PROOF RULES FOR
COMMUNICATING SEQUENTIAL PROCESSES

Gary Marc Levin
Ph.D. Thesis
August 1980

TR 80-435

Department of Computer Science
Cornell University
Ithaca, New York 14853
PROOF RULES FOR COMMUNICATING SEQUENTIAL PROCESSES

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment for the Degree of
Doctor of Philosophy

by
Gary Marc Levin
August 1980
Proof Rules for
Communicating Sequential Processes

Gary Marc Levin, Ph.D.,
Cornell University, 1980

This thesis presents proof rules for an extension of Hoare's Communicating Sequential Processes (CSP). CSP is a notation for describing processes that interact through communication, which provides the sole means of synchronizing and passing information between processes. A sending process is delayed until some process is ready to receive the message; a receiving process is delayed until there is a message to be received. It is this delay that provides synchronization.

A proof of a program is with respect to pre- and postconditions. A proof of weak correctness shows that execution of the program beginning in a state satisfying the precondition terminates in a state satisfying the postcondition, provided deadlock does not occur. A proof of strong correctness, in addition, shows that deadlock cannot occur.

A proof of weak correctness has three stages: a sequential proof, a satisfaction proof, and a non-interference proof. A sequential proof reflects the effects of a process running in isolation. A satisfaction proof combines sequential proofs of processes, reflecting the message passing and synchronization aspects of communication. A non-interference proof shows that no process affects the validity of the proof of another process.
The introduction of the satisfaction proof and our symmetric treatment of send and receive are important aspects of this thesis. By treating send and receive on an equal basis, we simplify our rules and allow the inclusion of send in guards.

A sufficient condition for freedom from deadlock is given that depends on the proof of weak correctness; this is used to prove strong correctness. In general, freedom from deadlock can be very hard to check. Therefore, we derive special cases in which we can reduce the work needed to verify that a program is free from deadlock.

We also present an algorithm for globally synchronizing processes; that is, each process can recognize that all processes are simultaneously in a given state. It works by recognizing a special class of deadlock. Having this algorithm allows us to modify programs that deadlock when the postcondition is established, so that they terminate normally.
Biographical Sketch

Gary Marc Levin was born in Baltimore, Maryland on February 1, 1954. He graduated from the Baltimore Polytechnic Institute in 1972. He received a BS in Computer Science and Statistics from the University of Delaware in 1975 and an MS in Computer Science from Cornell University in 1977. He is a member of Pi Mu Epsilon, Phi Kappa Phi, and the ACM.
To my mother

and

to the memory of my father
Acknowledgments

I would like to thank my advisor, David Gries, for the many hours he spent helping me. He guided me through both technical and expository difficulties, encouraging me to think and to write clearly. His responsibility extends only to the positive aspects of this thesis however.

Thanks also go to Fred Schneider — for his careful reading, for our long discussions together, and for organizing the Operating Systems Seminar, CS713, which provided an opportunity to present and to receive new ideas.

In general, I must extend thanks to all my friends in the department — for their patience when I needed to discuss my work and for their many suggestions.

Special thanks go to Carl Hauser, Rick Schlichting, and Daniel Zlatin.
Table of Contents

1 Introduction................................................................. 1

2 The notation CSP.......................................................... 4
  2.1 Simple commands..................................................... 4
  2.2 Composite commands............................................... 6
  2.3 Scheduling........................................................... 7
  2.4 Parameterization................................................... 8
  2.5 An example -- bounded buffer................................... 8
  2.6 Changes from Hoare's CSP......................................... 9

3 Proof rules for weak correctness....................................... 11
  3.1 Axioms.................................................................. 13
  3.2 Inference rules....................................................... 14
  3.3 Satisfaction proof................................................... 16
  3.4 Auxiliary variables................................................. 18
  3.5 Proof of non-interference......................................... 20
  3.6 Synchronously altered variables.................................. 22
  3.7 Universal assertions............................................... 23
  3.8 Summary................................................................ 23

4 Requirements for strong correctness.................................... 24
  4.1 A sufficient condition for freedom from deadlock............... 24
  4.2 Reducing the size of C(S)........................................... 27
  4.3 An awaited explanation............................................. 28

5 An example -- finding the minimum of a set........................ 31
  5.1 The program.......................................................... 31
  5.2 Sequential proof..................................................... 32
  5.3 Satisfaction proof................................................... 33
5.4 Non-interference proof........................................... 35
5.5 Proof of freedom from deadlock.................................. 36
5.6 A simpler proof of freedom from deadlock....................... 38
5.7 Comments on the proof........................................... 40
6 An operational model and proof of consistency.................... 42
   6.1 The model.................................................... 43
   6.2 Consistency with the model.................................. 44
7 Global synchronization in CSP...................................... 48
   7.1 The problem of global synchronization........................ 48
   7.2 A protocol for global synchronization......................... 50
   7.3 Protocol algorithm for a process.............................. 51
   7.4 Some process is signalled after all processes are quiet........ 53
   7.5 When a process is ready to quit, all processes are quiet..... 55
   7.6 The algorithm does not deadlock before some process is ready to quit .................................................. 58
   7.7 Comments.................................................... 59
8 Discussion.................................................................. 60
   8.1 Overview....................................................... 60
   8.2 The development of the communication axiom................... 61
   8.3 Logical Clocks................................................ 61
   8.4 Related work.................................................. 62
   8.5 Research...................................................... 64
9 Bibliography.......................................................... 67

vi
# Table of Figures

2.5.1 -- A bounded buffer........................................................................ 9
3.1.1 -- Equation..................................................................................... 13
3.2.1 -- Equation..................................................................................... 15
3.2.2 -- Equation..................................................................................... 15
3.2.3 -- An annotated program and corresponding proof......................... 17
3.3.1 -- Equation..................................................................................... 17
3.3.2 -- Equation..................................................................................... 17
3.3.3 -- Equation..................................................................................... 17
3.4.1 -- A bounded buffer (with auxiliary variables)................................. 20
3.5.1 -- Equation..................................................................................... 20
3.5.2 -- Equation..................................................................................... 20
4.1.2 -- Equation..................................................................................... 27
4.3.1 -- Equation..................................................................................... 28
4.3.2 -- Equation..................................................................................... 28
4.3.3 -- (a) Configuration set for 4.3.1

(b) Configuration set for 4.3.2................................................................... 29
4.3.4 -- Equation..................................................................................... 29
4.3.5 -- Equation..................................................................................... 29
4.3.6 -- Deadlock free

Not deadlock free.................................................................................... 30
5.1.1 -- Min............................................................................................. 31
5.2.1 -- Equation..................................................................................... 33
5.2.2 -- Equation..................................................................................... 33
5.2.3 -- Min (annotated).......................................................................... 34
5.5.1 -- Configurations H_{i}1, H_{i}2, and H_{i}3........................................ 36
5.5.2 -- Equation........................................... 37
5.5.3 -- Configurations A1, B1, and B2................. 38
6.2.1 -- $PQ[S] = PQ[PS^\pi]$.............................. 46
7.3.1 -- Protocol for global synchronization............ 52
Chapter 1

Introduction

Two common models for parallel computation are: centralized, where all processes share (have access to) all variables; and distributed, where the variables of each process are private, i.e. not accessible to other processes, and message passing is used to provide interaction. The centralized model has been well studied. Here we consider the distributed model, which corresponds to a system of processors, each with its own memory, and a communication network through which messages may be sent between processors.

To properly study algorithms, one needs a notation for their description. In this thesis we use an extension of Hoare's CSP (Communicating Sequential Processes) [10], which is derived from Dijkstra's simple programming notation [5].

The set of simple commands (assignment and skip) has been augmented by two communication commands, send and receive, which serve to synchronize and to pass information between processes. Message passing is typed and synchronous -- a process specifies the type of message to be communicated and the sender waits until the message has been received.

The commands alternation and repetition have been extended to allow send and receive to appear in guards. A guarded command can be selected based on another process's readiness to communicate messages of the correct type; this conditional communication is controlled by the requirement that message passing be typed and synchronous.

Parallel composition allows defining and naming of processes, where the variables of parallel processes are not shared.
An informal description of CSP is given in chapter 2. We list the commands and explain, intuitively, how they may be executed. We also explain how synchronization affects the order of execution. The major differences between CSP as we define it and as Hoare defined it are output guards, which we have added, and automatic termination, which we have deleted.

In this thesis, we separate the sequential proofs of processes, which may need assumptions about the effects of communication, from the satisfaction proofs, which show these assumptions to be valid. Satisfaction proofs place constraints on other processes, much as assumptions about the parameters of a procedure place constraints on how the procedure may be invoked.

In chapter 3, we give proof rules for weak correctness; that is, partial correctness and termination in the absence of deadlock. It is here that we introduce satisfaction and non-interference proofs and the notion of synchronously altered variables.

The satisfaction proof is our way of formalizing the intuitive notion that communication is a distributed form of assignment. In the rule of satisfaction, we have managed to incorporate both the synchronization and message passing aspects of communication.

In chapter 4, we present sufficient conditions for proving the absence of deadlock. Such proofs can require the analysis of a large number of cases; a method is given for reducing the complexity and number of these cases.

In chapter 5, we give a program for computing the minimum of a set. We then illustrate the use of our rules by proving that the program meets its specifications and does not deadlock.
Chapter 6 relates our proof rules to an operational model and proves the consistency of our system.

Chapter 7 contains another program with its proof; this program handles the problems of global synchronization and deadlock detection in CSP. Global synchronization can be used to solve the class of problems where a desired condition can be factored into the conjunction of subconditions over disjoint state spaces.

Chapter 8 summarizes our results and relates our work to that of others.
Chapter 2

The notation CSP

In this chapter we present an informal, operational semantics of CSP from which the reader can get an intuitive grasp of the meaning of the notation.

In dealing with concurrency we confront problems of synchronization and in doing so we will use the terms ready, blocked, and terminated. A ready process can execute a command, a blocked process is waiting for communication so that it can continue execution, and a process that has executed its last command is terminated. In our description below, we explain when a process is ready or blocked.

2.1 Simple commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip:</td>
<td>skip</td>
</tr>
<tr>
<td>assignment:</td>
<td>$x_1, \ldots, x_n := e_1, \ldots, e_n$ or $x := \bar{e}$</td>
</tr>
<tr>
<td>send:</td>
<td>$\text{A}!T(x)$</td>
</tr>
<tr>
<td>receive:</td>
<td>$\text{A}?T(x)$</td>
</tr>
</tbody>
</table>

The first two simple commands, from Dijkstra's sequential notation, place no restriction on execution; a process whose next command is skip or assignment is ready and can execute.

Execution of skip does nothing. It corresponds to the null statement of Algol 60; we prefer to have an explicit representation.

