align*}
\]

As our observer examines the interactions that occur within a process, he will witness the execution of the various procedures and the process initialization. Therefore, \( r'_{i, \text{init}} \) will verify that the sequence representing the interactions made by the process initialization that was taken from the process sequence \( h_i \) will satisfy the process initialization invariant \( r_{i, \text{init}} \). Also, \( r'_{i,j} \) will verify that the sequences representing the external actions of each procedure that are taken from the process sequence \( h_i \) will satisfy their corresponding procedure invariant \( r'_{i,k} \). A formal definition of \( r'_{i, \text{init}} \) and \( r'_{i,j} \) is given in the Appendix. Lastly, we must perform a consistency check, using the predicate \( \text{CONSIST} \), on the process sequence \( h_i \) to ensure that the observed values of the process variables are the values that were most recently assigned to them.

Using the Bounded Buffer example, the invariant for the process \( \text{BUFFER} \) will state that the number of returns from the procedure \( \text{BUFFER.RECEIVE} \) will be no more than the number of returns from the procedure \( \text{BUFFER.SEND} \).
V.13 Externalize Process

\[( r_i ) P_i \]

\[ ( \exists h_i. (h_i^{\text{ext}} = \text{STRIP}(h_i)) \land r_i ) |P_i| \]

When our observer moves outside of the process, he will witness the calls made to the procedures defined within the process \( P_i \) and any calls made by \( P_i \) to any of the procedures external to this process, but he will not witness the order of execution of the various procedures and the process initialization. Hence, the sequence \( h_i^{\text{ext}} \) that is seen by the observer should be obtained by STRIPping these elements from a sequence \( h_i \) that satisfies the invariant \( r_i \). Upon application of this axiom, the notation \( |P_i| \) will represent the externalized view of the process.

V.14 Parallel Composition

\[( r_i ) |P_i|, i = 1, \ldots, n \]

\[( r_1 \land \ldots \land r_n \land \text{COMPAT}(h_1^{\text{ext}}, \ldots, h_n^{\text{ext}}) ) \] \( |[P_1]| \ldots || |P_n| \]

Lastly, we will combine the individual process invariants and add a compatibility requirement, using the predicate \( \text{COMPAT} \), on the externalized view of the process interaction sequences. We will assume that no external procedure calls from another program are made to any of the processes defined within this program. The compatibility check ensures that any action in which two processes participate is recorded on the two process sequences in a mutually consistent fashion.
V.15 SORT Example

In order to compare our system with an existing system, we will use the following parallel sort program taken from Gerth et al.[20]. The program consists of the process $SORT_0$, the "user" process, and the processes $SORT_1, ..., SORT_n$ that implement the sort. The processes $SORT_1, ..., SORT_n$ maintain a sorted list of numbers given to them by $SORT_0$, with the smallest number being stored in $SORT_1$, the next smallest in $SORT_2$ ($SORT_1$'s immediate neighbor to the right), and so forth. This ordering is achieved by $SORT_1$ storing the smallest number and passing all other numbers to $SORT_2$, $SORT_2$ storing the second smallest number and passing all other numbers to $SORT_3$, and so forth. A process receives one of the remaining numbers from its successor after a number is output to its predecessor. The notation $SORT[i]$ represents $N-1$ identical processes where $SUCC$ refers to the next highest index of the processes. The $N^{th}$ process is slightly different since it does not have a successor; it is also given below.

This version of $SORT[33]$ is a modified version of the original found in [11]. It was determined that the guard in the procedure PUT, namely $Len < 2$, should be replaced by $Len = 1 \lor (Len = 0 \land Rest = 0)$. The original guard did not preserve the following characteristic of a priority queue.

$$\left( \bigwedge_{i=1}^{n} LEN_i = 1 \right) \implies \forall i < n \ [HERE_i \leq HERE_{i+1}]$$

Process $SORT[i]$

`Here : seq[2] int; Len, Rest, Temp: int;`

Proc PUT(c:int)

`When Len = 1 \lor (Len = 0 \land Rest = 0);`

`Len + := 1;
Here[Len] := c;`
end
end

Proc GET(#v:int)

When Len=1:

\[ v := Here[1]; \]
\[ Here := [ ]; \]
\[ Len := 0; \]

end
end

Begin

Here := [ ];
Rest := 0;
Len := 0;
Cycle

Len = 2:

If \[ Here[1] \leq Here[2]; \]

\[ Temp := Here[2]; \]
\[ Here := [Here[1]]; \]

\| Here[1] > Here[2]; \]

\[ Temp := Here[1]; \]
\[ Here := [Here[2]]; \]

end

Call SORT[SUCC].PUT(Temp);
Rest := 1;
Len := 1;

\| Len = 0 \& Rest > 0;

Call SORT[SUCC].GET(Temp)
Rest := 1;
Here := [Temp];
Len := 1;
end
end

Process \textit{SORT}_n

Here: seq[2] int; Len, Rest: int;

Proc PUT(c: int)
When Len = 0:
\begin{align*}
\text{Len}+ & := 1; \\
Here[\text{Len}] & := c;
\end{align*}
end
end

Proc GET(#v: int)
When Len = 1:
\begin{align*}
v & := Here[1]; \\
Here & := [ ]; \\
Len & := 0;
\end{align*}
end
end

Begin
\begin{align*}
\text{HERE} & := [ ]; \\
\text{REST} & := 0; \\
\text{LEN} & := 0;
\end{align*}
end
The process \textsc{Sort}_0 will submit the list of numbers to be sorted to the process \textsc{Sort}_1 and remove this list from \textsc{Sort}_1 one number at a time. The following process invariant for \textsc{Sort}_0 characterizes this behavior.

\[ r_0: \forall k. k \leq \#h_0 \ [\text{TYPE}(h_0, k) \in \{\text{"Call.to PUT"}, \text{"Call.to GET"}, \text{"Return from PUT"}, \text{"Return from GET"}\} \]

\[ \wedge \ [\text{NO. OF}(h_0[1 : k], \text{"Call.to GET"}) \leq \text{NO. OF}(h_0[1 : k], \text{"Call.to PUT"})] \]

Note: In the proof system given previously, we assumed that the procedures in a process are numbered 1, ..., m. In fact, as in this example, procedures usually have names rather than being numbered. Accordingly, the elements in the various sequences will refer to the procedures through their names rather than numbers. \( r_0 \) states that the only elements in \( h_0 \) correspond to calls to and returns from \textsc{Get} and \textsc{Put}; and that the number of calls to \textsc{Get} is less than or equal to the number of calls to \textsc{Put}.

Given the individual process invariants and \( r_0 \), the following program invariant \( r \) can be determined. The program invariant illustrates that the numbers taken from \textsc{Sort}_1 are indeed in ascending order. The function \textsc{Queue} maintains a sorted list of the elements that are received by the process \textsc{Sort}_1 and are contained within the processes \textsc{Sort}_1 through \textsc{Sort}_n.

\[ r: \forall k. k \leq \#h_0 \ [\text{TYPE}(h_0, k) = \text{"Return from GET"} \implies \]

\[ \text{VAL}(h_0, k) = \text{FIRST}(\textsc{Queue}(h_0[1 : k - 1])) \]

In order to simplify the following proof, the functions \textsc{Here}_i, \textsc{Len}_i, and \textsc{Rest}_i will use the program interaction sequence \( h \) to determine what the values of the local variables \textsc{Here}, \textsc{Len}, and \textsc{Rest} of the process \textsc{Sort}_i are at any given moment in time.
Basically, the proof of the program \textit{SORT} will examine the conjunction of the process invariants \( r_i \) and \( r_{i+1} \). We will establish that when the process \textit{SORT}, contains at least one element of the list of elements to be sorted, then the smallest element is less than or equal to any of the elements associated with the processes \textit{SORT}_{i+1}, \ldots, \textit{SORT}_n. Next, if we maintain a sorted list of all the elements that are received by \textit{SORT}_i from \textit{SORT}_{i-1}, the smallest element contained within \textit{SORT}_i will always correspond to the first (and therefore the smallest) element of the sorted list. Lastly, we will ensure that the elements passed from the process \textit{SORT}_{i-1} to the process \textit{SORT}_i are the same elements that \textit{SORT}_i passes back to \textit{SORT}_{i-1}. The proof, therefore, will verify each of these behaviors in order to establish the behavior of the entire program \((r)\).

To verify the program invariant \( r \), we must examine the relationship between the processes \textit{SORT}_i and \textit{SORT}_{i+1}. This relationship, \( r_i \land r_{i+1} \), has the following three properties. Once established, these properties will allow us to verify \( r \).

\begin{enumerate}
  \item \( \forall i \leq n \ (\forall k \leq \#h_i \ [[\text{LEN}_i(h_i[1:k]) \geq 1 \land \text{REST}_i(h_i[1:k]) > 0] \implies \forall j. i < j \leq n \ [\text{MIN}(\text{HERE}_i(h_i[1:k])) \leq \text{MIN}(\text{HERE}_j(h_j[1:k]))] \}) \)
\end{enumerate}

This first characteristic establishes that the smallest number associated with the sequence \textit{HERE}_i in \textit{SORT}_i is less than or equal to any of the numbers associated with the sequences \textit{HERE}_{i+1} through \textit{HERE}_n in the processes \textit{SORT}_{i+1} through \textit{SORT}_n. We will verify this characteristic using proof by induction on the length of the program interaction sequence. Therefore, we want to show the following.

\begin{enumerate}
  \item \( \forall i \leq n \ [[\text{LEN}_i(h'/i) \geq 1 \land \text{REST}_i(h'/i) > 0 \land h' \subseteq h] \implies \forall j. i < j \leq n \ [\text{MIN}(\text{HERE}_i(h'/i)) \leq \text{MIN}(\text{HERE}_j(h'/j))] \}) \)
\end{enumerate}

If \( h = \epsilon \) then the previous statement is satisfied. Next, we will assume there exists an \( h'' \) that is not the empty sequence that satisfies the previous statement. If
we extend h" by an element involving P_j, what effect does this communication have on P_i (i\neq j)? If (j+1) < i then this communication does not effect the validity of the previous statement; although, if i \leq (j+1) then we must examine the following four communications that could have occurred.

1. \( h' = h'' \land \text{"RETURN from PUT}_{j+1} \)"

The following two situations occur:

1. \( \text{MIN}(HERE_j(h''/j))=\text{MIN}(HERE_j(h'/j)) \)

2. If it is true that

\[
\text{MAX}(HERE_j(h''/j)) \leq \text{MIN}(HERE_{j+1}(h''/j+1))
\]

then

\[
\text{MAX}(HERE_j(h''/j))=\text{MIN}(HERE_{j+1}(h'/j+1))
\]

or if it is true that

\[
\text{MAX}(HERE_j(h''/j)) > \text{MIN}(HERE_{j+1}(h''/j+1))
\]

then

\[
\text{MIN}(HERE_{j+1}(h''/j+1))=\text{MIN}(HERE_{j+1}(h'/j+1))
\]

2. \( h' = h'' \land \text{"RETURN from GET}_{j+1} \)"

It is true that

\[
\text{MIN}(HERE_{j+1}(h''/j+1))=\text{MIN}(HERE_j(h'/j))
\]

3. \( h' = h'' \land \text{"RETURN from PUT}_j \)"

The following two situations occur:

1. \( \text{MIN}(HERE_{j-1}(h''/j-1))=\text{MIN}(HERE_{j-1}(h'/j-1)) \)
2. If it is true that
\[ MAX(HERE_{j-1}(h''/j-1)) \leq MIN(HERE_j(h''/j)) \]
then
\[ MAX(HERE_{j-1}(h''/j-1)) = MIN(HERE_j(h''/j)) \]

or if it is true that
\[ MAX(HERE_{j-1}(h''/j-1)) > MIN(HERE_j(h''/j)) \]
then
\[ MIN(HERE_j(h''/j)) = MIN(HERE_j(h''/j)) \]

4. \( h' = h'' \sim \text{"RETURN from GET}_i\)" 

Then it is true that
\[ MIN(HERE_j(h''/j)) = MIN(HERE_{j-1}(h''/j-1)) \]

In all four situations, the first characteristic holds. Now, we will examine what happens if \( h'' \) is extended by an element involving \( P_i \). Initially, we will assume \( REST_i(h''/i) > 0 \). Now, we must examine the three possible values of \( LEN_i(h''/i) \).

1. \( LEN_i(h''/i) = 0 \)

The only possible communication is
\[ h' = h'' \sim \text{"RETURN from GET}_i+1\)" 

and it is true that
\[ MIN(HERE_{i+1}(h''/i+1)) = MIN(HERE_i(h''/i)) \]

2. \( LEN_i(h''/i) = 1 \)

If the next communication is
\[ h' = h'' \sim "RETURN from PUT_i," \]

Then it is true that
\[ \text{MAX}(HERE_{i-1}(h''/i-1)) \leq \text{MIN}(HERE_i(h''/i)) \]

and
\[ \text{MAX}(HERE_{i-1}(h''/i-1)) = \text{MIN}(HERE_i(h''/i)) \]

Or it is true that
\[ \text{MAX}(HERE_{i-1}(h''/i-1)) > \text{MIN}(HERE_i(h''/i)) \]

and
\[ \text{MIN}(HERE_i(h''/i)) = \text{MIN}(HERE_i(h''/i)) \]

If the next communication is
\[ h' = h'' \sim "RETURN from GET_i," \]

Then it is true that
\[ \text{MIN}(HERE_i(h''/i)) = \text{MIN}(HERE_{i-1}(h'/i-1)) \]

and
\[ \text{LEN}_i(h'/i) = 0 \text{ so the statement is vacuously true.} \]

3. \( \text{LEN}_i(h''/i) = 2 \)

The only possible communication is
\[ h' = h'' \sim "RETURN from PUT_{i+1}" \]

Then if it is true that
\[ \text{MAX}(HERE_i(h''/i)) \leq \text{MIN}(HERE_{i+1}(h''/i+1)) \]

then
\[ \text{MAX}(HERE_i(h''/i)) = \text{MIN}(HERE_{i+1}(h'/i+1)) \]

Or if it is true that
\[
\text{MAX}(\text{HERE}_i(h''/i)) > \text{MIN}(\text{HERE}_{i+1}(h''/i+1))
\]

then
\[
\text{MIN}(\text{HERE}_{i+1}(h''/i+1)) = \text{MIN}(\text{HERE}_{i+1}(h'/i+1))
\]

Lastly, we must examine when \(\text{REST}_i(h''/i) = 0\) and \(\text{REST}_i(h'/i) > 0\). This situation can only occur when

\[h' = h'' \sim \text{return from PUT}_{i+1}\]

and it is true that
\[
\text{MAX}(\text{HERE}_i(h''/i)) = \text{MIN}(\text{HERE}_{i+1}(h'/i+1))
\]

\[
(2) \forall i. 1 \leq i < n \ [\forall k. k \leq \#h/i [LEN_i(h/i[1:k]) \geq 1 \implies
\text{MIN}(\text{HERE}_i(h/i[1:k])) = \text{FIRST}(\text{LIST}_i(h/i[1:k]))]
\]

The second characteristic illustrates the relationship between the sequence \(\text{HERE}_i\) in \(\text{SORT}_i\), and the numbers passed to the process \(\text{SORT}_i\) from \(\text{SORT}_{i-1}\) and contained within the processes \(\text{SORT}_1, \ldots, \text{SORT}_n\). Again, we will use induction on the length of the sequence \(h\). Clearly, \(h = \epsilon\) satisfies the previous statement. Next, we will assume there exists an \(h''\) that is not the empty sequence that satisfies the previous statement. Therefore, the possible communications that could occur are as follows.

1. \(h' = h'' \sim \text{return from GET}_i\)

\(LEN_i = 0\) so the previous statement is vacuously true.