To execute an assignment: determine the references represented by $x_1, \ldots, x_n$ and the values of $e_1, \ldots, e_n$, then store the values in the locations of the corresponding references, in left-to-right order. We frequently use $\bar{x}$ to represent the vector $x_1, \ldots, x_n$. 

4
The communication commands send and receive have three parts: a process name A, a template (which for simplicity we restrict to an identifier) T and a parameter list (a list of variables or expressions) \( \bar{x} \). A send command \( A!T(\bar{x}) \) and a receive command \( B?T(\bar{x}) \) form a matching pair if and only if \( A!T(\bar{x}) \) appears in process B, \( B?T(\bar{x}) \) appears in process A, they have the same template T, and \( \bar{x}:=\bar{\bar{x}} \) is a syntactically legal multiple assignment.\(^1\)

For the rest of the thesis, in order to simplify the discussion, we will use the notation \( A.T(\bar{x}) \) to refer to a send \( A!T(\bar{x}) \) or a receive \( A?T(\bar{x}) \) whenever it is immaterial which it is. This notation never appears in the text of a program; it is used only to reduce repetition in definitions.

A matching pair is executed as follows. Suppose execution of process \( B \) is at a command \( A.T(\bar{x}) \). If process \( A \) is not at a matching command \( B.T(\bar{y}) \), then \( B \) is said to be blocked and prepared to communicate. On the other hand, if \( A \) is at a matching command \( B.T(\bar{y}) \), then both processes are ready, and the commands may be executed simultaneously. Such execution consists of executing the multiple assignment statement \( \bar{x}:=\bar{y} \) (or \( \bar{y}:=\bar{x} \), depending on which command is the send). When such execution occurs, the processes are said to have synchronized at the matching pair.

---

\(^1\) The template is a means of distinguishing between different kinds of messages that are passed between a pair of processes. If there is no need for this distinction, templates can be omitted. If the parameter list has length one and there is no template, we omit parentheses. If there is no information to be passed (only the type of message is important), the parameter list and parentheses are omitted.
2.2 Composite commands

sequence: \[ S_1; \cdots; S_n \]
parallel: \[ [A_1::S_1 \ || \ \cdots \ || A_n::S_n] \]
alternation: \[ \text{if } b_1; c_1 + S_1 \ || \ \cdots \ || b_n; c_n + S_n \ \text{fi} \]
repetition: \[ \text{do } b_1; c_1 + S_1 \ || \ \cdots \ || b_n; c_n + S_n \ \text{od} \]

Composition provides the means of forming commands from simpler ones. Two of the composite commands place no restriction on execution (except for restrictions due to component commands): sequence and parallel.

Sequence is the most familiar form of composition. (In fact, it is frequently called composition; we have renamed it to avoid confusion.) To execute \[ S_1; \cdots; S_n \], execute \[ S_1 \], then \[ S_2; \ldots \], and finally \[ S_n \].

Execution of a parallel command consists of executing the component commands \[ S_1; \ldots; S_n \] concurrently; the command is completely executed when all component commands are completely executed. Commands \[ S_1; \ldots; S_n \] are referred to as processes and are named \[ A_1; \ldots; A_n \].

Alternation and repetition are formed from sets of guarded commands. A guarded command \[ b; c + S \] consists of a guard \[ b \]; \[ c \] and a command \[ S \]. In the guard, \[ b \] is a boolean expression and \[ c \] is either \text{skip} or \[ A.T(x) \].

We now explain when a guard is aborted, ready, or blocked. If \[ b \] is false, the guard is aborted. If \[ b \] is true and \[ c = \text{skip} \], the guard is ready. If \[ b \] is true and \[ c = A.T(x) \], the guard is prepared to communicate with process \[ A \]; it is ready when \[ A \] is prepared to communicate with this process\(^2\) and blocked at other times.

Execution of an alternative command selects a guarded command with a ready guard and executes the sequence \[ c; S \]. If \[ c \] is \text{skip}, execution is

\(^2\) Of course, \( A \)'s message must be going in the correct direction and be of the same type.
independent of other processes; but if \( c \) is a communication command, then a matching communication command must be executed simultaneously. When some guards are blocked and none are ready, the process is blocked and must wait. If all guards abort, the command (and the program) aborts.

Execution of the repetitive command is the same except that, whereas execution of alternation selects and executes one guarded command and is completed, for repetition selection is repeated until all guards are aborted.

The boolean true or the command \( \text{skip} \) may be omitted from guards, in which case the separating semicolon is omitted as well.

2.1 Scheduling

In sequential notations, there is the tacit assumption that a program continues executing at some finite speed until it reaches the end or aborts. This allows the conclusion that a program that could terminate will in fact terminate. In CSP, where there can be more than one process, we must recognize that there are sometimes choices to be made as to which processes are to proceed. This is a question of scheduling. Our assumptions are based on our interest in correctness in the absence of deadlock. Other interests, such as questions about starvation or fairness, may require other assumptions.

A system is deadlocked if some process is not terminated and no process is ready. Our scheduling assumption is: if any process is ready, progress must be made in a finite, bounded amount of time; i.e., if the system is not deadlocked, something will happen and progress will be made.
2.4 Parameterization

When many similar processes are desired we allow abbreviations like

\[ \left[ \left\lfloor A_1 \ldots : S_i \right\rfloor \right] \]

\[ \text{ieN} \]

to represent

\[ \left[ A_{n_1} : S_{n_1} \left| \cdots \left| A_{n_k} : S_{n_k} \right. \right. \right] \]

where \( N = \{n_1, \ldots, n_k\} \).

In the case of repetition and alternation, we use

\[ \square b_i : c_i + S_i \]

\[ \text{ieN} \]

to mean

\[ b_{n_1} : c_{n_1} + S_{n_1} \square \cdots \square b_{n_k} : c_{n_k} + S_{n_k} \]

2.5 An example -- bounded buffer

At this point, a simple, well-known example is the best way for readers to check their understanding. Because CSP assumes synchronous message passing, a common complaint about it is that processes are too tightly coupled. But it is simple to loosen the coupling and allow asynchronous behavior by inserting a buffer process between processes, as we now show.

Assume that two processes I and O wish to communicate through a buffer process B. Process I sends message e to O by using "B!e" and O receives a message in x by using "B?x". Process B is shown in Figure 2.5.1 for a buffer of size \( N > 0 \).

Process B can receive two kinds of messages: an integer to be buffered, which is a parameter without a template; and quit, which is a template without a parameter. B can receive an integer if the buffer is
Declare b[0:N-1], out, count, and done.

1) Buffered messages are, in order, in b[i mod N],
   for i such that out ≤ i < out+count.
2) 0 ≤ count ≤ N.
3) "*" is used to denote addition modulo N.

out, count, done := 0, 0, false;

if not done ∧ count < N; I!b[out=count] ⇥ count := count+1
   count > 0; O!b[out]    ⇥ out, count := out mod 1, count - 1
end;
I!quit ⇥ done := true
O!quit

Figure 2.5.1 -- A bounded buffer

not full (count < N) and process I has not indicated that there are no more
messages to be buffered (by sending the message quit to B). B can send a
buffered value if the buffer is not empty (count > 0). If the buffer is
neither full nor empty, B can either send or receive; what it does next
depends on the speeds of I and O and on the scheduler. B can also receive
quit from I; having received this message, B can no longer receive from I.
When the repetitive command terminates, B sends quit to O and terminates.

2.6 Changes from Hoare's CSP

There are two main differences between our notation and Hoare's [10]:
the inclusion of send in guards and the removal of automatic termination.

Hoare allows only receive in guards, although he suggests the
possibility of send guards. Others have also suggested the inclusion of
output guards [2,12,16]. Before making this change, we considered
implementation problems and determined that, at least in some environments
[15], there is no additional overhead in providing conditional output.
The program in section 2.5 illustrates one advantage of this addition. Without output guards, 0 would have to send a request for input to B. This would entail extra messages, place an unnecessary burden on the programmer, and change the form of process O. As is, 0 looks the same regardless of whether it is receiving input synchronously or from a buffer.

When all processes that a repetition is waiting for have terminated, Hoare's notation calls for automatic termination of the loop. We have not included this feature because it complicates the semantics. Furthermore, algorithms that use it can be difficult to modify. In particular, if the modification requires that the previously terminating processes continue, there is no simple way to terminate the loop; new signals and, possibly, restructuring of the algorithm will be needed. Instead, we assume that all termination of loops is done explicitly.3

Further motivation for these changes is found in chapter 3. There we explain how developing our proof rules helped us to make these design decisions.

---

3 The message quit is used in the bounded buffer example for this purpose.
Chapter 3

Proof rules for weak correctness

A proof of weak correctness (total correctness in the absence of deadlock) of a set of communicating processes consists of three parts: a sequential proof, a satisfaction proof, and a non-interference proof.

The sequential proof for each process is in the style of a Hoare-logic proof.

While creating the sequential proof, assumptions are made about the effect of execution of communication commands. A satisfaction proof shows that these assumptions are valid.

Although the original notation requires that each variable be local to a process, we introduce auxiliary variables, which are allowed to appear in more than one process. This does not violate the distributed model because auxiliary variables are not needed for execution, but only for the proof of correctness.

With the addition of auxiliary variables, it becomes necessary to show that execution of other processes cannot interfere with the validity of assertions; this is the property of non-interference, first described in [14]. The notions of synchronously altered variables and universal assertions are introduced to simplify proof of non-interference for CSP; in fact, with certain restrictions, non-interference becomes a syntactic property of the program and its sequential and satisfaction proofs.

We use the notation \([P] S [Q]\) to mean that execution of \(S\) begun in a state satisfying \(P\) is guaranteed to terminate in a state satisfying \(Q\), provided deadlock does not occur. The notation \(P_{\vec{x}}\) denotes conventional textual substitution: the predicate \(P\) with all free occurrences of the \(x_i\)
simultaneously replaced by the $e_i$, where $\bar{X}$ is a list of different identifiers.

Why did we choose to use the Hoare-logic rather than, say, Dijkstra's weakest precondition logic or a trace logic? We were motivated by the desire to keep our proof rules as close to an existing proof system as possible. This would mean that experience gained from the study of sequential programs could be extended to programs written in CSP.

Trace logics are really more suited to proving facts about programs in general. They can be used to prove, for example, that the proof rule that we give for repetition ensures termination. We feel that, when applied to specific programs, trace logics introduce too much detail, causing the proofs to become unwieldy.

Dijkstra's weakest precondition logic posed other problems. As will be seen, the rule for communication commands violates the Law of the Excluded Miracle; this means that our rules would be unsound and hence incompatible with Dijkstra's system. Additionally, there are problems in defining the weakest precondition for the parallel command. Normally, the weakest precondition for a composite command is a function of the weakest precondition of its component parts; for parallel constructs, this does not hold.