2. \(h' = h'' \sim \text{return from PUT}_i\)

If \(LEN_i(h'/i) = 1\) then
MIN(HERE(h'/i)) = LIST(h'/i) = FIRST(LIST(h'/i))

If LEN(h'/i) = 2 then either

MIN(HERE(h'/i)) = MIN(HERE(h''/i))

and

MIN(HERE(h'/i)) = FIRST(LIST(h''/i))

= FIRST(LIST(h'/i))

or

MIN(HERE(h'/i)) < MIN(HERE(h''/i))

and

MIN(HERE(h'/i)) = FIRST(MIN(HERE(h''/i)))

LIST(h''/i))

3. h' = h" \sim \text{"RETURN from PUT}_{i+1}"

It is true that

MIN(HERE(h''/i)) = MIN(HERE(h'/i))

and

MIN(HERE(h'/i)) = FIRST(LIST(h''/i))

= FIRST(LIST(h'/i))

4. h' = h" \sim \text{"RETURN from GET}_{i+1}"

It is true that

MIN(HERE(h'/i)) = MIN(HERE_{i+1}(h''/i+1))

= FIRST(LIST_{i+1}(h''/i+1))

and (using (1) and the definition of LIST_{i})
(3) \( \forall i. 1 \leq i < n \)

\[ \forall k. k \leq \#h/i \ [\text{TYPE}(h/i,k) = \text{"RETURN from GET,"}] \implies \]

\[ [\text{VAL}(h/i,k) = \text{FIRST}(\text{LIST}_i(h/i[1:k-1]))] \]

The third characteristic ensures the correspondence between the numbers passed from \( \text{SORT}_i \) to \( \text{SORT}_{i-1} \) and the numbers that were passed from \( \text{SORT}_{i-1} \) to \( \text{SORT}_i \). This characteristic follows almost directly from the previous one since

\( \text{TYPE}(h/i,k) = \text{"RETURN from GET,"} \implies [\text{LEN}_{i-1}(h/(i-1)[1:k]) = 1 \]

\[ \land \text{MIN}(\text{HERE}_i(h/i[1:k-1])) = \text{MIN}(\text{HERE}_{i-1}(h/(i-1)[1:k])) \]

\[ \land \text{VAL}(h/i,k) = \text{MIN}(\text{HERE}_i(h/i[1:k-1])) \]

\[ = \text{FIRST}(\text{LIST}_i(h/i[1:k-1]))] \]

It is not difficult to see that this last characteristic directly implies the program invariant \( r \).

V.16 Summary

We have presented a proof system for the language Distributed Processes which can be used for proving the correctness of DP programs. We used the concept of hierarchical decomposition in order to examine the different components of the program. This allowed us to verify a procedure by examining the externally observable behavior of a single execution of a procedure. After obtaining the verifications of the multiple executions of the procedures and the process initialization section, they were combined in order to verify a process. To obtain the externally visible behavior of the process, a projection is taken of the corresponding interaction sequence.
Finally, these verifications were combined to verify the entire program. As we examined higher and higher levels of the program, we combined the results from the previous level in order to draw conclusions about the next level.

By contrast, the proof system proposed by Gerth et al.[20] require assumptions about the behavior of the remaining processes in the proof of a process which must be justified in a "proof of cooperation". Since no proof of cooperation is needed in the system proposed, we feel that our methodology usually results in fairly simple proofs. This claim can be envisioned by comparing the proof of the Sort example to its counterpart given in[20]. Moreover, their proof system cannot handle the internal CALL statement.

Gerth et al.[20] are forced to add a number of additional assertions in order to adapt the methodology of Apt, Francez, and de Roever[4] to handle DP. They construct from a proof outline "frontiers of computation" which are the state of the program at intermediate points. Next, they determine a safety assertion which is an invariant over the computation of a program that asserts what the program state should obey when control arrives at some (or all) intermediate points in the program. Then, to show that some safety assertion holds for a program, one must construct a proof outline and show that the frontiers of computation derived from this outline imply the corresponding state assertion.

The methodology used to construct this proof system could easily be extended to higher and higher levels of parallelism. We stopped at the program level since DP could not support the definition of any higher program levels. It is our suggestion that DP be extended in order to support hierarchical decomposition.
CHAPTER VI

Universality of Approach

In this chapter, we will provide justification that our modular approach is universal and can be used to define the (partial correctness) axiomatics of any type of concurrency. Continuing the demonstration of applying our modular approach to define the semantics of concurrent programming languages, we will outline the work of [37] where this approach was used to define the semantics of a shared variable language. It will be shown that the four languages highlighted in this thesis are representative of the three classes of concurrent programming languages. Lastly, it will be argued that the semantics of any constructs that were not previously discussed can be defined using the given approach.

VI.1 Shared Variable Language

This parallel programming language uses communication by means of shared variables for synchronization. In this language, the assignment statements that occur inside a process of a COBEGIN differ from the assignments that occur outside a COBEGIN. The former assignments can have at most one reference to a shared variable, whereas no such restriction applies to the latter assignments. For convenience, we shall use the convention that the variables shared between processes of a COBEGIN have the names $r_1, \ldots, r_m$. The sequences of values of variables shared between the processes play an important and explicit role in our approach.
Therefore, for each process $P_i$ we will associate a sequence $h_i$ that will contain the values of the shared variables as seen by $P_i$.

VI.1.1 Shared Variable Assignment

\[
\{p_{h_i}^{\forall t}(\text{send}_j,t)\} r_j := t \{p\}
\]

The effect of an assignment to a shared variable $r_j$ in the process $P_i$ will be to extend the sequence $h_i$ by the element $\langle \text{send}_j, t \rangle$ where $t$ is the value of the expression in the current state of $P_i$ that is assigned to the $j^{th}$ shared variable.

VI.1.2 Assignment Statement

\[
\{\forall t. p_{e_j[r_j \leftarrow t]}^{x,h_i} \langle \text{receive}_j,t \rangle\} x := e_j \{p\}
\]

The effect of the assignment $x:=e_j$ in the process $P_i$ should be to assign the value of $e_j$ to $x$; however, the value of $e_j$ depends on the current value of $r_j$. If we assume that the current value of $r_j$ is $t$ then the effect of the assignment is to assign the value of $e_j$ with $r_j$'s value being $t$ ($e_j[r_j \leftarrow t]$) to $x$ and to extend the sequence $h_i$ by $\langle \text{receive}_j,t \rangle$. The rule contains a universal quantifier over $t$ since we have no idea what the current value of $r_j$ is.
VI.1.3 \textit{AWAIT} Statement

\begin{align*}
\forall T. \left[ \left( p \land e[R \leftarrow T] \land R = T \right) \Rightarrow p' \uparrow_{h_i} \langle \text{RECEIVE},(1,t_1),\ldots,(m,t_m) \rangle \right] \\
\{ p' \} S \{ q' \}
\end{align*}

\begin{align*}
\forall T. q' \uparrow_{h_i} R & \Rightarrow q_{h_i} \uparrow_{h_i} \langle \text{SEND},(1,t_1),\ldots,(m,t_m) \rangle \\
\{ p \} \text{ Await } e \text{ Then } S \{ q \}
\end{align*}

Consider the statement \textit{AWAIT} $e$ \textit{THEN} $S$ in the process $P_i$. If the value of the expression $e$ is true then $S$ is executed while the remaining processes are suspended. If the value of $e$ is false, $P_i$ is suspended for some period of time and $e$ is evaluated again to see if it now evaluates to true. In order to evaluate $e$, $P_i$ would have to obtain the values of the shared variables $R$. We record the fact that $P_i$ observed the values $T$ for $R$ by adding the element $\langle \text{RECEIVE},(1,t_1),\ldots,(m,t_m) \rangle$ to $h_i$. The postcondition of $S$, $q'$, may contain explicit references to $R$; however, the postcondition for the \textit{AWAIT} statement, $q$, can only refer to the sequence $h_i$. Also, we must record in $h_i$ the values that $P_i$ has assigned to $R$ during the execution of $S$ by adding the element $\langle \text{SEND},(1,t_1),\ldots,(m,t_m) \rangle$.

VI.1.4 Parallel Composition

\begin{align*}
\{ p_i \land h_i = e \} P_i \{ q_i \}, i = 1, \ldots, n \\
\{ p_1 \land \ldots \land p_n \land R = T \} \text{ Cobegin } P_1 \| \ldots \| P_n \text{ coend} \\
\{ q_1 \land \ldots \land q_n \land \text{COMPAT}''(\overline{T},h_1,\ldots,h_n) \}
\end{align*}

The initial values of the shared variables $R$ are $T$. The compatibility requirement, \textit{COMPAT}'', ensures that the values of the shared variables found in the
elements of \textit{TYPE receive}, and \textit{RECEIVE} are the values that the various processes had assigned to the shared variables. Examples of the application of this proof system are given in [37].

\textbf{VI.2 Classification of Concurrent Languages}

Despite the large variety of different concurrent programming languages, each can be viewed as belonging to one of three classes: procedure oriented, message oriented, or operation oriented. Languages in the same class provide the same basic kinds of mechanisms for process interaction and have similar attributes.

In procedure-oriented languages[1], process interaction is based on shared variables. These languages contain both active objects (processes) and shared, passive objects. Passive objects are represented by shared variables, usually with some procedures that implement the operations on the objects. Processes access the passive objects they require directly and thus interact by accessing shared objects. Because passive objects are shared, procedure-oriented languages must provide a means for ensuring mutual exclusion. The shared variable language introduced in section VI.1 is an example of such a language.

Message-oriented languages[1] provide \textit{SEND} and \textit{RECEIVE} operations as the primary means for process interaction. In contrast to procedure-oriented languages, there are no shared, passive objects – therefore, processes cannot directly access all objects. Instead, each object is managed by a single process. When an operation is to be performed on an object, a message is sent to the process containing the object to perform that operation, which may then respond with a completion message. It must be kept in mind, though, that the calling process can only request that the operation be performed; it can not require that the containing process perform the operations. CSP is an example of a message-oriented language.

Operation-oriented languages[1] provide remote procedure calls as the primary
means for process interaction. These languages combine the aspects of the other two classes. The difference between this class and the previous two is that in operation-oriented languages the caller of an operation and the caretaker process that implements it synchronize while the operation is executed. Once the execution of the operation finishes, both processes then proceed asynchronously. Distributed Processes and Ada are examples of operation-oriented languages.

Languages in any of these three classes are roughly equivalent in expressive power. It has been argued that procedure-oriented and message-oriented languages are equals in terms of expressive power, logical equivalence, and performance[1]. At the time of this analysis, operation-oriented languages only recently came into existence. At an abstract level, the three types of languages are interchangeable. One can transform any program written using the mechanisms found in the languages of one class into a program using the mechanisms of another class without affecting performance. However, the classes emphasize different styles of programming. Program fragments that are easy to describe using the mechanism of one class can be awkward to describe using the mechanisms of another.

VI.3 Applicability to Remaining Constructs

There are three main issues that underlie all concurrent programming notations:

(1) how to express concurrent execution
(2) how processes communicate, and
(3) how processes synchronize.

We have already shown how our modular approach to verification has been applied to a number of concurrent programming constructs from our analysis of the four languages but unfortunately not all notations have been examined. First, we
will enumerate the concurrent language notations that already have been examined.

I. Specifying Concurrent Execution
   1. Coroutines
   2. COBEGIN Statement
   3. Process Declarations

II. Synchronization Based on Shared Variables
   1. Busy-Waiting
   2. Semaphores
   3. Conditional Critical Regions
   4. Monitors

III. Synchronization Based on Message Passing
   1. Specifying Channels of Communication
   2. Synchronization
   3. Remote Procedure Calls

Next, we will explore how our modular approach to verification can be applied to define the semantics of the concurrent language notations that have not been previously examined.

VI.3.1 FORK and JOIN Primitives

The FORK statement specifies that a designated routine should start executing while the invoking routine continues executing. To synchronize with the completion of the invoked routine, the invoking routine can execute a JOIN statement. Executing a JOIN delays the invoking routine until the designated invoked routine has terminated. The FORK primitive provides a direct mechanism for dynamic
process creation which includes multiple activations of the same program text. We will assume that multiple FORKs to the same program will be paired with their corresponding JOINs in the reverse order of occurrence.

The FORK statement has been marred by a failure to separate process definition from process synchronization. Later proposals to concurrency separate these distinct concepts and impose some structure on a concurrent program. This structure allows easy identification of those program segments that can be executed concurrently. Consequently, such proposals are well suited for use with the axiomatic approach because the structure of the program itself clarifies the proof obligations. Therefore, it will be difficult to define the semantics of these primitives using our axiomatic verification approach.

Let us assume that a FORK can only name another program. We will also disallow recursion by enforcing the restriction that a program can only FORK to another program whose name is already known from the original program's context. Given these restrictions, we can view the program named in a FORK as a procedure in the language DP. Therefore, for each program named in a FORK, we will verify a single execution of the program and then consider an "externalized" view of this program which will include an interaction sequence that represents an interleaving of all the single incarnation interaction sequences of the program. Since multiple executions of the same program can be active at the same time, we will require that an incarnation number be added to the interaction sequence elements. This will not only allow us to distinguish between the numerous incarnations of the same program but will also aid in matching the appropriate FORK and JOIN statements with their corresponding incarnation of the named program. Given that these programs communicate by means of shared variables, the program sequence will be required to contain elements that are similar to those defined in the shared variable language which record the observations of the values of and the assignments to these variables.
Lastly, we will have to modify the function of the predicate \textit{COMPAT} in order to pair the elements of \textit{TYPE} fork and join to the elements of \textit{TYPE} call and return respectively. Remember that all the elements will contain an incarnation number and this number will be used to determine which elements match.

\textbf{VI.3.2 Nested Monitor Calls}

When structuring a system as a hierarchical collection of monitors, it is likely that monitor procedures will be called from within other monitors. Much discussion has arisen over what should be done if a process having made a nested monitor call is suspended in another monitor. Processes that attempt to invoke procedures in monitors from which nested calls have been made will become blocked. This situation can influence performance since blockage will decrease the amount of concurrency exhibited by the system.

An example of a hierarchical collection of monitors would be the hierarchical extension of DP. For example, the \textit{SORT} program could be considered as a separate monitor that can be “called” with a collection of elements to be sorted.

The adjustments that must be made to the rules in the proof system for DP are concerned only with the definition of the predicate \textit{COMPAT}. The new \textit{COMPAT} would not only determine whether the process interaction sequences are mutually compatible but will also construct a program interaction sequence that will consist of all elements of \textit{TYPE} call and return that did not have matching call.to or return from elements in the interaction sequences that are the arguments to \textit{COMPAT} at this level. These elements will represent all calls made from monitors at the “program” level. Notice that \textit{COMPAT} will have to ignore these elements when determining whether the interaction sequences under consideration are compatible. This constructed program sequence along with its invariant will be considered with the other monitor sequences and invariants in order to verify the program at this
higher level. This procedure can be applied repeatedly in order to define "higher" program levels.

VI.3.3 Path Expressions

An approach to defining a module subject to concurrent access is to provide a mechanism whereby a programmer can specify in one place in each module, all constraints on the execution of operations defined by that module. This is the approach taken in a class of synchronization mechanisms called path expressions. Using path expressions, a module that implements a resource has a structure that is similar to a monitor. The module will contain permanent variables, which store the state of the resource, and procedures which realize the operations on the resource. Path expressions in the header of each resource define the constraints on the order in which operations are executed. No synchronization code is programmed in the procedures. While path expressions provide an elegant notation for expressing synchronization constraints that are described operationally, they are poorly suited for specifying condition synchronization (to allow a process to perform an operation only when the state of the shared data object is appropriate for executing that particular operation). The determination of whether an operation can be executed might depend on the state of a resource which might not be directly related to the history of the performed operations.

Using path expressions, a bounded buffer would be defined by:

Module BUFFER(T);

Path N:(1:(DEPOSIT);1:(FETCH)) end;

Var Slots:Array[0...N-1] of T;

Head, Tail: 0...N-1;

Procedure DEPOSIT(p:T);
Begin

\[ \text{Slots}[\text{Tail}] := p; \]
\[ \text{Tail} := (\text{Tail} + 1) \mod N \]
end;

Procedure \text{FETCH}(\text{Var } it:T);
Begin
\[ it := \text{Slots}[\text{Head}]; \]
\[ \text{Head} := (\text{Head} + 1) \mod N \]
end;

Begin
\[ \text{Head} := 0; \]
\[ \text{Tail} := 0 \]
end

where \text{Path } N: (1:(\text{DEPOSIT});1:(\text{FETCH})) \end; \] ensures that

(i) activations of \text{DEPOSIT} are mutually exclusive

(ii) activations of \text{FETCH} are mutually exclusive

(iii) each activation of \text{FETCH} is preceded by a completed \text{DEPOSIT}, and

(iv) the number of completed \text{DEPOSIT} operations is never more than \( N \) greater than the number of completed \text{FETCH} operations.

Since path expressions eliminate the use of synchronization statements, a monitor interaction sequence will contain elements that represent calls made to the procedures defined within the monitor, calls made to procedures defined outside of the monitor and an element that will record the values of the monitor variables after the initialization section of the monitor has terminated. These elements will record
the same information as the corresponding elements defined in the proof system for the language DP. Therefore, we can use the rules defined for the DP procedures and processes in order to verify the monitor. However, we must modify the process rule so that the process invariant will include the invariant that represents the behavior of the path expression. This invariant can be determined by parsing the path expression which is actually a regular expression. Lastly, we can use the DP parallel composition rule to verify the program.

VI.3.4 Atomic Transactions

Ideally, we would like a remote procedure call to have the semantics that each call terminate only after the named remote procedure has been executed once. Unfortunately, a failure may mean that the caller is forever waiting for the response to a remote procedure call. A failure might occur if

(i) the message signifying the remote procedure invocation is lost by the network
(ii) the reply message is lost, or
(iii) the called procedure's host crashes during the execution of the remote procedure.

An atomic transaction is an all-or-nothing computation — either it installs a complete collection of changes to some variables or it installs no changes (even if interrupted by a failure). Moreover, atomic transactions are assumed to be indivisible in the sense that partial execution of an atomic transaction is not visible to any concurrently executing atomic transaction.

Our verification approach has been applied to fault-tolerant programs[27] where a distributed system can detect node crashes and can restart the failed node after alerting the remaining nodes in the system of the failure. In this work, the proof system of CSP was modified to handle "implicit" communication which represent
the failed process notifying the system of its failure and notification of an attempted communication with a failed and restarted process. The standard Hoare-logic was also extended to include an invariant (it is described using the same notation as the invariant given in the DP proof system). This invariant captures the behavior of the failure-prone process from the behavior of the failure-free process. The definition of COMPAT must be slightly modified in order to handle the elements that represent the internal communications.

We will assume that there exist user and transaction processes. A process that corresponds to a transaction can not issue a remote procedure call to another transaction process. We will also assume that the user process P, is notified of the failure of the transaction process by the addition of an element that represents the failure to its interaction sequence. As in the previous work, we can use the invariant notation to capture the failure-prone process behavior from the failure-free process behavior. Notice that this behavior only applies to the user process. The function of the predicate COMPAT will have to be modified so that it will not only match the elements that represent a "failure-free" remote procedure call and its return but will pair the "failed" remote procedure call elements in the user process' interaction sequence to the empty sequence in the transaction process' interaction sequence since this call theoretically never took place.

VI.3.5 Asynchronous Message Passing

With asynchronous message passing, the SEND statement is nonblocking since its execution never delays its invoker. Therefore, the system effectively has unbounded buffer capacity. Asynchronous message passing also allows a sender to get arbitrarily ahead of a receiver. Consequently, when a message is received, it contains information about the sender's state that is not necessarily its current state. Notice that any synchronous message passing language can be used to sim-
ulate asynchronous message passing by specifying a process to perform as a buffer. However, this process must theoretically possess unbounded memory.

We can simulate a language that uses asynchronous message passing by using the language CSP with the modification that CSP will not require the synchronization of communication between processes. This modification will not affect the rules that comprise the proof system of CSP since these rules do not require the use of the information that processes synchronize during communication. However, we must adjust the function of the predicate COMPAT. For example, the following CSP program will deadlock, but will terminate normally with the postcondition \( \{x = 0 \land y = 1\} \) if CSP uses asynchronous message passing.

\[
P_1:: \quad P_2!1; \quad P_2?x
\]

\[
P_2:: \quad P_1!0; \quad P_1?y
\]

Therefore, COMPAT must be modified to match the corresponding I/O commands despite the order in which they occur within a process.

VI.4 Summary

To justify our claim that our modular approach is universal, we began our argument with a quick summary of [37] that illustrates how the axiomatics of a simple concurrent language with shared variables may be defined using our approach. We then considered, in turn, each of the following constructs and briefly explained how each of them could be handled using our approach: FORK and JOIN primitives, nested monitor calls, path expressions, atomic transactions, and asynchronous message passing. We believe Chapters II, IV and V including the discussion in this chapter provide ample evidence to support the claim of universality of our approach.
CHAPTER VII

Soundness and Completeness of Approach

VII.1 Introduction

We now turn to the important task of demonstrating the soundness and (relative) completeness of our axiomatics. Initially, we prove the soundness and completeness of the EXIT statement using a modified version of the standard operational model of Apt et al.\[3\]. We then argue that the approach used in \[36\] to demonstrate the soundness and completeness of the CSP proof system of \[35\] can be adapted to demonstrate the soundness and completeness of proof systems defined using our approach for any concurrent language. This operational model deals with processes in isolation as does our modular approach to verification. To demonstrate how these methods might be used, we will define an operational model for the concurrent language DP and argue that the proof system given for DP in Chapter V is both sound and complete.

Our choice of an assertion language must permit the expression of the weakest precondition and the strongest postcondition as assertions. A global correctness property \{p\} S \{q\} will in practice have recursive assertions p and q. A natural conjecture then is that all intermediate assertions may also be chosen recursive. However, it has been shown in Apt et al.\[3\] that the set of recursively enumerable assertions is sufficiently large to allow the intermediate assertions to be chosen from them. Therefore, any assertion language that allows us to express the assertion that
corresponds to a set of states is acceptable.

VII.2 Operational Model for Sequential Statements

Our proof of the soundness and completeness of the EXIT statement is based on that given in Apt et al.\cite{3} for a proof system for a language without EXITs or GOTOs. In this work, a program is a binary relation over the state space $\Omega$ of initial and final states where $\Omega$ represents the set of all possible states. An assertion is a subset of $\Omega$. Let $p, q$ represent assertions and $\sigma, r$ represent states. Using this notation, the operational model of \cite{3} defined as follows:

$$\{p\} S \{q\} \text{ iff } \forall \sigma, r[(\sigma \in p \land \sigma S r) \Rightarrow r \in q]$$

Weakest Precondition

$$\text{WP}(S, q) = \{\sigma \mid \forall r(\sigma S r \Rightarrow r \in q)\}$$

Strongest Postcondition

$$\text{SP}(p, S) = \{r \mid \exists \sigma(\sigma \in p \Rightarrow \sigma S r)\}$$

Negation

$$\neg p = \{\sigma \mid \sigma \notin p\}$$

Sequential Composition

$$S_1; S_2 = \{(\sigma, r) \mid \exists \sigma'(\sigma S_1 \sigma' \land \sigma' S_2 r)\}$$
Looping

\[ p \circ \circ S = \{(\sigma, r) | \exists \sigma_0, \ldots, \sigma_n [\sigma = \sigma_0 \land \sigma_n = r \land
\forall i < n [\sigma_i \in p \land \sigma_i \sigma_{i+1}]]) \}

While

\[ p \circ S = \{(\sigma, r) | \sigma(p \circ S)r \land r \in -p) \]

Two obvious but important properties of the notations introduced are:

\[(p) S \{q\} \text{ iff } p \subseteq \text{WP}(S, q)\]

\[(p) S \{q\} \text{ iff } \text{SP}(p, S) \subseteq q\]

VII.3 Operational Model of EXITs

In our operational model, the operational definition of each statement \( S \) will consist of two components. The first component \( S^1 \) represents the normal termination of \( S \) and the second component \( S^2 \) represents a jump from \( S \) via an EXIT \( L \) statement; thus, these two components will correspond respectively to the postcondition and exit-condition of our axiomatics.

\[ S :: (S^1, S^2) \]

\[ S^1 \equiv \{(\sigma, r) | \sigma \xrightarrow{S} r\} \]

\[ S^2 \equiv \{(\sigma, L : r) | \sigma \xrightarrow{S} L : r\} \]

The first component of \( S \) represents all pairs of states such that when execution of \( S \) begins in the state \( \sigma \), the execution of \( S \) terminates in the state \( r \). The second
component of $S$ represents all pairs of states such that when execution of $S$ begins in the state $\sigma$, the execution of $S$ terminates with a jump to the label $L$ in the state $r$.

We can define the soundness of our axiomatic system as follows:

$$\vdash \{p\} S \{q\} \Sigma \Rightarrow \forall \sigma, r[(\sigma \in p \land (\sigma, r) \in S^1) \Rightarrow r \in q]$$

$$\land \forall L[(\sigma \in p \land (\sigma, L : r) \in S^2) \Rightarrow [(L : p') \in \Sigma \land r \in p']]$$

Next, we define the strongest postcondition $SP$ and strongest exit-condition $SE$.

$$SP(p, S) = \{r | \exists \sigma \in p \land (\sigma, r) \in S^1\}$$

$$SE(p, S) = \{L : r | \exists \sigma \in p \land (\sigma, L : r) \in S^2\}$$

In order to demonstrate the soundness of the given axioms, it is sufficient to prove

$$SP(p, S) \subseteq q \quad \text{and} \quad SE(p, S) \subseteq \Sigma$$

Lastly, to demonstrate the completeness of the given axioms, we must prove

$$\vdash \{p\} S \{SP(p, S)\} SE(p, S)$$

Even though $SP$ and $SE$ represent sets, when used in this context they will refer to the assertion and the set of label-assertion pairs, respectively.

VII.3.1 $\text{SKIP}$

Operational definition: $\langle \{(\sigma, \sigma) | \sigma \in \Omega\}, \Phi \rangle$
If the statement $\text{SKIP}$ is executed when the system is in state $\sigma$, the system will remain in the state $\sigma$. Suppose $\{p\} S \{q\} \Sigma$, then from the axiom for $\text{SKIP}$, $p \Rightarrow q$ and $\Phi \leftarrow \Sigma$. Hence,

$$SP(p, S) = \{ \sigma \mid \sigma \in p \}$$

$$SP(p, S) \subseteq p$$

The strongest postcondition will contain the set of all states that satisfy the precondition $p$. Therefore, it is clear that the set $SP$ is contained in the set of states that satisfy $p$.

$$SE(p, S) = \Phi$$

Using the axiom for $\text{SKIP}$, we can prove:

$$\vdash \{p\} S \{SP(p, S)\} SE(p, S)$$

VII.3.2 Assignment

Operational definition: $\{(\sigma, \sigma') \mid \sigma' = \sigma[x \leftarrow e], \Phi\}$

If the assignment statement is executed when the system is in state $\sigma$, the system will be in state $\sigma'$ when the execution of the assignment terminates which is the original state with the assignment of the expression $e$ to the variable $x$. Suppose $\{p\} S \{q\} \Sigma$ then for the axiom for the assignment, $\Phi \leftarrow \Sigma$.

$$SP(p, S) = \{ \sigma' \mid \exists \sigma \mid \sigma \in p \land \sigma' = \sigma[x \leftarrow e] \}$$

Now,

$$q^s = \{ \sigma \mid \sigma[x \leftarrow e] \in q \}$$
Hence,

\[ SP(p, S)_f^* = \{ \sigma \mid \sigma[x \leftarrow c] \in SP(p, S) \} \]

Suppose \( r \in p \) then by the definition of \( SP(p, S) \),

\[ r[x \leftarrow c] \in SP(p, S) \]

Hence, \( r \in SP(p, S)_f^* \) and therefore, \( p \Rightarrow SP(p, S)_f^* \)

Therefore, it is clear that the set \( SP \) contains the set of states that satisfy \( p \).

\[ SE(p, S) = \emptyset \]

Using the axiom for assignment we can prove,

\[ \vdash \{ p \} \subseteq \{ SP(p, S) \} \subseteq SE(p, S) \]

**VII.3.3 Unconditional \textit{EXIT}**

**Operational definition**: \( \langle \emptyset, \{(\sigma, L : \sigma) \mid \sigma \in \Omega \} \rangle \)

If the statement \textit{EXIT} \( L \) is executed when the system is in state \( \sigma \) then a jump to the label \( L \) must occur and the system will remain in state \( \sigma \). Suppose \( \{ p \} \subseteq \{ q \} \subseteq \Sigma \), then from the axiom for the unconditional \textit{EXIT}, \( \text{false} \Rightarrow q \) and \( \{(L : p)\} \Rightarrow \Sigma \).

Hence,

\[ SP(p, S) = \emptyset \]

\[ SE(p, S) = \{ L : \sigma \mid \sigma \in p \} \]

\[ SE(p, S) = \{ L : p \} \]
The strongest exit-condition will contain the set of all labeled states that satisfy the precondition $p$. It is clear that the set $SE$ is this set. Using the axiom for the unconditional EXIT statement, we can prove

$$\vdash \{p\} S \{SP(p,S)\} SE(p,S)$$

VII.3.4 Conditional EXIT

Operational definition:

$$\{((\sigma, r) \mid r = \sigma \land r \in \neg b), ((\sigma, L : r) \mid r = \sigma \land r \in b)\}$$

If the statement $EXIT L$ WHEN $b$ is executed when the system is in state $\sigma$, the system will remain in that state whether a jump occurs or not. Suppose $\{p\} S \{q\} \Sigma$, then from the axiom for the conditional EXIT, $p \land \neg b \Rightarrow q$ and $\{(L : p \land b)\} \rightarrow \Sigma$.

$$SP(p,S)=\{r \mid \exists \sigma [\sigma \in p \land r = \sigma \land r \in \neg b]\} \quad SP(p,S) \subseteq [p \land \neg b]$$

$$SE(p,S)=\{L : r \mid \exists \sigma [\sigma \in p \land r = \sigma \land r \in b]\} \quad SE(p,S) \subseteq (L : p \land b)$$

If the conditional EXIT statement is executed when the system is in state $\sigma$ and if $b \equiv \mathit{false}$, then no jump will occur and the system will remain in the same state. The strongest postcondition will contain the set of all states that satisfy $p \land \neg b$. If $b \equiv \mathit{true}$, then a jump to the label $L$ must occur and the system will remain in state $\sigma$. The strongest exit-condition will contain the set of all labeled states that satisfy $p \land b$. Using the axiom for the conditional EXIT statement, we can prove
VII.3.5 Sequential Composition

Operational definition: Consider the statement \( S = S_1; S_2 \), then \( \langle S^1, S^2 \rangle \) is defined as follows:

\[
\langle \{r | \exists \sigma \exists \sigma' \ (\sigma, \sigma') \in S_1^1 \land (\sigma', r) \in S_2^1 \} \rangle,
\]

\[
\langle \{r : L | (r, L : r) \in S_1^2 \lor \exists \sigma' \ (\sigma, \sigma') \in S_1^1 \land (\sigma', L : r) \in S_2^1 \} \rangle.
\]

If neither \( S_1 \) nor \( S_2 \) contains an \textit{EXIT} \( L \) statement and the system begins execution of \( S_1; S_2 \) in state \( \sigma \), then when the execution of \( S_1; S_2 \) terminates, the system will be in state \( r \). If \( S_1 \) contains an \textit{EXIT} \( L \) statement, then a jump must occur to the label \( L \) and the system will be in the corresponding state \( r \). On the other hand, if only \( S_2 \) contains an \textit{EXIT} \( L \), then a jump will occur to the label \( L \) after the execution of \( S_1 \) and the system will be in the corresponding state \( r \).