There are problems associated with Hoare-logic. Can a Hoare triple $(\{P\} S \{Q\})$ be manipulated in the same way that an assertion in the predicate calculus can? Is $\neg(\{P\} S \{Q\})$ a meaningful expression? We direct the interested reader to [8] for a discussion of these and related problems.
3.1 Axioms

skip: \( \{P\} \text{skip} \{P\} \)

assignment: \( \{P\} \bar{x} := c \{P\} \)

communication: \( \{P\} A.T(\bar{x}) \{Q\} \)

The axioms for skip and assignment are the conventional ones. The notation \( P_{\bar{x}} \) is usually used only when \( \bar{x} \) is a list of different identifiers, but in [8] it is extended to allow the \( x_i \) to be array elements and records so that the assignment axiom given above holds in general. For the purposes of this thesis one may use the former view and restrict multiple assignment to different, simple without loss of understanding.

We can explain the communication axiom as follows. Remember that \( \{P\} S \{Q\} \) means total correctness in the absence of deadlock and that sequential proofs only prove facts about execution of processes in isolation. With only one process being executed, communication commands necessarily deadlock; thus, anything be may assumed to be true upon termination of a communication command because termination never occurs!

The communication axiom allows proofs like

\( \{\text{true}\} A.T(\bar{x}) \{\text{false}\} \) \hspace{1cm} (3.1.1)

which implies that after execution, false is true. When processes are combined in parallel, the previous argument will no longer hold, because deadlock is no longer guaranteed. It is such proofs that the Law of the Excluded Miracle [5] was designed to avoid. A satisfaction proof will eliminate proofs of triples like (3.1.1) where deadlock will not occur.
1.2 Inference rules

sequence

\[ \{P\} S_1 \{Q\} \cdot \{Q\} S_2 \{R\} \]

\[ \{P\} S_1 ; S_2 \{R\} \]

\[ P \Rightarrow P' \cdot \{P'\} S \{Q'\} \cdot Q' \Rightarrow Q \]

consequence

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

\[ P \Rightarrow (\exists i : b_i) \]

\[ (\forall i : \{P \land b_i\} c_i ; S_i \{Q\}) \]

alternation

\[ \{P\} \textbf{if } b_1 ; c_1 + S_1 \textbf{ if } \cdots \textbf{ if } b_n ; c_n + S_n \textbf{ fi } \{Q\} \]

\[ (P \land (\exists i : b_i)) \Rightarrow t > 0 \]

\[ (\forall i : \{P \land b_i\} T := t ; c_i ; S_i \{P \land t < T\}) \]

repetition

\[ \{P\} \textbf{ do } b_1 ; c_1 + S_1 \textbf{ do } \cdots \textbf{ do } b_n ; c_n + S_n \textbf{ od } \{P \land (\forall i : \neg b_i)\} \]

where \( t \) is an integer function and \( T \) a fresh variable.

parallel

\[ (\forall i : \{P_i\} S_i \{Q_i\}) \text{ satisfied and interference-free} \]

\[ ((\forall i : P_i) [A_1 \vdash S_1 || \cdots || A_n \vdash S_n] \{(\forall i : Q_i)\}) \]

The inference rules for sequence and consequence are the conventional ones and will not be discussed here. The rules for alternation and repetition are also familiar, except that the hypothesis corresponding to a
guarded command now includes execution of a command in the guard. 1 The careful reader will note that the hypotheses for

$$\text{if } b_1; c_1 + S_1 \sqcap \ldots \sqcap b_n; c_n + S_n \text{ fi}$$

(3.2.1)

are the same as the hypotheses for

$$\text{if } b_1 + c_1; S_1 \sqcap \ldots \sqcap b_n + c_n; S_n \text{ fi}.$$  

(3.2.2)

How then do (3.2.1) and (3.2.2) differ? Actually, in the absence of deadlock there is no difference. In the case of (3.2.2), if alternative i is chosen (because $b_i$ is true), either command $c_i$ will execute and all is well (i.e. as it would be if this alternative were chosen in (3.2.1)) or $c_i$ will not execute and the system deadlocks. Obviously the difference is that (3.2.2) is more prone to deadlock; this is analyzed more formally in chapter 4.

It is clear that the rule for the parallel command implies that each component is executed, under the assumption that free variables of $P_i$ and $Q_i$ are limited to the local variables of $A_i$. The necessary introduction of shared auxiliary variables brings the need for non-interference proofs, so that our model of execution and the inference rule are still consistent. For a similar rule, see [14].

Technically, a sequential proof consists of a list of statements (either Hoare triples or statements of the predicate calculus), each of which is either an instance of an axiom or the conclusion of an inference rule with all the hypotheses of that rule preceding it in the list. This form is awkward and instead one usually gives an annotated program. In an annotated program, assertions are placed before and after the program (corresponding to the input-output specifications) and between the commands

1 The proof of the hypothesis may include an instance of the axiom for communication commands, necessitating a satisfaction proof.
of the program. We require that an assertion imply the precondition of the following command and, in turn, be implied by the postcondition of the preceding command. This convention allows us to include only one of two assertions where the rule of consequence is applied.

The functions pre and post are applied to commands and have as values, respectively, the assertion preceding and succeeding the command in the annotated program. At times we will only include the invariant of a loop once; the reader should consider the invariant to be the postcondition of each alternative. Usually the invariant is also chosen to be the precondition of the loop. Similarly, the postcondition of an alternative command is the postcondition of each of the alternatives and is not repeated.

In the case of commands in guards, the precondition is the conjunction of the precondition of the alternation (or the invariant of the repetition) and the boolean part of the guard; the postcondition of the guard is the precondition of the command that is guarded.

As a brief example consider, figure 3.2.3. Figure 3.2.3a is an annotated program, figure 3.2.3b is the list of Hoare triples and implications that corresponds to the annotated program.

1.1 Satisfaction proof

Consider any communication command S and its pre- and postconditions from a sequential proof. The communication axiom allows any postcondition because in isolation deadlock is inevitable. Now we are concerned with combining processes. When processes are run in parallel, deadlock is not inevitable and we must show that the assertions are still satisfied.

Suppose, then, that a matching communication pair appears in
\[ \{ x = a \land y = b \} \quad t := x \quad \{ t = a \land y = b \} \]
\[
\{ t = a \land y = b \} \quad x := y \quad \{ t = a \land x = b \} 
\]
\[
\{ t = a \land x = b \} \quad y := t \quad \{ y = a \land x = b \} 
\]
\[
\{ y = a \land x = b \} \quad \Rightarrow \quad \{ x = b \land y = a \} 
\]

(a)

\[
\{ x = a \land y = b \} \quad t := x ; \quad x := y ; \quad y := t \quad \{ x = b \land y = a \} 
\]

(b)

Figur e 3.2.3 -- An annotated program and corresponding proof

processes A and B, as follows:

\[
[ B :: \ldots \{ P \} \text{A} \cap \text{T}(\overline{x}) \{ Q \} \ldots ] | [ A :: \ldots \{ R \} \text{B} ! \cap \text{T}(\overline{x}) \{ S \} \ldots ] \quad (3.3.1)
\]

Should these two commands communicate, the effect would be equivalent to \( \overline{x} = \overline{e} \). Hence, \( (Q \land S) \) is true after communication if and only if \( (Q \land S) \overline{x} \overline{e} \) is true before. Before communication, both preconditions are true and we may assert \( (P \land R) \). Therefore, postconditions \( Q \) and \( S \) are satisfied if and only if

\[
(P \land R) \Rightarrow (Q \land S) \overline{x} \overline{e} \quad (3.3.2)
\]

The Rule of Satisfacton is that every matching pair of the form (3.3.1) must satisfy (3.3.2).

This fills the gap left by our violation of the Law of the Excluded Miracle. Consider, for example, the use of the communication axiom that caused us concern, (3.1.1). A matching receive would have the form

\[
\{ R \} \text{B} ! \cap \text{T}(\overline{x}) \{ S \} \quad (3.3.3)
\]
and we would be obliged to prove

\[(\text{true} \land R) \Rightarrow (\text{false} \land S)^{\overline{R}}_{\overline{E}} \quad \text{(or } R = \text{false})\]

If we can prove \( \neg R \), \( B!T(\overline{E}) \) can never be prepared to communicate, and the matching pair will never be executed. (In general, if \( \neg (P \land R) \) can be proved, then (3.3.2) is trivially satisfied, regardless of postconditions. This covers the cases where the logic of the program prevents the pair from communicating.) This formalizes the intuitive notion that communication is a distributed form of assignment.

Looking at (3.3.1) and (3.3.2), we see that with the initial assumption that variables be accessible in exactly one process, \( S = S^{\overline{X}}_{\overline{E}} \) so that \( P \) can have no bearing on \( S \). The introduction of auxiliary variables, to be handled next, will provide shared variables without violating the distributed model.

1.4 Auxiliary variables

The addition of auxiliary variables to the proof system allows assertions in distinct processes to refer to non-disjoint state spaces. Auxiliary variables are needed in order to relate the program variables of one process to the program variables of another. Auxiliary variables are defined by Owicki [14] for use in proofs in the centralized model. Our definition is adapted to the distributed model.

An auxiliary variable may affect neither the flow of control of a process nor the value of any non-auxiliary variable. Hence, auxiliary variables are not necessary to the computation and may be omitted from the program -- but not the proof. These conditions are ensured if the following syntactic restrictions are met.
Auxiliary variables may appear only

1) in assertions,

2) in assignments where expressions that reference auxiliary variables are assigned to auxiliary variables, and

3) as parameters in receives and in expressions as parameters of sends, where the assignment that corresponds to each matching pair must satisfy condition (2).

When augmenting programs with auxiliary variables, one can add assignments and extend the parameters of communication commands. When adding parameters to communication commands, the set of matching pairs must remain the same, because a change could affect the flow of control, which is not allowed.

Auxiliary variables can help describe global relations; local variables, only approximations to them. Commonly, preconditions for communication commands will assert that a local variable equals a global auxiliary variable. When communication occurs, each process involved will have a local variable equal to the global auxiliary variable, and hence all three variables will be equal.

Figure 3.4.1 contains the bounded buffer example of chapter 2, augmented with auxiliary variables IN and OUT. They contain the sequences of messages sent from process I and received by O, respectively. Process I now sends e with "$B!(e, IN \circ e)"$; $O$ receives $x$ with "$B?(x, OUT)"$.