Suppose we can prove \( \{p \} S_1; S_2 \{q \} \Sigma \), then there exists a \( q, \Sigma_1, \) and \( \Sigma_2 \) such that \( SP(p, S_1) \subseteq q, SP(q, S_2) \subseteq r, SE(p, S_1) \subseteq \Sigma_1, SE(q, S_2) \subseteq \Sigma_2, \) and \( \Sigma = \Sigma_1 \cup \Sigma_2. \)

\[
SP(p, S) = \{r | \exists \sigma \ (\sigma \in p \land \exists \sigma' \ (\sigma, \sigma') \in S_1^1 \land (\sigma', r) \in S_2^1)\}
\]

Since \( SP(SP(p, S_1), S_2) \subseteq r \) then \( SP(p, S) \subseteq r \)

\[
SE(p, S) = \{L : r | \exists \sigma \ (\sigma \in p \land (\sigma, L : r) \in S_1^2 \}
\]

\[
\lor \exists \sigma' \ ((\sigma, \sigma') \in S_1^1 \land (\sigma', L : r) \in S_2^1)\}
\]

\[
SE(p, S_1) \cup SE(SP(p, S_1), S_2) \subseteq \Sigma_1 \cup \Sigma_2 \Rightarrow SE(p, S) \subseteq \Sigma_1 \cup \Sigma_2.
The strongest postcondition of \( S_1; S_2 \) is the usual one. The strongest exit-condition of \( S_1; S_2 \) will contain both the strongest exit-condition of \( S_1 \), since a jump could occur in \( S_1 \); and the strongest exit-condition of \( S_2 \), since a jump could occur in \( S_2 \) given that a jump did not occur during the execution of \( S_1 \). The definition of \( \Sigma_1 \cup \Sigma_2 \) requires that the new exit-condition will contain at least the elements of the two previously mentioned sets. Using the axiom for sequential composition,

\[
\text{If } q \equiv \text{SP}(p, S_1) \text{ then } \text{SP}(q, S_2) \Rightarrow \text{SP}(p, S)
\]

\[
\text{If } \Sigma_1 \equiv \text{SE}(p, S_1) \text{ and } \Sigma_2 \equiv \text{SE}(q, S_2) \text{ then } \Sigma_1 \cup \Sigma_2 \subseteq \text{SE}(p, S)
\]

Therefore, \( \vdash \{p\} S \{\text{SP}(p, S)\} \text{SE}(p, S) \)

VII.3.6 Alternation

Operational definition: Consider the statement \( S; IF \ b \ THEN \ S_1 \ ELSE \ S_2 \). \( \Sigma_1 \cup \Sigma_2 \), then \( (S^1, S^2) \) is defined as follows:

\[
\{(\sigma, r) \mid (\sigma, r) \in S^1 \land \sigma \in b \lor (\sigma, r) \in S^2 \land \sigma \in \neg b\},
\]

\[
\{(\sigma, L:r) \mid (\sigma, L:r) \in S^1 \land \sigma \in b \lor (\sigma, L:r) \in S^2 \land \sigma \in \neg b\}
\]

If the alternation statement is executed when the system is in state \( \sigma \) and neither \( S_1 \) nor \( S_2 \) contains an \text{EXIT L} statement, then when the execution of the alternation statement terminates, the system will be in state \( r \). If either \( S_1 \) or \( S_2 \) contains an \text{EXIT L} statement, then a jump must occur to the label \( L \) and the system will be in the corresponding state \( r \).

\[
\text{SP}(p, S) = \{r \mid \exists \sigma \ (\sigma \in p \land ((\sigma, r) \in S^1 \lor (\sigma, r) \in S^2))\}
\]
Since \(SP(p, S_1) \subseteq q\) and \(SP(p, S_2) \subseteq q\)

Therefore, \(SP(p, S) \subseteq q\)

\[
SE(p, S) = \{L:r \mid \exists \sigma [\sigma \in p \land ((\sigma, L:r) \in S_1^2 \lor (\sigma, L:r) \in S_2^2)]\}
\]

\[
SE(p, S_1) \cup SE(p, S_2) \subseteq \Sigma_1 \cup \Sigma_2 \Rightarrow SE(p, S) \subseteq \Sigma_1 \cup \Sigma_2
\]

The strongest postcondition of the alternation statement is the usual one. The strongest exit-condition will contain both the strongest exit-condition of \(S_1\) and \(S_2\). The definition of \(\Sigma_1 \cup \Sigma_2\) requires that the new exit-condition will at least contain the elements of the two previously mentioned sets. Using the axiom for alternation,

\[
\vdash \{p\} S \{SP(p, S)\} SE(p, S)
\]

VII.3.7 Rule of Consequence

Operational definition:

\[
\{((\sigma, r) \mid (\sigma, r) \in S^1_1), ((\sigma, L:r) \mid (\sigma, L:r) \in S^2_2)\}
\]

Using the Rule of Consequence, we are given that

\[
SP(p', S) \subseteq q'
\]

\[
SE(p', S) \subseteq \Sigma_1
\]

If \(p \Rightarrow p'\) then \(SP(p, S) \subseteq q'\)

And

If \(q' \Rightarrow q\) then \(SP(p', S) \subseteq q\)

Lastly,

If \(\Sigma_1 \rightarrow \Sigma_2\) then from the definition of \(\rightarrow\), \(SE(p', S) \subseteq \Sigma_2\)
VII.3.8 LOOP

The following definitions will be used in the operational definition of the loop construct [3].

\[(p \cdot S)^1 \overset{def}{=} \{(\sigma, r) | \exists n \exists \sigma_0, \ldots, \sigma_n. \sigma = \sigma_0 \land \sigma_n = r \land \forall i < n. (\sigma_i \in p \land (\sigma_i, \sigma_{i+1}) \in S^1)\}\]

\[(p \cdot S)^2 \overset{def}{=} \{(\sigma, L : r) | \exists \sigma' |(\sigma, \sigma') \in (p \cdot S)^1 \land (\sigma', L : r) \in S^2\}\]

Operational definition: Consider S, a loop of the kind L:LOOP S' END L; then \((S^1, S^2)\) is defined as follows:

\[\begin{align*}
\{(\sigma, r) | (\sigma, L : r) \in ((\text{true}) \cdot \cdot S')^2\}, \\
\{(\sigma, L' : r) | (\sigma, L' : r) \in ((\text{true}) \cdot \cdot S')^2 \land L' \neq L\}
\end{align*}\]

Normal termination of a loop can only occur by means of an EXIT L statement that names the loop. Termination can also occur by means of an EXIT L statement that refers to an outer loop which is reflected in the exit-condition.

\[\text{SP}(p, S) = \{r | \exists \sigma | \sigma \in p \land (\sigma, L : r) \in ((\text{true}) \cdot \cdot S')^2\}\]

Since \((\sigma', L : r) \in (S')^2\) and \((L : r) \in \Sigma\)

then \(\text{SP}(p, S) \subseteq r\)

The strongest postcondition will contain all final states \(r\) that correspond to the label to which the jump was made. Therefore, \(r\) must satisfy the assertion associated with this label. Notice that this postcondition was originally an exit-condition.
\[ \text{SE}(p, S) = \{ L' : r \mid \exists \sigma \ [\sigma \in p \land (\sigma, L' : r) \in ((\text{true}) \ast S') \land L' \neq L] \} \]

Since \( \text{SE}(p, S') \subseteq \Sigma \) and \( \text{SE}(p, S') \setminus \{(L : r)\} = \text{SE}(p, S) \)

then \( \text{SE}(p, S) = \Sigma \setminus \{(L : r)\} \)

The strongest exit-condition of the loop body will remain unmodified, with the exception of the removal of the label-assertion pair that was used as the postcondition of this loop.

To demonstrate the completeness of this axiom, we require the use of the loop invariant \( q \).

\[ q \overset{df}{=} \{ r \mid \exists \sigma \ [\sigma \in p \land (\sigma, r) \in ((\text{true}) \ast S') \} \} \]

It is clear that this is the proper loop invariant since \( p \Rightarrow q \) and \( \{q\} S' \{q\} \Sigma \).

Therefore, it is known that \( \text{SE}(q, S') \subseteq \Sigma \).

\[ \text{SE}(q, S') \setminus \{(L : r)\} = \text{SE}(p, S) \text{ then } \text{SE}(p, S) = \Sigma \setminus \{(L : r)\} \]

This loop invariant will establish

\[ \vdash \{p\} S \{SP(p, S)\} \text{ SE}(p, S) \]

**VII.3.9 WHILE**

Operational definition: Consider \( S \), a loop of the kind \( L:\text{WHILE} \ b \ \text{DO} \ S' \ \text{ENDL} \); then \((S^1, S^2)\) is defined as follows:

\[ \langle \{ (\sigma, r) \mid (\sigma, r) \in (b \ast S') \land r \in [\neg b] \lor (\sigma, L : r) \in (b \ast S') \rangle, \]

\[ \{ (\sigma, L' : r) \mid (\sigma, L' : r) \in (b \ast S') \land L' \neq L \} \rangle \]
Normal termination of the WHILE loop can occur either by \( b = \text{false} \) or by means of an \textit{EXIT} \( L \) statement that names the loop. Termination can also occur by means of an \textit{EXIT} \( L \) statement that refers to an outer loop.

\[
\text{SP}(p, S) = \{ r \mid \exists \sigma \ (\sigma \in p \land (\sigma, r) \in (b \bullet S')^1 \land r \in \neg b) \} \\
\text{SP}(p, S) \subseteq [(p \land \neg b) \lor r]
\]

The strongest postcondition will contain all final states \( r \) that either satisfy \( p \land \neg b \) or correspond to the label to which the jump was made. In the latter case, \( r \) must satisfy the assertion associated with this label.

\[
\text{SE}(p, S) = \{ L' : r \mid \exists \sigma \ (\sigma \in p \land (\sigma, L' : r) \in (b \bullet S')^2 \land L' \neq L) \}
\]

Since \( \text{SE}(p \land b, S') \subseteq \Sigma \) and \( \text{SE}(p \land b, S') - \{(L : r)\} = \text{SE}(p, S) \)

then \( \text{SE}(p, S) = \Sigma - \{(L : r)\} \)

The strongest exit-condition of the loop body will remain unmodified, with the exception of the removal of the label-assertion pair that was used in the postcondition of this loop.

To demonstrate the completeness of this axiom, we require the use of the loop invariant \( q \).

\[
q \overset{\text{def}}{=} \{ r \mid \exists \sigma \ (\sigma \in p \land (\sigma, r) \in (b \bullet S')^1) \}
\]

It is clear that this is the proper loop invariant since \( p \Rightarrow q \) and \( \{q \land b\} S' \{q\} \Sigma \). Therefore, it is known that \( \text{SE}(q \land b, S') \subseteq \Sigma \).

\[
\text{SE}(q \land b, S') - \{(L : r)\} = \text{SE}(p, S) \text{ then } \text{SE}(p, S) \subseteq \Sigma - \{(L : r)\}
\]
This loop invariant will establish
\[ \{p\} S \{SP(p,S)\} SE(p,S) \]

VII.4 Operational Model of Concurrency

The "standard" operational model of a concurrent program would essentially consider the entire program at once and specify how to execute one step of the program at a time, interleaving commands from the different concurrent processes. It is rather difficult to prove the soundness and completeness of our type of axiomatics with respect to such a model since an essential aspect of our axiomatics is to consider each process in isolation from the other processes. The solution is to define two operational models: a standard ("global") model that considers the entire program at once; and a non-standard ("local") model that defines the operational semantics of each process in isolation, and then specifies how these operational semantics may be combined to obtain the operational semantics of the entire program.

The next step is to show that the two models are equivalent by showing that for any given program, the states that can be reached are the same in the two models. Lastly, one must show that the axiomatics is sound and complete with respect to the "local" model. This is no harder than proving soundness and completeness of axiomatics of sequential languages since the axiomatics and the local operational model are just two different formalizations of the same way of looking at concurrent programs.

This approach was used in [36] to demonstrate the soundness and completeness of the axiomatics of CSP. In this section we illustrate this method by briefly describing what the local and global models for DP would look like, how they can be shown to be equivalent, and how soundness and completeness of the axiomatics with respect to the local model may be demonstrated. Essentially, the same approach
may be used for the other languages/constructs we have considered.

VII.4.1 Local Operational Model

The local operational model is defined in terms of a relation "→" which will correspond to the execution of one step of a procedure. For convenience, we shall allow the empty procedure $E$; $\mathcal{N}$ will denote the set of natural numbers. We shall use the symbol $\sigma_{i,k}$ to denote the current state of the procedure $Q_k$ defined in the process $P_i$ which will consist of two components: $\Sigma_{i,k}$, the local state of the procedure; and $h_{i,k}$, the interaction sequence of the $k^{th}$ procedure in the $i^{th}$ process. Thus, $(P_{i,k}, \sigma_{i,k}) \rightarrow (P'_{i,k}, \sigma'_{i,k})$ specifies that executing one step of $P_{i,k}$ in the state $\sigma_{i,k}$ can lead to the state $\sigma'_{i,k}$ with $P'_{i,k}$ being the portion of the procedure yet to be executed. Therefore, $(S_{i,k}, \sigma_{i,k}) \rightarrow (S'_{i,k}, \sigma'_{i,k})$ if any one of the following clauses is satisfied:

1) $S_{i,k} \equiv \text{Process } P_i \ldots \ldots \text{Procedure } Q_k(\overline{X} \# \overline{Z}) \text{ Begin } T_{i,k} \text{ end}$
   
   $S'_{i,k} \equiv T_{i,k} \text{ end}$

   $\sigma'_{i,k} \equiv \sigma^*_{i,k}[\overline{X} \leftarrow \overline{X}_o, \overline{R} \leftarrow \overline{R}_o, h_{i,k} = \langle \text{call}, i, k, \overline{X}_o, \overline{R}_o \rangle]$ where $\overline{X}_o, \overline{R}_o \in \mathcal{N}$

   $\sigma^*_{i,k}$ is the initial state of the procedure $Q_k$ defined in the process $P_i$. This initial state will include the initial values of the local variables of the procedure.