Besides the definitions given in Figure 2.5.1, the invariant of the loop of Figure 3.4.1 contains the following predicate to relate the input stream, the output stream, and the buffered messages:

$$\text{IN}=\text{OUT} \circ \text{ob[OUT]} + \cdots + \text{ob[OUT}(\text{count}-1)\text{].}$$

---

2 Operator "$\circ$" denotes catenation of an element to a sequence.
out, count, done := 0, 0, false;
& ~done \land count < N; I?(b[out\#count], IN) \quad \text{+ count := count + 1}
& count > 0; O!(b[out], OUTob[out]) \quad \text{+ out, count := out \text{-} 1, count - 1}
& ~done; \quad \text{I?quit} \quad \text{+ done := true}
&
& O!quit

Figure 3.4.1 -- A bounded buffer (with auxiliary variables)

1.5 Proof of non-interference

Without auxiliary variables there is no need to prove non-interference. With disjoint state spaces, execution of one process cannot affect the state of, nor the validity of assertions about, another process (except when communication occurs, and the satisfaction proof takes care of this case). However, with auxiliary variables it is possible for execution of one process to affect assertions about another. Here we adapt Owicki's notion of non-interference [14] to CSP.

Let us introduce some terminology. Command S is parallel to assertion P if S is contained in a process of a parallel command and P is contained in a different process of the same parallel command. A matching pair S and R is parallel to P if both S and R are parallel to P. Note that neither S nor R may appear in the same process as P.

We must show for every command S parallel to P, that

\[ [P \land \text{pre}(S)] S [P]. \]  \hspace{1cm} (3.5.1)

Similarly, we must show for every matching pair, S: A?T(x) and R: B?T(x), parallel to P that

\[ (P \land \text{pre}(S) \land \text{pre}(R)) \Rightarrow P_{\overline{x}}^x. \]  \hspace{1cm} (3.5.2)

(3.5.1) implies that P is invariant over execution of any command S that is parallel to P. (3.5.2) does the same for matching pairs; this is
the implication that would arise from the satisfaction proof for the matching pair

\[ \{ p \land \text{pre}(S) \} S : \text{ait}(\bar{w}) \{ p \} \]

\[ \{ p \land \text{pre}(R) \} R : \text{bit}(\bar{w}) \{ p \} \]

As given, the requirements for a proof of non-interference assume that assignment commands and communications are atomic actions; i.e. we need only concern ourselves with the effect of one operation at a time. Is this a reasonable assumption? What about the other commands?

Yes, the assumption is reasonable. The only problem we need worry about is whether simultaneous execution of two commands can have an effect different from sequential execution of the commands? Simultaneous changes to variables local to different processes cannot interact, because there would be no shared variables in the expressions being assigned or communicated. Simultaneous changes to shared variables could, of course, cause trouble. Fortunately, the only shared variables are auxiliary variables. We assume that assignments and communications that include auxiliary variables are atomic. This introduces no constraint on implementation because auxiliary variables are not included in the execution of the program; our assumption of atomicity is, as was the introduction of auxiliary variables, purely for the purpose of constructing proofs.

A proof of non-interference, if approached mechanically, is an awesome task. Every assertion in every process must be compared against every command in every other process and against every matching pair, so it takes time proportional to the product of the lengths of the processes. Fortunately, through judicious structuring of the program and careful selection of the assertions and auxiliary variables, it is possible to
reduce the amount of work needed. The following notions of synchronously altered variables and universal assertions are important in designing good proofs.

1.6 Synchronously altered variables

Variable v is synchronously altered in process A if the only occurrences of v, outside of expressions, are in

1) the left part of assignments in process A,
2) receive's in process A, and
3) receive's in any process, from process A.

The value of an expression that contains only variables synchronously altered in A cannot change except when A progresses. Hence, there is no interference with assertions in the proof of A, provided that these assertions contain only variables synchronously altered in A.

More formally, a non-interference proof consists of proving instances of (3.5.1) and (3.5.2). But the definition of "parallel to" allows us to conclude that none of these instances will contain commands of the type described above. All other types of commands trivially satisfy (3.5.1) and (3.5.2) and there is no interference.

In many cases, synchronously altered variables arise naturally. In the bounded buffer example, IN and OUT are both synchronously altered in B and, respectively, in I and O. Hence, assertions about IN may be made in proofs about both I and B and no interference proof is needed.

Unfortunately, it is sometimes difficult to express global relations in terms of synchronously altered variables. The assertion that OUT is a prefix of IN is fine in B, but subject to interference in I and O. The following notion of universal assertions is also convenient in limiting non-interference proofs.
1.7 Universal assertions

Some assertions can be shown, in the sequential and satisfaction proofs, to be true everywhere: initially, finally, and between every pair of commands in all processes. Such assertions are said to be universal.

Universal assertions are not subject to interference. Why? Consider the two cases of a non-interference proof. We must show that the universal assertion holds after execution of a command or communication, given that the precondition and the universal assertion held before execution. But we have already shown that the universal assertion held afterwards (that's why it's universal!) Hence the proof must exist and we need not show it explicitly.

1.8 Summary

Our proof rules are an extension of the conventional rules for the proof of sequential programs. We have added the Rule of Satisfaction, which explains the effects of both synchronization and communication.

We made two changes to Hoare's CSP. We added the possibility of including send in guards and removed automatic termination of loops.

The first change was motivated by the fact that our communication axiom does not distinguish between send and receive. In fact, we would have had to make a restriction in our rules to prevent output guards.

The second change was motivated by the difficulties encountered when we attempted to use a rule that included automatic termination. The rule allowed the postcondition of a repetition to include the postcondition of processes that it waited on. This resulted in very complex postconditions of processes. We were pleased when we found that reducing the power of CSP made proving programs easier. See the discussion at the end of chapter 5 for more about our experiences in this direction.
Chapter 4

Requirements for strong correctness

The cooperative nature of CSP introduces a problem that does not exist in sequential notations. It is now possible for a process to reach a point at which progress must wait for synchronization with another process.

A process waits when it is blocked. In and of itself, blocking is not bad. If, however, some process is blocked and no process is ready, then no progress can be made and blocking will not end; this is known as deadlock.

We give here the conventional method for proving freedom from deadlock (see e.g. [14]), but tailored to the needs of CSP. Basically, one shows for each possible "waiting state", a state in which all processes might be blocked or terminated, that at least one process can make progress -- in our terminology, that one process is ready. Note that in CSP blocking can only occur at communication commands, at commands with communicating guards, and at the end of processes in parallel commands. The main problem is defining the waiting states in a clear but precise fashion.

4.1 A sufficient condition for freedom from deadlock

A configuration $K$ of command $S$ corresponds to a possible waiting state during the execution of $S$: a state in which each process currently in execution may be either blocked or terminated. (These processes will be components of $S$.) A configuration consists of a condition $\text{cond}(K)$, which is a condition that must hold if the processes of $S$ are waiting with this configuration, and a set of guard-command pairs $\text{commands}(K)$, which is the set of commands that could be executed next and conditions under which they
would be ready. The parts of a guard-command pair \( p \) are a boolean guard \( b(p) \) and a command \( c(p) \); the command is either a communication command or \text{skip}, indicating a non-communication command. Figure 4.1.1 defines the set of configurations of a command \( S \); we make some comments about the definition here.

The set \( C(S) \) of configurations is a function of the sequential proof for a program, and not of the program itself. Thus, for each command \( S \) of the program, we also have the precondition \( \text{pre}(S) \), the postcondition \( \text{post}(S) \) (applied only to processes), the invariant \( \text{inv}(S) \) (applied only to repetitions), and pair \((r,s)\), which is true if and only if \( c(r) \) and \( c(s) \) form a matching pair.

Function \( \text{join} \) combines configurations. If \( a_i \) is a configuration for process \( A_i \), then \( \text{join} \left( \bigcup \_{i=1}^{n} a_i \right) \) is the configuration in which the \( A_i \) are all waiting with their corresponding configuration, \( a_i \).

Note the differences between the configurations for alternation and repetition. Before execution of an alternative command, the precondition is true and \( c_i \) may be executed if \( b_i \) is true. Before a repetition of a repetitive command, the invariant is true and, in addition to possibly selecting a guard as for an alternative command, the loop may terminate if all the guards are false (~BB).

To prove absence of deadlock in \( S \), it remains to show that each configuration in \( C(S) \) is such that either it cannot be entered during execution or it contains one ready process. This is expressed for a configuration \( K \) in definition \( \text{DLF} \) (DeadLock Free), given in (4.1.2) below. The first disjunct asserts that the conjunction of the preconditions is false, which means that the corresponding state couldn't be entered during execution, the second disjunct asserts that a non-communication command of
1) A configuration has the form \( \langle \text{cond; commands} \rangle \).

2) A guard-command pair has the form \((b \rightarrow c)\).

3) \(BB = (\exists i: i \in 1..n: b_i)\)

4) \(\text{join}(u \{a_i\}) = (\forall i: i \in 1..n: \text{cond}(a_i)); u \text{ commands}(a_i)\) \[
\text{cond}(a_i) = \{i \in 1..n\}
\]

\[
\begin{array}{ll}
S & C(S) \\
\text{skip} & \emptyset \\
\overline{x} := c & \emptyset \\
A,T(\overline{x}) & \{[\text{pre}(S): \{\text{true } S\}]\} \\
S_1 ; \cdots ; S_n & u \ C(S_i) \ i \in 1..n \\
\text{if } b_i; c_i \rightarrow S_i & \{[\text{pre}(S): u \ \{(b_i \rightarrow c_i)\}] \cup (u \ C(S_i)) \ i \in 1..n\} \\
\Box \cdots \Box b_n; c_n \rightarrow S_n \text{ fi} \\
\text{do } b_i; c_i \rightarrow S_i & \{[\text{inv}(S): \{\neg BB + \text{skip}\}] \cup (u \ \{(b_i \rightarrow c_i)\}]\} \ i \in 1..n\} \\
\Box \cdots \Box b_n; c_n \rightarrow S_n \text{ od} & u (u \ C(S_i)) \ i \in 1..n\} \\
[A_1; S_1 & \{\text{join}(u \ \{a_i\}) \ | \ a_i \in C(S_i) \cup \{\text{post}(S_i): \emptyset\} \} \\
\| \cdots \| A_n; S_n & \{[\forall i: i \in 1..n: \text{post}(S_i): \emptyset]\} \\
\end{array}
\]

\textbf{Figure 4.1.1 -- Configuration sets for commands}
the configuration can be executed, and the third asserts that a matching
pair can be executed.

\[ DLF(K) : \neg \text{cond}(K) \lor (\exists c \in \text{command}(K) : c(s) = \text{skip} : b(s)) \lor \]
\[ (\exists r, s \in \text{command}(K) : \text{pair}(r, s) : b(r) \land b(s)) \quad (4.1.2) \]

Proving that DLF is satisfied for all \( K \) in \( C(S) \) can require work on
the order of the product of the lengths of the processes in \( S \). Often,
however, by judicious choice of the structure of the program and of the
sequential proofs of the processes, one can reduce the analysis to a few
simple cases.

4.2 Reducing the size of \( C(S) \).

Many times there are communication pairs where both send and receive
are local to a parallel command PARA that is not the entire program PROG
under consideration. Because we are only interested in showing that all
configurations in \( C(\text{PROG}) \) satisfy DLF, we can justify removing a
configuration from \( C(\text{PARA}) \) by proving that removal will not affect our
results. When applied to composite commands, \( C \) either unions or joins the
configuration sets of the component commands. Obviously, if configuration
\( x \) satisfies DLF, then the answer to the question of whether all members of
a set of configurations satisfy DLF is unaffected by including \( x \) in the
set. The following theorem proves that this holds for the function join as
well. Together, this means that we can remove from \( C(\text{PARA}) \) any
configuration that satisfies DLF.

**Theorem 4.1**

If configuration \( x \) satisfies DLF, then, for any configuration \( y \),
\( \text{join}([x, y]) \) satisfies DLF.
**Proof**

1) Assume that $x = [P; X]$. Then, from the hypothesis,

$$P \Rightarrow (\exists s \in X: c(s) = \text{skip}; b(s)) \vee (\exists s \in X: \text{pair}(r, s); b(r) \wedge b(s))$$

2) But from this we conclude, for any $Q$ and $Y$, that

$$(P \land Q) \Rightarrow (\exists s \in X \cup Y: c(s) = \text{skip}; b(s)) \vee (\exists s \in X \cup Y: \text{pair}(r, s); b(r) \wedge b(s)).$$

3) In particular, for $y = [Q; Y]$, $\text{join}([x, y])$ satisfies DLF.

**4.1 An awaited explanation**

With the use of theorem 4.1, we can now explain the difference between the commands (4.3.1) and (4.3.2).

$$\text{if } b_1; c_1 + S_1 \bigcirc \ldots \bigcirc b_n; c_n + S_n \text{ fi} \quad (4.3.1)$$

$$\text{if } b_1 + c_1; S_1 \bigcirc \ldots \bigcirc b_n + c_n; S_n \text{ fi}. \quad (4.3.2)$$

In section 3.2, we mentioned that the inference rules for (4.3.1) and (4.3.2) have the same hypotheses, but that (4.3.2) is more prone to deadlock. By considering $C((4.3.1))$ and $C((4.3.2))$ we can formalize this intuitive notion. See figures 4.3.3a and 4.3.3b. Assume that the precondition for both commands is $P$ and, because of the hypotheses for the alternative command, that $P \Rightarrow BB$.

$C(S_i)$ is common to both sets of configurations and cannot be the source of their difference. Because $P \Rightarrow BB$, the configuration $[P: [(b_i + \text{skip}) | i \in 1..n]]$ satisfies DLF and can be removed from $C((4.3.2))$. Hence, the question comes down to: how do (4.3.4) and (4.3.5) differ?
\[ C((4.3.1)) = \{ [P: \{(b_i + c_i) \mid i \in \ldots n\}] \mid \iota \in \ldots n \} \cup (u \cup C(S_i)) \]

*Figure 4.3.3a -- Configuration set for 4.3.1*

\[ C((4.3.2)) = \{ [P: \{(b_i + \text{skip}) \mid i \in \ldots n\}] \mid \iota \in \ldots n \} \cup \{ \} \cup C(S_i) \]

*Figure 4.3.3b -- Configuration set for 4.3.2*

\[ [P: \{(b_i + c_i) \mid i \in \ldots n\}] \]

\[ [P \land b_i : (\text{true} + c_i)] \mid i \in \ldots n \]

*(4.3.4)  (4.3.5)*

**Theorem 4.2**

For any configuration \( x = [Q:X] \), if, for all \( a \in (4.3.5) \), \( \text{join}([a, x]) \) satisfies DLF, then \( \text{join}((4.3.4) \cup \{x\}) \) satisfies DLF.

**Proof**

1) By assumption,

\[ (\forall i: i \in \ldots n: (P \land b_i \land Q) \Rightarrow (\exists \iota \in [c_i]: \lambda X: c(s) = \text{skip}: b(s)) \]

\[ \lor (\exists r, \iota \in [c_i]: \lambda X: \text{pair}(r, s): b(r) \land b(s)). \]

2) This is equivalent to

\[ (P \land Q) \Rightarrow (\forall i: i \in \ldots n: b_i \lor (\exists \iota \in [c_i]: \lambda X: c(s) = \text{skip}: b(s)) \]

\[ \lor (\exists r, \iota \in [c_i]: \lambda X: \text{pair}(r, s): b(r) \land b(s)). \]

3) But \( P \Rightarrow BB \). Choose \( i_0 \) such that \( b_{i_0} \) is true, under the assumption \( P \). Then

\[ (P \land Q) \Rightarrow (\exists \iota \in [c_i]: \lambda X: c(s) = \text{skip}: b(s)) \]

\[ \lor (\exists r, \iota \in [c_i]: \lambda X: \text{pair}(r, s): b(r) \land b(s)). \]

4) This in turn implies that
\[(P \land Q) \Rightarrow (\exists s \in \{c_i\} \forall x : c(s) = \text{skip}; b(s))
\]
\[
\lor (\exists s \in \{c_i\} \forall x : \text{pair}(r, s) : b(r) \land b(s))
\]

or that \(\text{join}((4.3.4)u(x))\) satisfies DLF.

This shows that (4.3.2) can be replaced by (4.3.1) in any program. That the converse is not true, and hence that (4.3.2) is more prone to deadlock, is easily seen from an example.

Consider the program in figure 4.3.6a. The first alternative will be selected and the program will terminate normally. Now consider the program in figure 4.3.6b. There is now no reason why the first alternative should be selected and should the second be selected, deadlock will occur.

\[A:: \text{if true; B!1 + skip \[ true; B?a + skip \]} \| B :: A?b\]

Figure 4.3.6a -- Deadlock free

\[A:: \text{if true + B!1 \[ true + B?a \]} \| B :: A?b\]

Figure 4.3.6b -- Not deadlock free
Chapter 5

An example -- finding the minimum of a set.

In this chapter we tie together the previous three chapters by means of an extended example. We present a program, its proof of weak correctness, a formal proof of freedom from deadlock, and an informal proof of freedom from deadlock that draws on ideas from the formal proof.

5.1 The program

Process B should receive from process A the minimum of the set \{a_i | i \in 1..N\}. Define process A to be the parallel execution of N processes Min(i), i \in 1..N, as shown in figure 5.1.1.

A:: [ || Min(i)]
   i \in 1..N

Min(i):: integer my_min, their_min, my_size, their_size;
     my_min, my_size := a_i, 1;

do  [] 0 < my_size < N; Min(j)!(my_min, my_size)
   j \in 1..N \& i \neq j
      + my_size := 0

do  [] 0 < my_size < N; Min(j)?(their_min, their_size)
   j \in 1..N \& i \neq j
      + my_min, my_size := min(my_min, their_min), my_size+their_size
od;

if my_size = 0 + skip
   [] my_size = N + B!my_min
fi

Figure 5.1.1 -- Min
Initially, each process Min(i) is responsible for the value \( a_i \). As execution progresses, an executing process Min(i) is responsible for the minimum of a set of values, the number of values in this set being \( m \) size. Min(i) tries to relieve itself of this responsibility by sending the minimum of the set, together with the size of the set, to another process Min(j) (say). If successful, the set of values for which Min(i) is responsible becomes empty, Min(i) terminates, and Min(j) becomes responsible for the minimum of the union of its own set and Min(i)'s. Execution continues until \( N-1 \) processes have become not responsible and one process is responsible for the minimum of the set of all values. This process then gives complete responsibility to B by sending B the minimum value.

To prevent deadlock, which would occur if all executing processes refused to receive, each process must agree to take on additional responsibility. Min(i) must continue to send or receive until it has become responsible for either the null set or a set of size \( N \). In the second case, the minimum value that Min(i) has received is the minimum of \( \{a_i \mid 1 \leq i \leq N\} \), and it sends this value to process B.

This explanation gives the flavor of the processes, but does not really provide sufficient information for answering questions about termination, deadlock or even partial correctness.

5.2 Sequential proof

First we give a formalization of the ad hoc description of Min. Given the formal description, it is fairly straightforward to see that Min is weakly correct. The annotated program is given in figure 5.2.3.

To remove concerns about repetitions in the set \( \{a_i\} \), we deal with sets over the range 1 to \( N \), corresponding to the elements \( a_1, a_2, ..., a_N \).
For each process Min(i) define auxiliary variable set_i as the indices of the values for which Min(i) is "responsible". Therefore, my_min is the minimum and my_size the size of set_i for which Min(i) is responsible. When Min(i) no longer has responsibility, set_i = \emptyset and my_size=0. Furthermore, let auxiliary variable set_0 contain the set for which process B is responsible.

MIN(my_min, my_size, set_i) is an invariant of the loop of each process, where MIN is defined by

\[ \text{MIN}(\text{mn}, \text{sz}, S) \iff (sz = |S| \land (S = \emptyset \lor \text{mn} = \min_{j \in S} a_j)) \quad (5.2.1) \]

Predicate UNION, given in (5.2.2), expresses the fundamental property that exactly one process is responsible for each a_i. It is universally true, i.e. initially, finally, and between any pair of commands in all processes. UNION will not be repeated at each assertion; it should be kept in mind.

\[ \text{UNION:} \quad \bigcup_{i \in 0..N} \text{set}_i = 1..N \land (\forall i,j: 0 \leq i,j \leq N: \text{set}_i \cap \text{set}_j = \emptyset) \quad (5.2.2) \]

The variant function t used to prove termination is (\#i: set_i \neq \emptyset). It is non-negative, non-increasing, and decreases with each message sent.