2) $S_{i,k} \equiv \text{end}$
   
   $S'_{i,k} \equiv E$

   $\sigma'_{i,k} \equiv \sigma_{i,k}[h_{i,k} \leftarrow h_{i,k} \neg \text{return}, i, k, \overline{Z}(\sigma_{i,k}), \overline{R}(\sigma_{i,k})]$}

3) $S_{i,k} \equiv \text{skip}$
   
   $S'_{i,k} \equiv E$

   $\sigma'_{i,k} = \sigma_{i,k}$
4) \( S_{i,k} \equiv x := e \)

\[ S'_{i,k} \equiv E \quad \sigma'_{i,k} \equiv \sigma_{i,k}[x \leftarrow e] \]

5) \( S_{i,k} \equiv \text{If } b_1 : T_1 | \ldots | b_n : T_n \text{ end, } \models b_r(\sigma_{i,k}) \)

\[ S'_{i,k} \equiv T_r \quad \sigma'_{i,k} = \sigma_{i,k} \]

6) \( S_{i,k} \equiv \text{Do } b_1 : T_1 | \ldots | b_n : T_n \text{ end, } \models b_r(\sigma_{i,k}) \)

\[ S'_{i,k} \equiv T_r ; S_{i,k} \quad \sigma'_{i,k} = \sigma_{i,k} \]

7) \( S_{i,k} \equiv \text{Do } b_1 : T_1 | \ldots | b_n : T_n \text{ end, } \forall r \in n. \models b_r(\sigma_{i,k}) \)

\[ S'_{i,k} \equiv E \quad \sigma'_{i,k} = \sigma_{i,k} \]

8) \( S_{i,k} \equiv \text{Call } P_j, Q_I(\overline{x} \# \overline{z}) \)

\[ S'_{i,k} \equiv E \]

\[ \sigma'_{i,k} \equiv \sigma_{i,k}[\overline{z} \leftarrow \overline{y}, h_{i,k} \leftarrow h_{i,k} \neg (\text{call to } i, k, j, l, \overline{x}(\sigma_{i,k})) \neg (\text{return from } i, k, j, l, \overline{y})] \]

where \( \overline{y} \in \overline{y} \)

9) \( S_{i,k} \equiv \text{When } b_1 : T_1 | \ldots | b_n : T_n \text{ end, } \models b_r(\sigma_{i,k}) \)

\[ S'_{i,k} \equiv T_r \quad \sigma'_{i,k} = \sigma_{i,k} \]

10) \( S_{i,k} \equiv \text{When } b_1 : T_1 | \ldots | b_n : T_n \text{ end, } \forall r \in n. \models b_r(\sigma_{i,k}) \)

\[ S'_{i,k} \equiv S_{i,k} \]

\[ \sigma'_{i,k} \equiv \sigma_{i,k}[\overline{R} \leftarrow \overline{r}, h_{i,k} \leftarrow h_{i,k} \neg (\text{suspend } i, k, \overline{R}(\sigma_{i,k})) \neg (\text{resume } i, k, \overline{r})] \]

where \( \overline{r} \in \overline{r} \)
11) $S_{i,k} \equiv T_{i,k}; S$ where $(T_{i,k}, \sigma_{i,k}) \rightarrow (T'_{i,k}, \sigma'_{i,k})$ and either
\[
[T'_{i,k} \equiv E, S'_{i,k} \equiv S] \quad \text{or} \quad [T'_{i,k} \not\equiv E, S'_{i,k} \equiv T'_{i,k}; S]
\]

Next, we must examine the process initialization section. The statements numbered 3 through 11 that were defined previously for procedures can also be contained within the initialization section. When referring to these statements, given that they are associated with the initialization section, we will assign the value 0 to $k$ since we are referring to the 0th procedure of the process $P_i$. In order to continue the definition of the relation “$\rightarrow$”, we need to examine the statements that are indigenous to the initialization section.

12) $S_{i,0} \equiv \text{Begin } T \text{ end}$

\[S'_{i,0} \equiv T \text{ end} \quad \sigma'_{i,0} = \sigma_{i,0}\]

13) $S_{i,0} \equiv \text{end}$

\[S'_{i,0} \equiv E \quad \sigma'_{i,0} = \sigma_{i,0}[h_{i,0} \leftarrow h_{i,0} \sim \langle \text{end}; i, 0, R(\sigma_{i,0}) \rangle]\]

14) $S_{i,0} \equiv \text{Call } P_i.Q_k(\overline{X} \# \overline{Z})$

\[S'_{i,0} \equiv E \quad \sigma'_{i,0} = \sigma_{i,0}[\overline{Z} \leftarrow \overline{V}, \overline{R} \leftarrow \overline{T},
\]
\[h_{i,0} \leftarrow h_{i,0} \sim \langle \text{call.to}; i, 0, l, k, \overline{X}(\sigma_{i,0}), R(\sigma_{i,0}) \rangle \sim \langle \text{return.from}; i, 0, i, k, \overline{V}, \overline{T} \rangle]\]
\[\text{where } \overline{V}, \overline{T} \in \overline{N}\]
15) $S_{i,o} \equiv \text{Cycle } b_1:T_1 \mid \ldots \mid b_n:T_n \text{ end}, \quad \models b_r(\sigma_{i,o})$

$S'_{i,o} \equiv T_r; S_{i,o} \quad \sigma'_{i,o} = \sigma_{i,o}$

16) $S_{i,o} \equiv \text{Cycle } b_1:T_1 \mid \ldots \mid b_n:T_n \text{ end}, \quad \forall r \in n. \quad \models b_r(\sigma_{i,o})$

$S'_{i,o} \equiv S_{i,o} \quad \sigma'_{i,o} \equiv \sigma_{i,o}[R \leftarrow T, h_{i,o} \leftarrow h_{i,o} \neg (\text{suspend},i,0,R(\sigma_{i,o})) \neg (\text{resume},i,0,T)]$

where $T \in \mathcal{N}$

Our axiomatics only allows us to prove invariants (not postconditions) of DP processes/programs. Therefore, we must define the operationally valid invariants for a given program. This will allow us to argue the soundness of our axiomatics by showing that every invariant that is axiomatically provable is also operationally valid. We can argue the completeness of the axiomatics by showing the converse.

Definition:

$$(S_{i,k}, \sigma_{i,k}^*) \rightarrow (S_{i,k}^*, \sigma_{i,k}^*)$$

is a sequence of pairs $(S_{i,k}, \sigma_{i,k}^*)$, $\ldots$, $(S_{i,k}^*, \sigma_{i,k}^*)$ that is a local computation sequence for the procedure $Q_{i,k}$ if $S_{i,k}^* \equiv P_{i,k}$ and for each $v$ such that $1 \leq v < u$

$$(S_{i,k}^v, \sigma_{i,k}^v) \rightarrow (S_{i,k}^{v+1}, \sigma_{i,k}^{v+1})$$

This definition captures the notion of a single execution of the procedure $Q_{i,k}$. It is the second component in $\sigma_{i,k}^v$ that specifies the interaction sequence $h_{i,k}$ that will satisfy the strongest invariant of a single execution of the procedure. We can determine a set of these local computation sequences that will represent the executions of the many incarnations of the procedure $Q_{i,k}$. If we merge the sequences
in this set so that the predicate $CHECK$ is satisfied then we will obtain a new set of local computation sequences $(P_{i,k}^i \leadsto P_{i,k})$ whose corresponding interaction sequences will satisfy the strongest invariant of the procedure. We can continue this methodology by merging the sets of local computation sequences of the procedures and initialization section in such a way that the predicate $CONSIST$ is satisfied. This new set of local computation sequences for the process $(C_i \leadsto C_i')$ which will determine the strongest invariant for the process. Lastly, by merging the process local computation sequences so that $COMPAT$ is satisfied, we obtain the set of local computation sequences for the program $(C' \leadsto C)$. 

VII.4.2 Global Operational Model

The global operational model is defined by specifying the relation "$\Rightarrow$" between configurations of a program where "$\Rightarrow$" denotes the execution of one step of a DP program. A program $P$ will be an $n$-tuple, $(C_1,\ldots,C_n)$, with $C_i$ representing the configuration of the process $P_i$. A configuration is defined as follows:

$$C_i: ((Q_o,Q_1,Q_2,\ldots,Q_m),(V_i,h_i),((PC_o,L_o,h_{i,o}^o,0,0),$$

$$(PC_1,L_1,h_{i,1}^i,1,INCARN(i,1)),\ldots,(PC_k,L_k,h_{i,k}^i,m,INCARN(i,m))),$"

$$(h_{i,1},h_{i,2},\ldots,h_{i,m}))$$

The first component of the configuration of the process $P_i$ specifies the corresponding initialization section $(Q_o)$ and procedures $(Q_1,\ldots,Q_m)$ defined in $P_i$. The second component contains the current values of the process variables $(V_i)$ along with the interaction sequence $(h_i)$ associated with this process. The third component is a $(k+1)$-tuple consisting of elements that represent the program counter $(PC_i)$, current values of the local variables $(L_j)$ and the interaction sequence $(h_{i,j}^i)$ associated with each incarnation of the procedures and initialization
section of the process \( P_i \). Each tuple also includes the corresponding procedure
and incarnation number. The last component consists of the procedure interaction
sequences \( (h_{i,1}, \ldots, h_{i,m}) \). Notice that \( h'_{i,e} \equiv h_{i,e} \) since there exists only one incarna-
tion of the initialization section and that \( MERGE(h'_{i,k}, h'_{i,k}, \ldots, h'_{i,k}) \equiv h_{i} \) since each
procedure sequence associated with a single incarnation of that procedure will be
contained within the procedure interaction sequence. Therefore, the initial config-
uration \( (C'_1, \ldots, C'_n) \) of a DP program is defined as follows:

\[
\forall j < n. \quad C'_j = \langle (Q_{o_j}, Q_{1_j}, Q_{2_j}, \ldots, Q_{m_j}), (V_{j,e}), \{(PC_{o_j}, L_o, e, 0, 0)\}, (e, \ldots, e) \rangle
\]

where \( PC_{o_j} \) points to the beginning of the initialization section of the process \( P_i \).
Therefore, \( (C_1, \ldots, C_n) \Rightarrow (C'_1, \ldots, C'_n) \) specifies that executing one step of the pro-
gram \( P \) in the configuration \( (C_1, \ldots, C_n) \) can lead to the configuration \( (C'_1, \ldots, C'_n) \).

Let \( PC_j(P_i) \) determine which statement the \( j^{th} \) program counter of the \( i^{th} \) process
is pointing, \( ACTIVE(PC_j(P_i)) \) determine whether the statement \( PC_j(P_i) \) is cur-
rently executing and \( LOC(i, k, S) \) determine the location of the statement \( S \) in the
procedure \( Q_k \) located in the process \( P_i \).

We now turn to the task of defining \( "\Rightarrow" \) by examining what changes could
occur to \( C_i \) when executing one step of a procedure in the program. In the following
discussion, we shall omit clauses of the kind \( "C'_j \equiv C_j" \) for all \( j \neq i \); thus \( C'_j \equiv C_j \).

1) \( \exists j < k. \ ACTIVE(PC_j(P_i)) \land PC_j(P_i) = \text{Skip} \Rightarrow \)

\[
C'_i \leftarrow C_i[PC_j(P_i) \leftarrow PC_j(P_i)+1]
\]

2) \( \exists j < k. \ ACTIVE(PC_j(P_i)) \land PC_j(P_i) = x := e \Rightarrow \)

\[
C'_i \leftarrow C_i[PC_j(P_i) \leftarrow PC_j(P_i)+1, \ V_i \leftarrow V_i[x := e]]
\]

where \( x \in V_i \)
\[ C'_i \leftarrow C_i[PC_j(P_i) \leftarrow PC_j(P_i)+1, \; L_j \leftarrow L_j[x \leftarrow e]] \]

where \( x \in L_j \)

3) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{If } b_1; T_1 | \ldots | b_n; T_n \; \text{end} \land \]

\[ \models b_r(C_i) \Rightarrow C'_i \leftarrow C_i[PC_j(P_i) \leftarrow \text{LOC}(i,j,T_r)] \]

4) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{Do } b_1; T_1 | \ldots | b_n; T_n \; \text{end} \land \]

\[ \models b_r(C_i) \Rightarrow C'_i \leftarrow C_i[PC_j(P_i) \leftarrow \text{LOC}(i,j,T_r)] \]

5) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{Do } b_1; T_1 | \ldots | b_n; T_n \; \text{end} \land \]

\[ \forall r \in n. \; \models b_r(C_i) \Rightarrow C'_i \leftarrow C_i[PC_j(P_i) \leftarrow \text{LOC}(i,j,\text{end})] \]

6) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{Call } P_m.Q_n(\overline{X} \# \overline{Y}) \Rightarrow \]

\[ \neg ACTIVE(\text{PC}_j(P_i)) \land \]

\[ C'_i \leftarrow C_i[PC_j(P_i) \leftarrow PC_j(P_i)+1, \; h_i \leftarrow h_i \uparrow \text{ELEMENT}, \]

\[ h_i \cdot \text{PROC}(PC_j) \leftarrow h_i \cdot \text{PROC}(PC_j) \uparrow \text{ELEMENT}, \]

\[ h'_i \cdot \text{PROC}(PC_j) \leftarrow h'_i \cdot \text{PROC}(PC_j) \uparrow \text{ELEMENT}; \]

where \( \text{ELEMENT} = \text{(call.to,i,PROC}(PC_j),m,n,\overline{X}(C_i)) \)

7) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{When } b_1; T_1 | \ldots | b_n; T_n \; \text{end} \land \]

\[ \models b_r(C_i) \Rightarrow C'_i \leftarrow C_i[PC_j(P_i) \leftarrow \text{LOC}(i,j,T_r)] \]

8) \( \exists j < k. \; ACTIVE(\text{PC}_j(P_i)) \land PC_j(P_i) = \text{When } b_1; T_1 | \ldots | b_n; T_n \; \text{end} \land \)

\[ \forall r \in n. \; \not\models b_r(C_i) \Rightarrow \neg ACTIVE(\text{PC}_j(P_i)) \land \]

\[ C'_i \leftarrow C_i[h_i \cdot \text{PROC}(PC_j) \leftarrow h_i \cdot \text{PROC}(PC_j) \uparrow \text{ELEMENT}, \]
where \( \text{ELEMENT} = \langle \text{suspend}, i, \text{PROC}(P_{C_j}), V_i(C_i) \rangle \)

9) \( \forall i < k. \neg \text{ACTIVE}(P_{C_i}(P_i)) \Rightarrow \)

\[
\begin{align*}
\exists j < k. & \quad \text{PC}_j(P_i) = \text{When} \ b_1:T_1 \ | \ldots | b_n:T_n \ \text{end} \land \ b_r(C_i) \Rightarrow \\
& \text{ACTIVE}(P_{C_i}(P_i)) \land \\
& C'_i \leftarrow C_i \langle h_{i,\text{PROC}(P_{C_j})} \leftarrow h_{i,\text{PROC}(P_{C_j})}^{\text{ELEMENT}}, \\
& \quad h_{i,\text{PROC}(P_{C_j})} \leftarrow h_{i,\text{PROC}(P_{C_j})}^{\text{ELEMENT}} \rangle \rangle \\
\end{align*}
\]

where \( \text{ELEMENT} = \langle \text{resume}, i, \text{PROC}(P_{C_j}), V_i(C_i) \rangle \)

or

\[
\begin{align*}
\exists j < k. & \quad \text{PC}_j(P_i) = \text{Cycle} \ b_1:T_1 \ | \ldots | b_n:T_n \ \text{end} \land \ b_r(C_i) \Rightarrow \\
& \text{ACTIVE}(P_{C_i}(P_i)) \land \\
& C'_i \leftarrow C_i \langle h_{i,0} \leftarrow h_{i,0}^{\langle \text{resume}, i, 0, V_i(C_i) \rangle} \rangle \\
\end{align*}
\]

or

\[
\begin{align*}
\exists j < k. & \quad \text{PC}_j(P_i) = \text{Procedure} \ Q_i(\overline{X} \# \overline{Z}) \Rightarrow \text{ACTIVE}(P_{C_j}(P_i)) \\
\end{align*}
\]

10) \( \text{PROJECT}(C_i, Q_{i,j}) \equiv Q'_{i,j}; Q''_{i,j} \) where \( Q'_{i,j} \neq E \land \)