5.1 Satisfication proof

To prove satisfaction, we must show that (3.3.2) holds for each matching pair of the form (3.3.1). Examination of the program reveals two classes of matching pairs:

---

1 An expression of the form (\#i: P(i)) denotes the number of values i for which P(i) is true.
\[ \forall i \leq N: \text{set}_i = \{i\} \land \text{set}_0 = \emptyset \]

\[ B:: \{\text{MIN}(0, 0, \text{set}_0)\} \]
\[ \text{if } \square \text{MIN}(i)?(m, \text{set}_0, \text{set}_i) \Rightarrow \text{skip} \]
\[ \land \forall i < N: \text{set}_i \]
\[ \{\text{MIN}(m, N, \text{set}_0)\} \]

\[ A:: [ \]
\[ \land \{ \forall i \leq N: \text{set}_i = \{i\} \land \text{MIN}(i)\} \{\text{MIN}(0, 0, \text{set}_i)\} \]  
\[ \land \forall i \leq N \}

\[ \{\forall i: \text{set}_i = \{i\}\} \]

\[ \text{MIN}(i):: \]
\[ \{\text{set}_i = \{i\}\} \]
\[ \text{my}\_\text{min}, \text{my}\_\text{size} := a_i, 1; \]

\[ \{\text{MIN}(\text{my}\_\text{min}, \text{my}\_\text{size}, \text{set}_i)\} \]

\[ \text{do} \]
\[ \land \quad 0 < \text{my}\_\text{size} < N; \text{MIN}(j)?(\text{my}\_\text{min}, \text{my}\_\text{size}, \text{set}_i \cup \text{set}_j, \emptyset) \]
\[ \land \forall j < N \land \forall j \leq N \}
\[ \{\text{MIN}(\text{my}\_\text{min}, \text{my}\_\text{size}, \text{set}_i \cup \text{set}_j, \emptyset)\} \]
\[ \text{my}\_\text{size} := 0 \]

\[ \land \quad 0 < \text{my}\_\text{size} < N; \text{MIN}(j)?(\text{their}\_\text{min}, \text{their}\_\text{size}, \text{set}_i \cup \text{set}_j, \emptyset) \]
\[ \land \forall j < N \land \forall j \leq N \}
\[ \{\text{MIN}(\text{my}\_\text{min}, \text{their}\_\text{min}, \text{my}\_\text{size} \land \text{their}\_\text{size}, \text{set}_i)\} \]
\[ \text{my}\_\text{min}, \text{my}\_\text{size} := \text{min}(\text{my}\_\text{min}, \text{their}\_\text{min}), \text{my}\_\text{size} \land \text{their}\_\text{size} \}
\[ \text{od}; \]

\[ \{\text{MIN}(\text{my}\_\text{min}, \text{my}\_\text{size}, \text{set}_i) \land (\text{my}\_\text{size} = 0 \lor \text{my}\_\text{size} = N)\} \]

\[ \text{if } \text{my}\_\text{size} = 0 \Rightarrow \text{skip} \{\text{MIN}(\text{my}\_\text{min}, 0, \text{set}_i)\} \]
\[ \land \text{my}\_\text{size} = N \Rightarrow \{\text{MIN}(\text{my}\_\text{min}, N, \text{set}_i)\} \]
\[ \{\text{MIN}(\text{my}\_\text{min}, 0, \text{set}_i)\} \]

**Figure 5.2.3 -- Min (annotated)**
Hi: Min(i)! (my_min, my_size, set_i, set_j, $\emptyset$) occurring in Min(j).

Mj: Min(j)?(their_min, their_size, set_i, set_j) occurring in Min(i)

and

B!(my_min, set_i, $\emptyset$) occurring in Min(i),

Min(i)?(m, set_0, set_i) occurring in B.

Considering the first pair Hi, Mj, and priming local variables of Min(j) to distinguish them from those of Min(i), we must show that

\((\text{pre}(\text{Hi}) \land \text{pre}(\text{Mj})) \Rightarrow (\text{post}(\text{Hi}) \land \text{post}(\text{Mj})) \land \text{their_min, their_size, set_i, set_j})\)

\((\text{my_min}', \text{my_size}', \text{set_i}, \text{set_j}, \emptyset)\)

where \(\text{pre}(\text{Hi}) = 0 < \text{my_size}' < \text{N} \land \text{MIN}(\text{my_min}', \text{my_size}', \text{set_i}) \land \text{UNION}\)

\(\text{pre}(\text{Mj}) = 0 < \text{my_size} < \text{N} \land \text{MIN}(\text{my_min}, \text{my_size}, \text{set_i}) \land \text{UNION}\)

\(\text{post}(\text{Hi}) = \text{Min}(\text{my_min}', 0, \text{set_i}) \land \text{UNION}\)

\(\text{post}(\text{Mj}) = \text{MIN}(\text{min}(\text{my_min}, \text{their_min}), \text{my_size}+\text{their_size}, \text{set_i}) \land \text{UNION}\)

This straightforward exercise is left to the reader.

For the second matching pair, looking at the annotated proof in figure 5.2.3 we see that satisfaction holds if

\((\text{Min}(0, 0, \text{set}_0) \land \text{MIN}(\text{my_min}, \text{N}, \text{set_i}) \land \text{UNION})\)

\(\Rightarrow (\text{MIN}(m, \text{N}, \text{set}_0) \land \text{Min}(\text{my_min}, 0, \text{set_i}) \land \text{UNION})\)

\((\text{m, set_j, set_i}) \land \text{my_min, set_i, $\emptyset$})\)

which obviously holds.

\subsection*{5.4 Non-interference proof}

The auxiliary variables in the proof are: \(\text{set}_0, ..., \text{set}_N\). Variable \(\text{set}_i\) (for \(i \in \{1 .. N\}\)) is altered in two types of communications: in the receiving guards of process Min(i) and in the guards of Min(j) that receive
from Min(i). Thus, set\_i is synchronously altered in Min(i). Variable set\_0 is only altered in a receive in B and so is synchronously altered in B. The only auxiliary variable referred to in assertions in Min(i) (except for the universal assertion UNION) is set\_i; hence, assertions in this proof refer only to synchronously altered variables and there is no interference.

5.5 Proof of freedom from deadlock

In this section we present a formal proof that Min is free from deadlock; in the next section we give an informal proof based on the same reasoning. We would expect most people to use the informal style of reasoning; however, the formal is needed to support the informal.

In our proof we will refer to variables that have the same name, but belong to different processes. To identify a variable of process Min(i), we superscript the variable with the index i; e.g. my\_size\_i.

We first consider C([\parallel Min(i)]), because most configurations in i\in\ldots\ N
this set satisfy DLP. The configurations for Min(i), M\_i\_1, M\_i\_2, and M\_i\_3, are given in figure 5.5.1.\footnote{UNION is also a conjunct of the condition of all these configurations (remember that UNION is universal). In M\_i\_1, we have used the command Min(j).x to represent both send and receive with arbitrary parameters. This is all right because we know that the types of the parameters agree.}

\begin{align*}
M\_i\_1: & \text{MIN}(\text{my\_min, my\_size, set\_i}) \\
& \{((\text{my\_size}=0 \lor \text{my\_size}=N \rightarrow \text{skip}) \\
& \lor (0<\text{my\_size}<N \rightarrow \text{Min}(j).x)) \} \\
& j\in\ldots\ N \land i\neq j
\end{align*}

\begin{align*}
M\_i\_2: & \text{MIN}(\text{my\_min, N, set\_i}): ((\text{true} + \text{B!(my\_min, set\_i, \#))}) \\
M\_i\_3: & \text{MIN}(\text{my\_min, 0, set\_i}): \#
\end{align*}

Figure 5.5.1 -- Configurations M\_i\_1, M\_i\_2, and M\_i\_3
The set $C([i \mid \text{Min}(i)])$ is formed by joining configurations from each $i \in 1..N$

$\text{Min}(i)$. We consider, in turn, the possible ways of forming these configurations.

1) Any configuration joining $M_i^1$ and $M_j^1$, where $i \neq j$, must satisfy DLF.

Either $\text{my\_size}_i^1 = 0 \lor \text{my\_size}_j^1 = N$ (or similarly for $j$), in which case there is non-communication progress; or $0 < \text{my\_size}_i^1 < N \land 0 < \text{my\_size}_j^1 < N$, in which case there is communication progress.

2) Any configuration that includes in its join $M_i^1$, for exactly one $i$, satisfies DLF.

If $\text{my\_size}_i^1 = 0 \lor \text{my\_size}_i^1 = N$, there is non-communication progress.

In the case that $0 < \text{my\_size}_i^1 < N$, we know $set_i \neq \emptyset$ and, from UNION,

$(\forall j: j \in 1..N: set_j \neq 1..N)$.

a) If the configuration includes $M_j^2$, it satisfies DLF because

$\text{cond}(M_j^2) \Rightarrow set_j = 1..N$, which contradicts

$(\forall j: j \in 1..N: set_j \neq 1..N)$.

b) If the configuration includes $M_j^3$ for $j \neq i$, it satisfies DLF because $\text{cond}(M_j^3) \Rightarrow (set_j \neq \emptyset \lor set_j = 1..N)$. Along with

$(\forall j: j \in 1..N: set_j \neq 1..N)$ we conclude $(\forall j: j \in 1..N \land j \neq i: set_j = \emptyset)$

and, from UNION, $set_i = 1..N$ which contradicts our assumption.

3) Any configuration that includes $M_i^2$ and $M_j^2$ for $i \neq j$ satisfies DLF.

$\text{cond}(M_i^2) \land \text{cond}(M_j^2) \land i \neq j \land \text{UNION}$ yields a contradiction.

4) No configuration in the set

$\text{join}(\{M_i^2, M_j^3 \mid j \in 1..N \land j \neq i\} \mid i \in 1..N)$ (5.5.2)

satisfies DLF. We will consider these later.
Chapter 6
An operational model
and proof of consistency

We present an operational model of CSP and show how the assertions in
an annotated program are related to the program state. Trees are used to
represent current points of control in executing processes; states
represent memory. Operation Next models execution of a program by changing
trees and states.

Trees have nodes identified with either a command, the end of a
process (denoted "end_A" for process A), or the special mark "||". We
refer to the command as if it were the node, e.g. "the node $x := \bar{e}$" instead
of "the node identified with $x := \bar{e}$". The leaves of the tree are the
commands that may be executed next.

A state is a function that maps identifiers to values. (If necessary
a process name may be used to distinguish between variables with the same
name.) The value of an expression $e$ in program state $PS$ is denoted by
$e[PS]$. We use the notation $(PS; x := \bar{e}[PS])$ to represent the state that
differs from $PS$ only for variables $x$, which have values $\bar{e}[PS]$. This new
state corresponds to the result of executing the assignment $x := \bar{e}$,
beginning in state $PS$.

To simplify our discussion, we assume that
1) a null boolean guard is replaced by true,
2) null command guard is replaced by skip, and
3) a simple communication command $A.T(x)$ (i.e. a communication
command that does not appear in a guard) is replaced by
if true; $A.T(x) + \text{skip fi}$. 

42
The initial tree consists of two nodes. The root is identified with the end of the program -- "end_program"; its child is identified with the program text with the above changes. The initial state maps all identifiers to undefined except for those identifiers for which initial values are known. This state must satisfy the precondition of the program.

Operation Next selects from tree T either a leaf, a pair of matching leaves (corresponding to a matching pair), or a subtree with root "||" and children that are identified with the end of processes. These are replaced by a tree (trees), which may be empty; this operation produces T'. This replacement corresponds to transfer of control. At the same time, the program state PS is changed to a new program state PS'. We show that PS satisfies the conjunction of the preconditions of the leaves, which we call PRE(T). The precondition of "end_A", for process A, is the postcondition of A.