\( \exists i. \text{ACTIVE}(P_{C_i}(P_i)) \in Q'_{i,j} \land \)

\[
\langle C_1, \ldots, C_i, \ldots, C_n \rangle \Rightarrow \langle C'_1, \ldots, C'_i, \ldots, C'_n \rangle \Rightarrow \\
|\text{PROJECT}(C'_i, Q'_{i,j}) \neq E, \\
\text{PROJECT}(C'_i, Q'_{i,j}) \equiv \text{PROJECT}(C'_i, Q'_{i,j}); Q''_{i,j}| \\
\]

or

\[
\begin{align*}
|\text{PROJECT}(C'_i, Q'_{i,j}) \equiv E, \text{PROJECT}(C'_i, Q'_{i,j}) \equiv Q''_{i,j}| \\
\end{align*}
\]
11) \( \exists j < k. \text{ACTIVE}(PC_j(P_i)) \land PC_j(P_i) = \text{Procedure } Q_i(\overline{X \# \overline{Z}}) \Rightarrow \)

\[
C'_i \leftarrow C_i \mid PC_j(P_i) \leftarrow PC_j(P_i) + 1, h_i \leftarrow h_i \cdot \text{ELEMENT},
\]

\[
h_{i,t} \leftarrow h_{i,t} \cdot \text{ELEMENT}, h'_{i,t} \leftarrow h'_{i,t} \cdot \text{ELEMENT}]
\]

where \( \text{ELEMENT} = (\text{call}_{P_i,l_{C_i}V_i(C_i),\text{INCARN}(i,l)}) \)

\[
\forall m \prec n. \forall j' < k. \neg \text{ACTIVE}(PC_j(P_m)) \land PC_j(P_i) = \text{Call } P_i.Q_i(\overline{X \# \overline{Z}}) \land
\]

\[
ELEMB(h_m,\text{PROC}(PC_j(P_m)),1) =
\]

\[
\text{(call}_{P_i,l_{C_i}V_i(C_i),\text{PROC}(PC_j(P_m)),\text{PROC}(PC_j(P_i)),\overline{X}(C_m)) \land
\]

\[
L_j \cap L_{j'} = \overline{X} \Rightarrow
\]

\[
\text{ADD}(ELEMB(h_m,\text{PROC}(PC_j(P_m))),\text{INCARN}(i,l))]
\]

12) \( \exists j < k. \text{ACTIVE}(PC_j(P_i)) \land PC_j(P_i) = \text{end} \Rightarrow \)

\[
C'_i \leftarrow C_i \mid h_i \leftarrow h_i \cdot \text{ELEMENT},
\]

\[
h_{i,\text{PROC}(PC_j(P_i))} \leftarrow h_{i,\text{PROC}(PC_j(P_i))} \cdot \text{ELEMENT},
\]

\[
h'_{i,\text{PROC}(PC_j(P_i))} \leftarrow h'_{i,\text{PROC}(PC_j(P_i))} \cdot \text{ELEMENT},
\]

\[
PC_j(P_i) \leftarrow \text{LOC}(i,\text{PROC}(PC_j(P_i)),\text{PROC})]
\]

where \( \text{ELEMENT} =
\]

\[
(\text{return}_{i,\text{PROC}(PC_j(P_i)),L_i(C_i),V_i(C_i),\text{INCARN}(i,\text{PROC}(PC_j(P_i))))}
\]

\[
\neg \text{ACTIVE}(PC_o(P_i)) \land PC_o(P_i) = \text{Call } P_i.Q_{PROC}(PC_j(P_i))(\overline{X \# \overline{Z}}) \Rightarrow \)

\[
C'_i \leftarrow C'_i \mid h_{i,o} \leftarrow h_{i,o} \cdot \text{return}_{i,0,i,\text{PROC}(PC_j(P_i)),L_i(C_i),V_i(C_i)),
\]

\[
PC_o(P_i) \leftarrow PC_o(P_i) + 1]
\]

\[
\forall m \prec n. j' < k. \neg \text{ACTIVE}(PC_j(P_m)) \land \)

\[
PC_j(P_m) = \text{Call } P_i.Q_{PROC}(PC_j(P_i))(\overline{X \# \overline{Z}}) \land
\]

\[
\text{INCARN}(i,\text{PROC}(PC_j(P_i))) = \text{COPYB}(h_m,\text{PROC}(PC_j(P_m)),1) \Rightarrow
\]

\[
\text{ACTIVE}(PC_j(P_m)) \land
\]
In order to continue the definition of the relation \( \Rightarrow \), we need to examine the process initialization section. The statements numbered 1 through 10 that were defined previously for procedures can also be contained within the initialization section.

13) \( ACTIVE(PC_0(P_i)) \land PC_0(P_i)=\text{Begin} \Rightarrow C'_i \leftarrow C_i[PC_0(P_i) \leftarrow PC_0(P_i)+1] \)

14) \( ACTIVE(PC_0(P_i)) \land PC_0=\text{end} \Rightarrow C'_i \leftarrow C_i[h_{i,o} \leftarrow h_{i,o}^\text{ELEMENT}, h_i \leftarrow h_i^\text{ELEMENT}] \)

\[ \text{where ELEMENT} = \langle \text{end}, i, 0, V_i(C_i) \rangle \]

15) \( ACTIVE(PC_0(P_i)) \land PC_0(PC_0(P_i))=\text{Call } P_i, Q_k(X#Z) \Rightarrow C'_i \leftarrow C_i[h_{i,o}^\text{ELEMENT}(\text{call.to.i,0,i,k,X}(C_i), V_i(C_i))] \)

10) \( ACTIVE(PC_0(P_i)) \land PC_0(P_i)=\text{CYCLE } b_1:T_1 | \ldots | b_n:T_n \text{ end} \land \Rightarrow C'_i \leftarrow C_i[PC_0(P_i) \leftarrow LOC(i,0,T_r)] \)

17) \( ACTIVE(PC_0(P_i)) \land PC_0(P_i)=\text{Cycle } b_1:T_1 | \ldots | b_n:T_n \text{ end} \land \forall r \in n. \neg \models b_r(C_i) \Rightarrow C'_i \leftarrow C_i[h_{i,o} \leftarrow h_{i,o}^\text{suspend.i,0,V_i(C_i)}] \)
VII.4.3 Soundness and Completeness of Axiomatics

Definition of Soundness:

If \( (r) [P_1 \parallel \ldots \parallel P_n] \) is provable and

\[
C^n; (C'_1, \ldots, C'_n) \Rightarrow C; (C_1, \ldots, C_n)
\]

Then the process interaction sequences \((h_1, \ldots, h_n)\) given in the configuration \((C_1, \ldots, C_n)\) will satisfy \(r\).

To verify soundness one must first demonstrate that the local operational model corresponds to the axiomatics, i.e. show that if \((C_1, \ldots, C_n) \Rightarrow (C_1, \ldots, C_n)\) then the interaction sequences \(h_1, \ldots, h_n\) in the configuration \((C_1, \ldots, C_n)\) will satisfy \(r\). Next, one must examine a given global computation sequence

\[
(C_1, \ldots, C'_n), \ldots, (C'_1, \ldots, C'_n)
\]

for \([P_1 \parallel \ldots \parallel P_n]\) where

\[
(C'_v, \ldots, C'_n) \Rightarrow (C'^{v+1}_1, \ldots, C'^{v+1}_n), \ \forall v, 1 \leq v < u
\]

This global computation sequence can be used to construct local computation sequences for \(P_1, \ldots, P_n\). Notice that the given local model of concurrency is defined in terms of the procedure interaction sequences. The construction of the local computation sequence proceeds as follows: replace all configurations \(C_v\) by \(C'_v\); starting with the last configuration \(C'_u\) thus obtained, delete all configurations \(C'_v\) which satisfy the conditions \(C'^{v-1}_i \equiv C'_v (1 < v \leq u)\). The sequence obtained after deleting all such pairs is the required local computation sequence for \(P_i\) and its correspondence with the actual local computation sequence \((C'_i \Rightarrow C_i)\) must be demonstrated. Notice that each actual local computation sequence is given by \((r) [P_1 \parallel \ldots \parallel P_n]\) and each process local computation sequence, \(C'_i \Rightarrow C_i\), will specify
an $h_i$ that satisfies $r$. The fact that $COMPAT(h_1^n, \ldots, h_n^n) \equiv \text{true}$ follows immediately from the definition of "→". These two results together establish the soundness of the axiomatics with respect to the global model.

**Definition of Completeness:**

$\exists [P_1 \parallel \ldots \parallel P_n]$ is provable where $r$ is the assertion corresponding to the set

\[
((h_1, \ldots, h_n) \mid \exists (C_1, \ldots, C_n). ((C_1, \ldots, C_n) \Rightarrow (C_1, \ldots, C_n))
\]

and $(h_1, \ldots, h_n)$ is the value of the process sequences in $(C_1, \ldots, C_n)]\}

Completeness is also proved in two steps. We must first prove completeness with respect to the local model. To do this, we define the strongest invariant for a procedure, process and program (in the same way as Apt et al.[3] define the strongest postcondition for sequential programs) and demonstrate that these invariants are provable using the axiomatics. Next, we show that a program computation sequence in the local model is also a computation sequence in the global model for the same program. This demonstrates that the strongest invariant for the program defined using the local model is also the strongest invariant for the same program in the global model. These two results demonstrate the completeness of the axiomatics with respect to the global model.

**VII.5 Summary**

We argued that the task of demonstrating the soundness and completeness of proof systems defined using our approach for any concurrent language is also universal. Briefly, the procedure for proving soundness and completeness proceeds as follows: define a (non-standard) local operational model that is similar in spirit to the axiomatic definition (specify the operational semantics of each process in
isolation and then specify how the semantics of the individual processes may be combined to obtain the operational semantics of the whole program). Next, one must show that the operational semantics defined above is equivalent to a standard (global) operational semantics where all the processes of the program would be considered at the same time, as far as the states that can be reached are concerned. The final step is to demonstrate that the proof system is sound and complete with respect to the local operational model.
CHAPTER VIII

Summary and Conclusions

VIII.1 Summary

Since the early seventies, research concentrated on the development and application of techniques for the verification of sequential programs. Today, the production of distributed computing systems has emphasized the need for similar formal methods to also be employed in the construction and verification of concurrent programs. A formal approach to program verification is essential when working with distributed programs, since human intuition has been demonstrated as unreliable when dealing with several processes which are executing simultaneously and interacting extensively. Moreover, the verification of parallel programs is complicated by the nondeterministic manner in which parallel programs effect each other. Therefore, the correctness of parallel programs depends not only on the functionality of the individual processes but also on the possible interactions among those processes.

The main contribution of this thesis is to develop a modular axiomatic semantics that can be used to define the semantics of various concurrent systems. The approach is modular or compositional since the proofs of the individual processes of a program are completely independent of each other. This allows us to consider an individual process without making assumptions about the behavior of the other processes in the program. Since the semantics of a process is independent of the rest of the program, we have the flexibility to change any one process without causing
a change in the verification of the other processes in the program. Once we obtain the individual process proofs, we only need to combine the postconditions of these processes in order to describe the behavior of the entire program.

The important concepts underlying our modular approach to concurrency are as follows:

1. The semantics of a process of a concurrent program is essentially the behavior of the process as seen by an external observer of the program and this behavior is captured in the sequence valued variable $h_i$.

2. When obtaining the semantics of a process $P_i$ in a program, we make no assumptions about the behavior of the other processes.

We begin our illustration of how our modular approach is applied to concurrent languages by specifying appropriate axioms and rules of inference for CSP. In order to express the externally observable behavior of a process we associate, with each process, a sequence that will record every input and output command that this process executes. Since we consider each process in isolation, the axiom corresponding to the input command requires us to consider all possible values that could be sent to this process. The parallel composition rule allows us to restrict the value received by the input command by the use of the predicate $COMPAT$. $COMPAT$ essentially requires that the communications between the various processes be recorded in the sequences associated with the individual processes in a mutually consistent fashion.

In Chapters IV and V, we deal with the problem of defining a modular axiomatics of two different types of concurrency: Ada and Distributed Processes (DP). Before we could discuss the semantics of an Ada task, we examined the sequential statements contained within a task which include a variety of multiple exit statements; namely, the $EXIT$, $RAISE$, and $TERMINATE$ statements. It is the statements used to raise "exceptions" and the exception handlers that play an
important role in defining the semantics of Ada tasks.

In Chapter V we present how our approach may be used to axiomatize DP which exhibits a more complex form of concurrency than does Ada due to the coroutine structure present in an individual DP process. Thus, we must deal with two types of interactions when considering DP — the one between the processes of a program and the communication between the procedures of an individual DP process. Corresponding to the two levels of concurrency, we have two rules of composition: the first one allows us to obtain the behavior of a process from the behaviors of its procedures; the second one allows us to obtain the behavior of a DP program given the behaviors of its processes.

We believe that the approach of specifying the behavior of an individual process in terms of the sequence of interactions between this process and the other processes, obtaining this behavior by considering the process in isolation from the other processes and then combining the behaviors of the individual processes to obtain the behavior of the entire program is universal and can be used to define the axiomatics of any type of concurrency. Our argument includes a quick summary of [37] that illustrates how the axiomatics of a simple concurrent language with shared variables may be defined using our approach. The languages discussed in this thesis were selected so that our approach would be applied to at least one language from each of the three classes of concurrent languages defined by Andrews & Schneider[1]. We also explain how each of the following constructs could be handled using our approach; FORK and JOIN primitives, nested monitor calls, path expressions, atomic transactions, and asynchronous message passing. The information within the heart of thesis along with this discussion provides ample evidence to support the claim of universality of our approach.

Lastly, we demonstrate the soundness and completeness of our axiomatics by adopting the approach used in [36]. The procedure for proving soundness and
completeness proceeds as follows: define a local operational model that specifies the operational semantics of each process in isolation and then specify how the semantics of the individual processes may be combined to obtain the operational semantics of the whole program. Next, we define a global operational model where all the processes of the program would be considered at the same time. The local operational model is equivalent to the global operational model as far as the states that can be reached are concerned. Using the approach of Apt et al.[3], the final step demonstrates that the proof system is both sound and complete with respect to the local operational model.

VIII.2 Advantages of the Approach

An important aspect of our approach is that in proving a property of one of the processes, say $P_i$, we must consider it in isolation; thus no knowledge of the "expected" behavior of the remaining processes in the program may be used when we are dealing with the proof of $P_i$. The advantage of dealing with the processes in isolation in this fashion is that the property of $P_i$ thus proved will necessarily be valid irrespective of what the remaining processes in the system may do. As a result, our rule for parallel composition is rather simple. Moreover, using our approach, it is easy to informally "see" – rather than axiomatically verify – that the various assertions are valid at the appropriate points (as in the case of sequential programs) since the assertions in each process refer entirely to the variables local to that process and the validity of the assertions in the process $P_i$ depend entirely on what that process does.

By contrast, in the proof systems proposed by Owicki & Gries[32], Apt et al.[4], Gerth et al.[19,20], and Barringer & Mearns[8], the proofs of the individual processes of a program rely on assumptions about the behavior of the remaining processes. These assumption must then be justified in a "proof of cooperation" when the
proofs of the individual processes are tied together. No such proof of cooperation is needed in the approach proposed since no assumptions may be made. The proof of cooperation is often the most difficult part of the proof and the absence of such a proof results in considerably simpler proofs. Also, when using our approach, a change in one of the processes can only affect the proof of that process, whereas such a change using the other approaches may affect the proof of not only the modified process but of all the other processes in the program.

Another important difference between the approach outlined in this thesis and that of the afore mentioned systems[4,8,19,20,32] is that there are no auxiliary variables in our approach. This simplifies our proofs since the introduction of suitable auxiliary variables usually requires considerable ingenuity. It can be argued that our approach allows the program prover to only include that information which is externally observable in the interaction sequences at each level of proof. Our interaction sequence use is structured and limited which alleviates from the program prover the burden of the knowledge of their use.

Given a proof of a program using our approach, it is possible to translate it into a proof using one of the other approaches[4,8,19,20,32] by introducing the interaction sequences as auxiliary variables and using the predicate COMPAT as a global invariant. The converse is not, however, true and in general, there is no simple method of converting a proof using one of the other approaches into a proof using our approach. Thus, from one point of view, our approach is a restriction of the other approaches. However, our approach generates proof systems that are (relatively) complete (as proved in this thesis) and are capable of being used for proving the correctness of any programs written in the given concurrent language.
VIII.3 Drawbacks of the Approach

When using deductive systems to define program semantics, a limitation arises simply from the nature of deductive systems. Gödel's incompleteness theorem[21] shows that every deductive system will admit true theorems whose truth cannot be proven within the system. Therefore, any set of program semantics based on deductive systems will admit correct programs whose correctness cannot be proven. We then must only be concerned with proving the relative completeness of our proof systems.