6.1 The model

In most cases, Next does not change the state PS; we consider these cases first.

1) A leaf $S_1; \cdots; S_n$ (where $1 \leq n$) may be replaced by a tree with root $S_2; \cdots; S_n$ and child $S_1$.

2) A leaf $[A_1::S_1||\cdots||A_n::S_n]$ may be replaced by a tree with root "||" and children "end_A_i" for $1 \leq i \leq n$; each "end_A_i" has child $S_i$.

3) A subtree consisting of root "||" and children "end_A_i" (which are leaves), for $1 \leq i \leq n$, may be removed. This corresponds to termination of execution of a parallel command.

---

1 In the case of the repetitive command, we use the invariant rather than the precondition.
4) A leaf skip may be removed.

5) A leaf do b_1; c_1 + S_1 [ [ ... [ b_n; c_n + S_n od may be removed if and only if -¬(b_1 v ... v b_n)(PS); otherwise

if b_1; c_1 + S_1 [ [ ... [ b_n; c_n + S_n fi may be made the child of this node.

6) A leaf if ... [ b_i; skip + S_i [ [ ... fi may be replaced by S_i if and only if b_i(PS).

In the two remaining cases, PS may change.

7) A leaf x:=x may be removed. PS' = (PS; x:=x(PS)).

8) Two leaves if ... [ b_i; A!T(x) + S_i [ [ ... fi and

if ... [ b_j; B?T(x) + S_j [ [ ... fi may be replaced, respectively, by S_i and S_j, if (b_i ^ b_j)(PS) and A!T(x) and B?T(x) are a matching pair. PS' = (PS; x:=x(PS)).

4.2 Consistency with the model

We now show that PRE(T)[PS] always holds. To this purpose, we also show that when a subtree becomes empty, the postcondition of the commands corresponding to the subtree becomes true. Generally, induction will be on the syntactic complexity of the commands. Repetition is not amenable to this treatment and is handled separately.

In cases 1 through 6, PS does not change. Therefore if PRE(T)[PS] is true, we know that the conjuncts of PRE(T') that correspond to the unchanged leaves of T' are satisfied by PS' and we need only check that the preconditions of the new leaves are satisfied.

1) The new precondition pre(S_0) equals the old precondition pre(S_1; ...; S_n) and so is satisfied. Inductively, after execution
of $S_1$, post($S_1$) will be true. From the sequential proof, we know

$$\text{post}(S_1) \Rightarrow \text{pre}(S_2; \cdots; S_n);$$

hence, after execution of $S_2; \cdots; S_n$, post($S_1; \cdots; S_n$) holds.

2) From the sequential proof we have

$$\text{pre}(\{A_1::S_1\} || \cdots || A_n::S_n) \Rightarrow \text{pre}(S_1) \land \cdots \land \text{pre}(S_n).$$

After execution of $S_1$, post($S_1$) holds, implying pre("end $A_1$").

This assumes we can prove that non-interference implies that assertions about one process are unaffected by execution of commands in other processes. See steps 7 and 8.

3) pre("end $A_1$") $\land \cdots \land$ pre("end $A_n$") equals post($S_1$) $\land \cdots \land$ post($S_n$),

which in turn implies post($\{A_1::S_1\} || \cdots || A_n::S_n$).

4) pre(skip) $\Rightarrow$ post(skip).

5) If do $b_1; c_1 + S_1 \square \cdots \square b_n; c_n + S_n$ od is removed, the invariant and $\neg(b_1 \lor \cdots \lor b_n)$ (=$\neg$BB) are true in PS, implying the postcondition of the repetition; otherwise

if $b_1; c_1 + S_1 \square \cdots \square b_n; c_n + S_n$ fi is made the child of the repetition and the precondition of the alternation (the invariant and BB) is satisfied. After execution of the alternation the invariant is once again true. Additionally, execution of the alternation decreases the value of an integer function that is bounded below if the invariant and BB are true; this we know from the sequential proof. Hence, this replacement cannot be made infinitely often and eventually the repetition is removed.

6) if $\cdots \square b_i; \text{skip} + S_i \square \cdots \text{fi}$ is replaced by $S_i$ only if $b_i$[PS].

In this case, pre(if $\cdots \square b_i; \text{skip} + S_i \square \cdots \text{fi}) \land b_i$ is true,

which implies pre($S_i$). After execution of $S_i$, post($S_i$)
( \Rightarrow \text{post}(\text{if ... } \Box b_i; \text{skip} + S_i \Box ... \text{fi})) \text{ is true.}

In cases 7 and 8, PS may change. However, all preconditions of unchanged leaves are parallel to the commands of the changed leaves and we may use the non-interference proofs to show that these preconditions remain true.

7) $\overline{x} := \overline{e}$ is removed. Let $Q_{\overline{e}} = \text{pre}(\overline{x} := \overline{e})$ and $P$ be the precondition of some unchanged leaf. By the non-interference proof, $(Q_{\overline{e}} \land P) \Rightarrow P_{\overline{e}}$; hence, $\text{PRE}(T) \Rightarrow (P \land Q)_{\overline{e}}$ where $P \land Q$ is the conjunction of the preconditions of the unchanged leaves. If we abbreviate $(P \land Q)$ as $PQ$, we see in figure 6.2.1 that $PQ_{\overline{e}}[PS] = PQ[PS']$.

Hence, the postcondition of the assignment and the preconditions of the unchanged leaves hold after execution.

$$PQ_{\overline{e}}[PS] = PQ_{\overline{e}}[PS][PS]$$

\hspace{1cm} evaluate $\overline{e}$ before substitution

$$= PQ_{\overline{e}}[PS][PS; \overline{x} := \overline{e}[PS]]$$

\hspace{1cm} $\overline{x}$ does not appear in $PQ_{\overline{e}}[PS]$.

\hspace{1cm} so we can change the state at $\overline{x}$

$$= PQ[PS; \overline{x} := \overline{e}[PS]]$$

\hspace{1cm} state has correct values for $\overline{x}$

$$= PQ[PS']$$

\hspace{1cm} definition of $PS'$

Figure 6.2.1 -- $PQ_{\overline{e}}[PS] = PQ[PS']$

8) if $\ldots \Box b_i; A!T(\overline{e}) + S_i \Box \ldots$ fi and
if $\ldots \Box b_j; B?T(\overline{e}) + S_j \Box \ldots$ fi are replaced by $S_i$ and $S_j$, only if $(b_i \land b_j)$ is true and $A!T(\overline{e})$ and $B?T(\overline{e})$ are a matching pair.

The postconditions of the alternatives imply the postconditions of the alternations, so after execution the postcondition of the alternation holds. To see that the preconditions of $S_i$ and $S_j$ hold and that the preconditions of the unchanged leaves remain true, we make use of the proofs of satisfaction and non-
interference. Let the preconditions of the alternations be \( Q_i \) and \( Q_j \); we may assert \( Q_i \land Q_j \land b_i \land b_j \) in PS. The proof of satisfaction for this matching pair shows that

\[
(Q_i \land Q_j \land b_i \land b_j) \Rightarrow (\text{pre}(S_i) \land \text{pre}(S_j))^{X^*}
\]

Now by the same argument as was used for assignment, we know that \( \text{PRE}(T^*|PS^*) \) must hold.

The relation between the condition DLF and the model of execution should be clear. A configuration corresponds to a set of leaves. If the condition of the configuration is false, that set of commands cannot be leaves of a tree, for this would imply that \( \text{false}[PS] = \text{true} \). On the other hand, if the commands do represent a set of leaves, the condition will be satisfied by PS (because the proof system is consistent with the model) and DLF implies that there are leaves that may be replaced.
Chapter 7
Global synchronization in CSP

The synchronization primitives provided in CSP are designed to synchronize pairs of processes. Sometimes, however, it is necessary to synchronize all processes; that is, to wait until some condition is met in all processes at the same time. One solution is to have all processes report to a central process; this introduces problems associated with centralized systems. Our solution is to have the process that last meets its required condition to recognize that the system is synchronized and to signal the others.

Should the notion of global synchronization prove useful, it will be desirable to provide a notational extension to the parallel command to describe where a process should continue after synchronization.

7.1 The problem of global synchronization

We are given a system of processes, each of which establishes a predicate through interaction with other processes. Whenever a process is in a state satisfying its predicate, it is said to be quiet, otherwise it is active. No pair of quiet processes communicate. However, a quiet process may be called upon to respond to communication from an active process and, in response to the request, may become active. Hence a process can be alternately quiet and active and it is not possible to simply let a quiet process terminate or continue with another part of the program. Instead we must synchronize the system when all processes are quiet and then let each process terminate or continue as is appropriate.

If quiet processes only become active in response to a communication, then the system deadlocks when all processes are quiet. We prefer to avoid
the negative connotations associated with the term deadlock. Rather, just as a set of guarded commands with no true guards can be viewed as an error (in the case of alternation) or as a proper condition for termination (in the case of repetition), we view the condition of all processes being quiet as a correct one and refer to it as global synchronization.

Here we show how to modify a system so that some process, in particular the root of a spanning tree generated by the algorithm, recognizes that all others are quiet. The root can then start a message down the tree, informing the other processes. Algorithms for propagating information through a spanning tree are discussed by Chang [4]; any of these algorithms may be used.

One goal in the design of this algorithm was that the protocol interfere as little as possible with the normal execution of the system. To this end, the protocol only passes messages between pairs of quiet processes. We use the adjective basic when referring to the processes as originally written; protocol refers to that which is added.

The protocol requires the following restrictions on the topology of the communications network and on the nature of the processes involved.

1) The communications network defines a connected, undirected graph with nodes for processes and an edge between B and C if B and C can communicate. It is assumed that messages can be sent in either direction. The neighbors of B are N(B).

2) It is possible to classify the state of a process as either active or quiet, where
   a) the distinction is in terms of local information only,
   b) eventually all processes are quiet.
c) no basic messages are passed between pairs of quiet processes, and

d) the state never changes from quiet to active except due to a basic communication.

When all processes are quiet (which will occur due to b), they will remain quiet, because no basic communication will occur (due to c and d). Hence, the system is synchronized and the processes should be informed.

3) It is possible to add alternatives (guarded commands) to each process that can execute when the process is quiet.

The third restriction may be relaxed by incorporating the algorithm into the low level communications protocol. In that case, some means must be provided by which a process may inform the protocol that it is quiet; the protocol can assume that basic communication makes a process active.

7.2 A protocol for global synchronization

Our solution consists of dynamically generating subtrees of the communications network, where all nodes in the subtrees are quiet and the root of the subtree became quiet after all its descendants did. When one tree spans the network, a signal is received by the root.