At first glance, it appears that our approach requires extensive analysis at the parallel composition stage. This belief comes from the necessity of analyzing an infinite number of varying length interaction sequences. The proof is simplified by performing an inductive proof on the length of each process interaction sequence. We believe that this proof is simpler than the cooperation check required by [4,8,19,20,32].

Lastly, by not allowing assumptions to be made about the behavior of the other processes in the proof of a process, we pay a price in that our process postconditions and invariants are usually weaker than the corresponding postconditions and invariants of [4,8,19,20,32]. Despite this weakness, our approach generates proof systems that are (relatively) complete.

VIII.4 Comparison to Related Works

There exist several proof systems for other concurrent programming languages which also avoid cooperation checks by the use of invariants and histories. We briefly comment on these approaches.

In Apt[2], a formal justification is given for the proof system of Apt et al.[4] which advocates the use of assumptions about the behaviors of the remaining pro-
cesses of the program in the proof of a single process. Therefore, the proof system of [4] is very similar to the work of Gerth et al.[19,20] and Barringer & Marns[8]. However, the transformation of the approach of [4] to the concurrent languages Ada and DP[8,19,20] was not straightforward and required the introduction of multiple control point assertions, and additional types of bracketed sections and inference tests. It is interesting to note that in order to prove the completeness of the proof system of [4], Apt[2] required the use of histories.

The main tenent behind the systems of Lamport[28] and Lamport & Schneider[29] is the use of invariance to reason about concurrent programs. To use their proof system, one must determine a predicate $I$ such that if the program is started in any state satisfying $I$, then every state reached during its execution satisfies $I$. This approach considers each atomic action in isolation and ignores the history of computation. Their assertions include predicates that are a function of the program location which requires additional axioms to capture the semantics of the program location predicates. Also, the COBEGIN rule requires that the program invariant must be the same as the invariants used for the subcomponents of the program which makes their proof system difficult to use and does not advocate modularity.

The proof system of Misra, Chandy and Smith[31] is centered around the trace of a process which is a chronological sequence of all interactions that a process has with its environment in a particular computation. Their rules use induction on the number of messages transmitted. The specification of a process requires three propositions: the pre- and postconditions are assertions on the traces of the process and the activity condition is an assertion on the empty/nonempty status of the channels connected to the process' ports. It is interesting to note that the Communication Axiom is similar in spirit to the predicate COMPAT. More importantly, in the proof that a process will never violate its assertions, Misra et al.[31] allow assumptions about the numbers that a process receives from the other processes in
the program which is a major departure from the modular approach presented in this thesis.

In Zhou and Hoare[44], an assertion is a predicate with free channel names, each of which stands for the sequence of values which have been communicated along that channel up to some moment in time. A process invariantly satisfies an assertion if that assertion is true before and after each communication by that process. They regard a process as being defined not by its internal states and transitions, but rather by its externally visible behavior; or, more precisely, by the set of all traces of its possible communications with its neighbors. The approach used in this proof system[44] is closely related to the proposed modular approach. However, their approach has only been applied to the language CSP and they explicitly use invariants where as we only use the invariant notation when dealing constructs that do not terminate.

Zwiers, de Roever, and van Emde Boas[45] believe that it is possible to infer a specification for the whole construct from specifications of the constituent syntactic components of that construct, without additional knowledge of the internal structure of these components. Therefore, they also stress the importance of compositionality. However, it is unknown whether their approach is universal.

VIII.5 Future Work

A natural observation we can make when examining the two levels of concurrency of DP is that our approach can easily be applied to deal with hierarchical parallelism. We stopped at the program level when constructing the proof system for DP since the language could not support the definition of any higher program levels. It is our suggestion that DP be extended to support hierarchical decomposition. A process should be able to "call" another program without having to know the internal structure of the program. With this extension, the SORT example
given in Chapter V could have been better written with the $N$ sorting processes comprising one program and a process of another program submitting a list of numbers to the sorting program and receiving the sorted list in return. The process should not need to be aware of the contents of the sorting program or how the sorting method is implemented in order to obtain a sorted list.

One aspect of Ada that we do not deal with in our axiomatics is shared variables. We decided to ignore shared variables mainly because there doesn't seem to be any (informal) description of exactly what these variables are and exactly how they are shared between the various tasks. Once a description becomes available, we could include shared variables into our proof system for Ada using the approach of recording the observed and assigned values of the shared variables on the corresponding interaction sequence.

It is interesting to note the differences between Ada and Distributed Processes that are emphasized when one examines the corresponding proof systems. The coroutine structure of a DP process causes multiple control points which justifies the complexity of the process rule when compared to the single control point task rule. However, multiple control points permit the dynamic nesting of procedure calls which can increase performance but they do not permit busy wait. Also, DP processes are more nondeterministic that Ada tasks and therefore cannot give priority to incoming procedure calls. Lastly, in Ada, an element of type return is always paired with the first preceding call element which represents a single execution of an ACCEPT body. This is not the case in DP since a return element can be paired with any previous call element if multiple incarnations of the procedure body exist.

A natural question to consider is whether our modular approach will be able to handle any future concurrent programming languages. One can imagine that a future concurrent language would combine a small number of the constructs identified by Andrews & Schneider[1] since the majority of existing concurrent programming
languages consist of a small number of these constructs. We believe that the semantics of the individual constructs defined by our approach can be combined to define the semantics of future concurrent programming languages.
APPENDIX

FUNCTIONS AND PREDICATES

Chapter II

\[ B(g_k) \] The Boolean portion of the guard \( g_k \). Notice that \( B(g_k) = \text{true} \) if \( g_k \) is of the form \( P_j?x \) or \( P_j!y \)

\[ C(g_k) \] The I/O portion of the guard \( g_k \). Notice that \( C(g_k) = \text{SKIP} \) if \( g_k \) is a purely Boolean guard

\[ \text{PB} \] The set of indices of the purely Boolean guards

\[ \text{IO} \] The set of indices of the I/O guards

\[ D(g_k) \] The index of the process addressed in the I/O portion of the guard \( g_k \)

\[ h/i \] Remove all elements from the sequence \( h \) that do not reference the process \( P_i \)

\[ \text{COMPAT}(h_1, \ldots, h_n) = \exists h. \left( \left[ \forall i. 1 \leq i \leq n \Rightarrow h/i = h_i \right] \land \left[ \forall m. 1 \leq m \leq \#h \Rightarrow \left[ \text{ELEM}(h,m) = (i,T,r) \right] \Rightarrow \left[ \forall j \in T \mid h[m+1:\#h]/j = e] \right] \right) \right) \]

\[ \#h \] The number of elements in the sequence \( h \)

\[ Tr(h) \] The sequence obtained from \( h \) by replacing each element in \( h \) by the value being communicated in the element; that is, \( Tr(h) \) is the sequence of "values" communicated in \( h \)

\[ \text{EVEN}(X) \] \( X \) is an even number

\[ \text{ELEMB}(h,k) \] The \( k^{th} \) element from the right end of the sequence \( h \)
\( ELEM(h,k) \) \hspace{1cm} \text{The } k^{th} \text{ element from the left end of the sequence } h \\

\( REST(h) \) \hspace{1cm} \text{The sequence obtained from } h \text{ by removing } ELEM(h,1) \text{ from } h \\

\( f(S,h) \equiv \begin{align*}
S & \quad \text{if } h = \varepsilon \\
S - \{ ELEM(h,1) \} & \quad \text{if } \# h = 1 \\
\{ (S - \{ ELEM(h,1) \}) \cup \{ ELEM(h,2) \}, REST(REST(h)) \} & \quad \text{Otherwise}
\end{align*} \\

\( ODD(X) \) \hspace{1cm} X \text{ is an odd number} \\

\( h[i:k] \) \hspace{1cm} \langle ELEM(h,i), ELEM(h,i+1), \ldots, ELEM(h,k) \rangle \\

\( g(T,h) \text{ equiv} \) \hspace{1cm} \begin{align*}
T & \quad \text{if } h = \varepsilon \\
T \cup \{ ELEM(h,1) \} & \quad \text{if } \# h = 1 \\
g((T \cup \{ ELEM(h,1) \}) - \{ ELEM(h,2) \}, REST(REST(h))) & \quad \text{Otherwise}
\end{align*} \\

\textbf{Chapter III} \\

\( \Sigma \) \hspace{1cm} \text{The set containing all label-assertion pairs associated with the exit-condition} \\

\( LABELS(\Sigma) \) \hspace{1cm} \text{The set of labels in } \Sigma \\

\( \Phi \) \hspace{1cm} \text{The empty set} \\

\( PROPER(\Sigma) \) \hspace{1cm} \text{Determines whether the label names in the set } \Sigma \text{ are unique} \\

\( \Pi \) \hspace{1cm} \text{The set of all exception names} \\

\( \Psi(H) \) \hspace{1cm} \text{The set of all exception names enumerated within the exception handler } H. \\

\( ASSERT(\Sigma,E) \equiv \begin{align*}
p & \quad \text{where } (E:p) \in \Sigma
\end{align*} \)
\[ \text{EXCPT}(H) = \begin{cases} \Pi & \text{If the exception handler contains an OTHERS clause} \\ \Psi(H) & \text{Otherwise} \end{cases} \]

\[ \text{REST}(\Sigma, \text{EXCPT}(H)) \]  
The set obtained by removing all exception-assertion pairs from \( \Sigma \) whose exception name is contained in \( \text{EXCPT}(H) \) along with the Terminate pair.

Chapter IV

\[ \text{EXCEPTION}(\Sigma) \]  
The set of exception names in \( \Sigma \)

\[ \text{EXCEPT}(T_i, A_j) \]  
The set of exception-assertion pairs that contains all exceptions that could be raised during the execution of the \( \text{ACCEPT} \) statement \( T_i, A_j \)

\[ \text{TYPE}(h, k) \]  
The "type" of the \( k^{th} \) element from the left of \( h \). The possible types of elements used in this system are \{call.to, return.from, exception, call, return, except, CALL, RETURN, EXCEPTION\}. At the program level, the two element types that characterize an accepted entry call and its return are combined into an element of type CALL and RETURN respectively. Also, the two element types that characterize the propagation of an exception are combined into an element of type EXCEPTION.

\[ \text{REST}(h) = \begin{cases} h' & \text{if } h = h[1:1] \text{ } h' \end{cases} \]

\[ \text{TASK}(h, k) \]  
The task number of the \( k^{th} \) element from the left of the sequence \( h \)

\[ \text{TASK.CALL}(h, k) \]  
The number of the task called in the \( k^{th} \) element from the left end of the sequence \( h \)

\[ \text{ACCEPT}(h, k) \]  
The number of the \( \text{ACCEPT} \) statement in the \( k^{th} \) element from the left end of the sequence \( h \)

\[ \text{VALP}(h, k) \]  
The value of the parameters in the \( k^{th} \) element from the left end of the sequence \( h \)
\( h/i \)  \( \varepsilon \)  if \( h=\varepsilon \)

\[
\text{(call.to,i,j,k,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=CALL} \wedge \text{TASK(h,1)=i} \wedge \\
\text{TASK.CALL(j) \wedge ACCEPT(k)} \wedge \\
\text{VALP(h,1)=X]} \)

\[
\text{(call,i,k,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=CALL} \wedge \\
\text{TASK.CALL(h,1)=i} \wedge \text{ACCEPT(h,1)=k} \wedge \\
\text{VALP(h,1)=X]} \)

\[
\text{(return,i,k,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=RETURN} \wedge \\
\text{TASK.CALL(h,1)=i} \wedge \text{ACCEPT(h,1)=k} \wedge \\
\text{VALP(h,1)=X]} \)

\[
\text{(return.from,i,j,k,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=RETURN} \wedge \\
\text{TASK.CALL(h,1)=i} \wedge \text{ACCEPT(h,1)=k} \wedge \\
\text{VALP(h,1)=X]} \)

\[
\text{(except,i,k,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=EXCEPTION} \wedge \\
\text{TASK.CALL(j) \wedge ACCEPT(k)} \wedge \\
\text{e} \in \text{EXCEPT(T,A_k)} \wedge \text{VALP(h,1)=X]} \)

\[
\text{(exception,i,j,k,e,}X) \sim \text{REST(h)/i}
\]

if \( \text{[TYPE(h,1)=EXCEPTION} \wedge \\
\text{TASK(h,1)=i} \wedge \text{TASK.CALL(h,1)=j} \wedge \\
\text{ACCEPT(h,1)=k} \wedge \text{e} \in \text{EXCEPT(T,A_k)} \wedge \\
\text{VALP(h,1)=X]} \)

\{\text{Generate the task sequence } h_i \text{ from the program sequence } h\}

\[
\text{COMPAT(h,1,\ldots,n)} \equiv \exists h. \ [i \leq n \ [h/i = h_i]]
\]

\[
\text{ODD(h,k)} \quad \text{The } k^{th} \text{ element of the sequence obtained by removing all the even numbered elements from } h
\]

\[
\text{EVEN(h,k)} \quad \text{The } k^{th} \text{ element of the sequence obtained by removing all the odd numbered elements from } h
\]

\[
\text{BUFFERS.PUT} \quad \text{The number of elements of type "return" from the ACCEPT statement } \text{PUT in the process sequence } h_B
\]
The number of elements of type "return" from the ACCEPT statement GET in the process sequence $h_B$

The sequence generated from $h_B$ by concatenating the values of the elements (call,B,P,POOL(inm mod N))

The sequence generated from $h_B$ by concatenating the values of the elements (return,B,G,POOL(outm mod N))

The sequence generated by concatenating the values of the elements of $ODD(h_P,k)$

The sequence generated by concatenating the values of the elements of $EVEN(h_C,k)$

**CHAPTER V**

The $k^{th}$ element from the left end of the sequence $h$

The $k^{th}$ element from the right end of the sequence $h$

$(ELEM(h,i), ELEM(h,i+1), \ldots, ELEM(h,k))$

The "type" of the $k^{th}$ element from the left of $h$. The possible types of elements used in this system are: {call.to, return.from, call, return, suspend, resume, end, ICALL, IRETURN, CALL, RETURN}. At the process level, the two element types that characterize an internal procedure call are combined into an element of type ICALL. Also, the two element types that characterize the return from an internal procedure call are combined into an element of type IRETURN. At the program level, this combination of elements also occurs for the external procedure call and return which are renamed CALL and RETURN respectively.

The process number of the $k^{th}$ element from the left of the sequence $h$

The procedure number of the $k^{th}$ element from the left of the sequence $h$

The incarnation number of the $k^{th}$ element from the left of the sequence $h$
VAL(h,k) The values of the shared variables and the parameters in the k'th element from the left of the sequence h

VALS(h,k) The values of the shared variables in the k'th element from the left of the sequence h

VALP(h,k) The values of the parameter variables in the k'th element from the left of the sequence h

PROCESS.CALL(h,k) The number of the process called in the k'th element from the left of the sequence h

PROC.CALL(h,k) The number of the procedure called in the k'th element from the left of the sequence h

REST(h) ≡ h' if h = ELEM(h,1) ^ h'

NO.CALLS(h) The number of times an element of type "call" appears in the sequence h

TRIM(h,k) Remove the incarnation number from the k'th element from the left of the sequence h

CHECK(h) ≡ ∀k. 2≤k≤#h [TYPE(h,k)=call => TYPE(h,k-1)∈{return,suspend} ∧ ∀j<k [TYPE(h,j)=call ⇒ COPY(h,j)≠COPY(h,k)]]

SEQ(h,k) ε if h=ε

(call,i,k,آثار,cpy)^SEQ(REST(h),k)

if [PROC(h,1)=0 ∧ TYPE(h,1)=ICALL ∧ PROCESS(h,1)=i ∧ PROC.CALL(h,1)=k ∧ VAL(h,1)=(آثار,آثار) ∧ COPY(h,1)=cpy]

(return,i,k,آثار,cpy)^SEQ(REST(h),k)

if [PROC(h,1)=0 ∧ TYPE(h,1)=IRETURN ∧ PROCESS(h,1)=i ∧ PROC.CALL(h,1)=k ∧ VAL(h,1)=(آثار, إليه) ∧ COPY(h,1)=cpy]

ELEM(h,1)^SEQ(REST(h),k) if PROC(h,1)=k

SEQ(REST(h),k) Otherwise
\{Generate a sequence from the elements in the sequence \( h \) that reference the procedure \( k \}\}

\[
\text{SUBSEQ}(k,h) \quad \epsilon \quad \text{if } [(k > \text{NO.CALLS}(h)) \lor (h = \epsilon)]
\]

\[
\text{TRIM}(h,1) \& \text{g}(\text{REST}(h), \text{PROC}(h,1), \text{COPY}(h,1))
\quad \text{if } [(k-1) \land (\text{TYPE}(h,1) = \text{call})]
\]

\[
\text{SUBSEQ}(k-1, \text{REST}(h)) \quad \text{if } [(k \neq 1) \land (\text{TYPE}(h,1) = \text{call})]
\]

\[
\text{SUBSEQ}(k, \text{REST}(h)) \quad \text{Otherwise}
\]

\[
g(h,j,\text{cpy}) \quad \epsilon \quad \text{if } h = \epsilon
\]

\[
\text{TRIM}(h,1) \& \text{g}(\text{REST}(h), j, \text{cpy})
\quad \text{if } [\text{PROC}(h,1) = j \land \text{COPY}(h,1) = \text{cpy}
\quad \land \text{TYPE}(h,1) \neq \text{return}]
\]

\[
\text{TRIM}(h,1)
\quad \text{if } [\text{PROC}(h,1) = j \land \text{COPY}(h,1) = \text{cpy}
\quad \land \text{TYPE}(h,1) = \text{return}]
\]

\[
g(\text{REST}(h), j, \text{cpy}) \quad \text{Otherwise}
\]

\{Generate a sequence from the elements in the sequence \( h \) that reference the \( k^{th} \) execution of the given procedure with their incarnation number removed\}

\[
f_{\text{init}}(h) \quad \epsilon \quad \text{if } h = \epsilon
\]

\[
\text{(call.to,i,0,i,j,X,vasion)} \& f_{\text{init}}(\text{REST}(h))
\quad \text{if } [\text{TYPE}(h,1) = \text{ICALL} \land \text{PROCESS}(h,1) = i \land
\quad \text{PROC.CALL}(h,1) = j \land \text{VAL}(h,1) = (X, invasion)]
\]

\[
\text{(return.from,i,0,i,j,X,vasion)} \& f_{\text{init}}(\text{REST}(h))
\quad \text{if } [\text{TYPE}(h,1) = \text{IRETURN} \land \text{PROCESS}(h,1) = i \land
\quad \text{PROC.CALL}(h,1) = j \land \text{VAL}(h,1) = (X, invasion)]
\]

\[
\text{ELEM}(h,1) \& f_{\text{init}}(\text{REST}(h))
\quad \text{if } [\text{PROC}(h,1) = 0 \land \text{TYPE}(h,1) \in \{\text{suspend,}
\quad \text{resume,call.to,return.from}\}]
\]

\[
\text{ELEM}(h,1) \quad \text{if } \text{TYPE}(h,1) = \text{end}
\]
\[ f_{\text{init}}(\text{REST}(h)) \quad \text{if } PROC(h,1)\neq 0 \]

\{Generate a sequence from the elements in the sequence h that reference the process initialization\}

\[ r'_{i,\text{init}} \equiv r_{i,\text{init}} f_{\text{init}}(h) \]

\[ r'_{i,j} \equiv r_{i,j} \text{SEQ}(h,j) \]

\[ \text{CONIST}(h) \equiv \forall k \leq \#h \ [\text{TYPE}(h,k)\in \{\text{call, resume, RETURN}\} \Rightarrow \]

\[ \text{VALS}(h,k) = \text{VALS}(h,k-1)] \]

\{The values of the process variables observed by the process are the values that have been most recently assigned to them\}

\[ \text{STRIP}(h) \]

\( \epsilon \quad \text{if } h=\epsilon \)

\[ \text{ELEM}(h,1)^{\text{STRIP}}(\text{REST}(h)) \quad \text{if } \text{TYPE}(h,1)\in \{\text{call, return}\} \]

\[ \text{TYPE}(h,1),i,j,k,\bar{x},\text{cpy})^{\text{STRIP}}(\text{REST}(h)) \quad \text{if } \{\text{TYPE}(h,1)\in \{\text{call, return, from}\} \land \]

\[ \text{PROCESS}(h,1)=i \land \text{PROC.CALL}(h,1)=k \land \]

\[ \text{PROCESS.CALL}(h,1)=j \land \text{VALP}(h,1)=\bar{x} \land \]

\[ \text{COPY}(h,1)=\text{cpy} \}

\[ \text{STRIP}(\text{REST}(h)) \quad \text{Otherwise} \]

\{The sequence obtained by removing the elements from the process sequence that an external observer would not see\}

\[ h/i \]

\( \epsilon \quad \text{if } h=\epsilon \)

\[ \{\text{callToList, return, from}\}^{\text{REST}}(h)/i \quad \text{if } \{\text{TYPE}(h,1)=\text{CALL} \land \text{PROCESS}(h,1)=i \land \]

\[ \text{PROCESS.CALL}(h,1)=j \land \text{VALP}(h,1)=\bar{x} \land \]

\[ \text{PROC.CALL}(h,1)=k \}

\[ \{\text{callToList, return, from}\}^{\text{REST}}(h)/i \quad \text{if } \{\text{TYPE}(h,1)=\text{CALL} \land \text{PROC.CALL}(h,1)=k \land \]

\[ \text{PROCESS.CALL}(h,1)=i \land \text{VALP}(h,1)=\bar{x} \} \]
(return from, i, j, k, \overline{V}) \wedge REST(h)/i
\quad \text{if } \{TYPE(h,1) = \text{RETURN } \wedge \text{PROCESS}(h,1) = i \wedge \text{PROC.CALL}(h,1) = j \wedge \text{PROC.CALL}(h,1) = k \wedge \text{VALP}(h,1) = \overline{V}\}

(return, i, k, \overline{V}) \wedge REST(h)/i
\quad \text{if } \{TYPE(h,1) = \text{RETURN } \wedge \text{PROC.CALL}(h,1) = k \wedge \text{PROC.CALL}(h,1) = i \wedge \text{VALP}(h,1) = \overline{V}\}

\{\text{Generate the process sequence } h, \text{ from the program sequence } h\}

COMPAT(h_1, \ldots, h_n) \equiv \exists h. [\forall i \leq n \: [h/i = h_i]]

\textbf{Functions Used in \textit{SORT} Example}

\textbf{TYPEB}(h,k)
\quad \text{The "type" of the } k^{th} \text{ element from the right of the sequence } h

\textbf{VALB}(h,k)
\quad \text{The values of the parameters in the } k^{th} \text{ element from the right of the sequence } h

\textbf{LR}(h) \equiv \begin{cases} h' & \text{if } h = h' \wedge \text{EMB}(h,1) \\ \text{ELEM}(h,1) & \end{cases}

\textbf{FIRST}(h) \equiv \text{ELEM}(h,1)

\textbf{NO.OF}(h,t)
\quad \begin{cases} 0 & \text{if } h = \epsilon \\ \text{NO.OF}(\text{REST}(h),t) + 1 & \text{if } TYPE(h,1) = t \\ \text{NO.OF}(\text{REST}(h),t) & \text{Otherwise} \\ \end{cases}

\{\text{The number of elements of } TYPE \ t \text{ in the sequence } h\}

\textbf{HERE}_i(h)
\quad \begin{cases} \epsilon & \text{if } (h = \epsilon) \lor \{\text{TYPEB}(h,1) = \text{"RETURN from GET,"} \} \\ \text{\textbf{HERE}}_i(\text{LR}(h)) \wedge \text{VALB}(h,1) & \text{if } TYPE(h,1) = \text{"RETURN from PUT,"} \\ \text{\textbf{HERE}}_i(\text{LR}(h)) - \text{MAX}(\text{\textbf{HERE}}_i(\text{LR}(h))) & \end{cases}

\textbf{NO.OF}(h,1)
\quad \begin{cases} 0 & \text{if } h = \epsilon \\ \text{NO.OF}(\text{REST}(h),1) + 1 & \text{if } TYPE(h,1) = 1 \\ \text{NO.OF}(\text{REST}(h),1) & \text{Otherwise} \\ \end{cases}

\{\text{The number of elements of } TYPE \ 1 \text{ in the sequence } h\}
if $TYPEB(h,1) = \text{"RETURN from } PUT_{i+1}\text{"}$

$(VALB(h,1))$ if $TYPEB(h,1) = \text{"RETURN from } GET_{i+1}\text{"}$

$HERE_i(LR(h))$ Otherwise

\{The numbers associated with $SORT_i$ (at most 2) that are from the list of numbers to be sorted\}

$REST_i(h) = NO.OF(h, \text{"RETURN from } PUT_{i+1}\text{"}) -\linebreak
NO.OF(h, \text{"RETURN from } GET_{i+1}\text{"})$

\{The number of elements from the list of elements to be sorted that are contained in the processes $SORT_{i+1}, \ldots, SORT_n$\}

$LEN_i(h) = NO.OF(h, \text{"RETURN from } PUT_i\text{"}) +\linebreak
NO.OF(h, \text{"RETURN from } GET_{i+1}\text{"}) -\linebreak
NO.OF(h, \text{"RETURN from } GET_i\text{"}) -\linebreak
NO.OF(h, \text{"RETURN from } PUT_{i+1}\text{"})$

\{The number of elements from the list of elements to be sorted that are contained in $SORT_i$ (the length of $HERE_i$)\}

$QUEUE(h)$

\[ \begin{cases} \varepsilon & \text{if } h = \varepsilon \\ REST(Queue(LR(h))) & \text{if } TYPEB(h,1) = \text{"Return from } GET_1\text{"} \\ INSERT(Queue(LR(h)), VALB(h,1)) & \text{if } TYPEB(h,1) = \text{"Return from } PUT_1\text{"} \end{cases} \]

$INSERT(q,k)$

\[ \begin{cases} (k) & \text{if } q = \varepsilon \\ k \sim q & \text{if } (q \neq \varepsilon) \land (k \leq FIRST(q)) \end{cases} \]

$FIRST(q) \sim INSERT(REST(q),k)$ Otherwise

\{A sorted list of the numbers that are contained in the processes $SORT_1, \ldots, SORT_n$\}
**MIN(h)** \[ELEM(h,1) \quad \text{if } LEN_i(h) = 1 \lor \]
\[\quad [LEN_i(h) = 2 \land (ELEM(h,1) \leq ELEM(h,2))]\]
\[ELEM(h,2) \quad \text{if } LEN_i(h) = 2 \land [ELEM(h,2) < ELEM(h,1)]\]
\[\infty \quad \text{if } LEN_i(h) = 0\]

{The smallest element in the sequence \(HERE_i\)}

**MAX(h)** \[ELEM(h,1) \quad \text{if } LEN_i(h) = 1 \lor \]
\[\quad [LEN_i(h) = 2 \land (ELEM(h,1) \geq ELEM(h,2))]\]
\[ELEM(h,2) \quad \text{if } LEN_i(h) = 2 \land [ELEM(h,2) > ELEM(h,1)]\]
\[\infty \quad \text{if } LEN_i(h) = 0\]

{The largest element in the sequence \(HERE_i\)}

**LIST_i(h)** \(\varepsilon\) \quad \text{if } h = \varepsilon

\[\text{INSERT}(LIST_i(LR(h)), VALB(h,1))\]
\[\quad \text{if } TYPEB(h,1) = "RETURN from PUT,"\]

\[\text{REST}(LIST_i(LR(h))) \quad \text{if } TYPEB(h,1) = "RETURN from GET,"\]

{A sorted list of the numbers that are contained in the processes \(SORT_i, \ldots, SORT_n\)}

Chapter VI

**COMPAT**(t_1, \ldots, t_m, h_1, \ldots, h_n) =

\[\exists h. [h \in MERGE'(h_1, \ldots, h_n) \land \]
\[\forall k \leq \#h \ [TYPE(h,k) = \text{RECEIVE} \Rightarrow \]
\[\quad TYPE(h,k) = \text{SEND} \land VAL(h,k) = g(T, h[1:k-1]) \land \]
\[\forall k \leq \#h \ [TYPE(h,k) = \text{receive}_j \Rightarrow VAL(h,k) = f_j(t_j, h[1:k-1]) \land \]
\[\forall j \leq m [r_j = f_j(t_j, h)]]\]

**MERGE'(h_1, \ldots, h_n)** The set of all sequences obtained by merging \(h_1, \ldots, h_n\) in all possible ways
Chapter VII

\[ f_j(t,h) \begin{cases} 0 & \text{if } h=\varepsilon \\ f_j(t,REST(h)) & \text{if } TYPE(h,1) \notin \{SEND, send\} \\ f_j(t',REST(h)) & \text{if } ELEM(h,1) = (send, t') \\ f_j(t',REST(h)) & \text{if } ELEM(h,1) = (SEND, (1, t'_1), \ldots, (j, t'_j), \ldots, (m, t'_m)) \end{cases} \]

\[ g(t_1, \ldots, t_m, h) \begin{cases} (t_1, \ldots, t_m) & \text{if } h=\varepsilon \\ g(t_1, \ldots, t_m, REST(h)) & \text{if } TYPE(h,1) = \{receive, \Rightarrow RECEIVE\} \\ g(t_1, \ldots, t'_1, \ldots, t_m, REST(h)) & \text{if } ELEM(h,1) = (send, t'_1) \\ g(t'_1, \ldots, t'_m, REST(h)) & \text{if } ELEM(h,1) = (SEND, (1, t'_1), \ldots, (m, t'_m)) \end{cases} \]

**Ω** The set of all possible states

**E** The empty procedure

**N** The set of natural numbers

**MERGE**(h, h₁, ..., hₙ) A function that returns h, the set of all sequences obtained by merging h₁, ..., hₙ in all possible ways

**PCₖ(Pᵢ)** The statement to which the \(k^{th}\) program counter of the process \(Pᵢ\) points

**ACTIVE**(PCₖ(Pᵢ)) Determines whether the statement \(PCₖ(Pᵢ)\) is currently executing

**LOC**(l, k, S) The location of the statement S within the procedure \(Qₖ\) in the process \(Pᵢ\)

**PROC**(PCₗ(Pᵢ)) The procedure number of the statement at \(PCₗ(Pᵢ)\)

**PROJECT**(Cᵢₗ, Qᵢₗ) The statements of \(Qᵢₗ\) that follow \(ACTIVE(PCₗ(Pᵢ))\) to the end of the procedure
**INCARN\((i,l)\)**  The incarnation number of the \(l^{th}\) procedure in the \(i^{th}\) process

**ADD\((\ldots, INCARN(i,k))\)**

Add the incarnation number of the \(k^{th}\) procedure in the \(i^{th}\) process to the element \((\ldots)\)

**COPYB\((h,k)\)**  The incarnation number of the \(k^{th}\) element from the right end of the sequence \(h\)
LIST OF REFERENCES


36. Soundararajan, N. Consistency and Completeness of an Axiomatic Semantics
for CSP. Preliminary Draft, The Ohio State University, 1983.


38. Soundararajan, N. Total Correctness of CSP Programs. To appear in *ACTA Informatica*.