We assume the existence of a logical clock (see chapter 8). All references to time will be with respect to this clock. We employ messages of the form $Q(t)$ to mean "Some process became quiet at time $t".$

In the quiet state, a process is ready to send or receive $Q$-messages from all its neighbors. For each process, define $M$ as the set of times received in $Q$-messages as well as the times when the process became quiet. Further, let $\max_t M$ be the maximum of $M$. It is desired that the process establish that $Q(\max_t M)$ has been sent to or received from all neighbors.
In a finite period after the last process becomes quiet (at time $T$), all edges will have transmitted the message $Q(T)$.

When a $Q$-message is received by a process with a time $t'$ (adding $t'$ to this process's set $M$) that is greater than the previous maximum of $M$, it defines an incoming engagement edge for $t'$ [6]. Subsequent incoming $Q(t')$-messages are acknowledged. After all outgoing $Q(t')$-messages have been acknowledged, the incoming engagement edge is acknowledged (with an accept-message).

Consider the tree whose edges are the engagement edges for some time, say $t$. The process with no incoming engagement edges is the root; it is the process that became quiet at time $t$. If all of its outgoing messages are acknowledged, indicating that the tree spans the system and that all processes are quiet, the root can signal the system with messages along outgoing engagement edges.

2.1 Protocol algorithm for a process

We introduce the following new variables into each process, $A_x$: $\text{max}_t$, $\text{time}_i$, and $\text{reply}_i$ for $i \in N(A_x)$, parent, children, and answered. These definitions need only hold when $B$ is quiet. The maximum of set $M$ is $\text{max}_t$. Associated with the edge to neighbor $A_i$ are $\text{time}_i$, the maximum time transmitted in a $Q$-message along this edge (in either direction), and $\text{reply}_i$, which can range over the values $\text{no}$, $\text{wait}$, and $\text{ok}$. If $\text{reply}_i=\text{wait}$, then there was an engagement edge (for $\text{max}_t$) from this process to $A_i$. If $\text{reply}_i=\text{ok}$, then $Q(\text{max}_t)$ either was sent to $A_i$ and did not define an engagement edge or was received from $A_i$. In any other case, $\text{reply}_i=\text{no}$. The edge to $A_{\text{parent}}$ is the incoming engagement edge for $\text{max}_t$: if $A_x$ became quiet at time $\text{max}_t$, there is no incoming engagement edge and parent=$x$. There are outgoing engagement edges to the processes indexed by
the set children, and answered implies that the incoming engagement edge has been acknowledged.

The basic program is assumed to be a repetitive command such that each guarded command will terminate in a finite time. Hence, a process will eventually be ready to select a guard of the repetitive command. We add the guarded commands in figure 7.3.1 to the repetitive command; each guarded command is replicated for $i \in \Pi(A_x)$.

\[\text{quiet} \land \text{time}_i < \text{max}_t \land A_i \parallel \text{Q(max}_t)\]
\[\quad + \text{time}_i := \text{max}_t;\]
\[\quad A_i \parallel \text{reply}_i;\]
\[\quad \text{if reply}_i = \text{wait} \rightarrow \text{children} := \text{children} \cup \{i\}\]
\[\quad \quad \text{if reply}_i = \text{wait} + \text{skip}\]
\[\quad \quad \text{fi}\]
\[\text{fi}\]

\[\text{quiet};\]
\[A_i \parallel \text{Q(t)}\]
\[\quad + \text{if } t < \text{max}_t \rightarrow A_i \parallel \text{mo}\]
\[\quad \quad \text{t} := \text{max}_t + A_i \parallel \text{ok};\]
\[\quad \quad \text{time}_i, \text{reply}_i := t, \text{ok}\]
\[\quad \quad \text{t} > \text{max}_t + A_i \parallel \text{wait};\]
\[\quad \quad \text{parent, children} := i, \emptyset;\]
\[\quad \quad (\forall i: i \in \Pi(A_x); \text{reply}_i := \text{mo});\]
\[\quad \quad \text{time}_i, \text{reply}_i, \text{max}_t := t, \text{ok}, t\]
\[\quad \text{fi}\]

\[\text{quiet} \land (\forall j: j \in \Pi(A_x); \text{reply}_j := \text{ok})\]
\[\quad \land \text{parent} = i \land \text{answered}; A_i \parallel \text{accept} \rightarrow \text{answered} := \text{true}\]
\[\text{fi}\]

\[\text{quiet} \land \text{reply}_i = \text{wait}; A_i \parallel \text{accept} \rightarrow \text{reply}_i := \text{ok}\]

Figure 7.3.1 -- Protocol for global synchronization
The basic program must be changed so that when process $A_x$ becomes quiet at time $q_t$ it establishes:

$$\text{quiet} \land \max_t = q_t \land$$

$$\neg\text{answered} \land (\forall i: i \in N(A_x) : \text{reply}_i = \text{no}) \land$$

$$\text{parent} = x \land \text{children} = \emptyset.$$ 

When the process begins, it must be the case that

$$\left(\forall i: i \in N(A_x) : \text{time}_i = 0\right).$$

When a process becomes quiet, it will be the case that

$$\left(\forall i: i \in N(A_x) : \text{time}_i = \max_t\right).$$

From this we may prove that

$$\left(\forall i: i \in N(A_x) : \text{time}_i \leq \max_t\right)$$

is universal.

2.4 Some process is signalled after all processes are quiet

A process is ready to quit if has become quiet and has sent or received $Q(T)$ along all incident edges and had the messages acknowledged, where $T$ is the time the process last became quiet. If this condition is met in process $A$, then $A$ is the last process to become quiet. The last acknowledgment is the signal that the system is synchronized.

In the next section, we show that "ready to quit" actually implies that all processes are quiet. Here we will show that eventually some process does become ready to quit.

First recognize that the protocol does not interfere with the basic computation. No protocol alternative contains a loop; therefore, unless there is deadlock, each alternative eventually terminates. The guards of the basic alternatives are unchanged; therefore, when control returns to
the point where an alternative may be chosen, the basic computation may proceed.

Now see that, once all processes are quiet, eventually some process is ready to quit. Let $T$ be the time the last process became quiet. Each $Q$-message sent increases the value of some variable $t_j^i$ (time in process $A_j$), but $t_j^i$ is bounded above by $\max_t^i$ which, in turn, is bounded above by $T$; therefore only a finite number of $Q$ messages can be sent.

At least one process has $\max_t^i = T$ (the one that became quiet at $T$). Any edge incident on a process with $\max_t^i = T$ will eventually have $Q(T)$ sent (first alternative) or received (second); hence, eventually all $t_j^i$ will be equal $T = \max_t^i$.

Together $r_j^i$ and $r_j^i$ define an edge of the graph. For every edge $i, j$ with $t_i^j = \max_t^i$ and $t_j^i = \max_t^j$, neither $r_j^i$ nor $r_j^i$ will be $\text{no}$; if it is a not-yet-accepted engagement edge, it will have reply equal $\text{wait}$ at the source of the engagement. Because engagement edges form trees, some node must have all reply's equal to $\text{ok}$; this node can send an $\text{accept}$-message to its parent (third alternative), which can receive the $\text{accept}$-message because its reply equals $\text{wait}$ (fourth). In this way the number of reply's that equal $\text{wait}$ decrease; therefore, eventually all processes have all reply's equal $\text{ok}$. At this time, the process that became quiet at $T$ is ready to quit.

We can state the previous argument more formally by giving an invariant for the repetitive command that results from combining the basic program with this protocol. The invariant formalizes the idea of engagement edges and describes the properties of the (sub)spanning tree that we construct. If $R$ last became quiet at time $T$ and $\max_t^R = T$, then $R$
is the root of the (sub)spanning tree defined by the children sets. In this tree, there is a path (of engagement edges) from R to each leaf \( v_n \) of the form

\[
(v_0 = R, v_1, \ldots, v_n)
\]

where

1) \( \text{parent}_R^R = R \)

2) \( v_{i+1} \in \text{children}^{i} \Rightarrow \text{parent}^{i+1} = v_i \), for \( 0 \leq i < n \)

3) for all processes \( v \), \( \max_{T}^T \prec v \Leftrightarrow v \) is in this tree

4) for some \( k, 0 \leq k < n \),

\[
\text{reply}^v_{v_1} = \text{wait}, \ldots, \text{reply}^v_{v_k} = \text{wait}, \text{reply}^v_{v_{k+1}} = \text{ok}, \ldots, \text{reply}^v_{v_n} = \text{ok}
\]

and for all \( k, 0 \leq k < n \), \( \text{reply}^v_{v_k} = \text{ok} \). (The first \( k \) edges in the path are the not-yet-acknowledged engagement edges.)

5) if \( v_i \) is in the tree and \( v_j \in H(v_i) \), then \( v_i < T \Leftrightarrow \text{reply}^v_{v_j} = \text{ok} \)

We can now say that a process \( R \) is ready to quit if \( \text{parent}_R = R \wedge (\forall v : v \in H(R) : \text{reply}^R_v = \text{ok}) \). From the invariant and this definition of ready to quit, we will be able to show that when a process is ready to quit, all processes are quiet.

### 7.5 When a Process is Ready to Quit, All Processes are Quiet

Let \( R \) be a process that is ready to quit and \( t \) be the time at which it became quiet. A process is said to have been engaged to \( t \) (by \( B \)) if it has received (from \( B \)) a message of the form \( Q(t) \). Process \( R \) becomes engaged to \( t \) when it becomes quiet. We consider a \( Q \)-message to be received if the recipient replies \text{ok} or \text{wait}. If the reply is \text{no}, we consider the \( Q \)-message not to have been sent. This is not a restriction
on the implementation, but a definition for purposes of this argument. Having been engaged, a process accepts \( t \) by sending an accept-message to the process that engaged it. For each time \( t \), we can now divide the lifetime of a process into three phases:

1) \( \text{EN}_t \) -- has not yet been engaged to \( t \)

2) \( \text{EA}_t \) -- has been engaged to, but has not yet accepted, \( t \)

3) \( \Delta_t \) -- has accepted \( t \).

We will use these terms as predicates; i.e. we will say process \( B \) is \( \text{EA}_t \) to mean that \( B \) has been engaged to time \( t \), but has not yet accepted \( t \).

Note that processes are engaged to times (events). This allows us to differentiate between two times at which the same process became quiet. We assume that the process that became quiet at \( t \) (and is now ready to quit) has been engaged to \( t \). Because it did not receive the message \( Q(t) \), it has no process to respond to and it is never in state \( \Delta_t \).

An engagement is a fragile thing. Having received the message \( Q(t) \), process \( B \) is engaged only so long as it does not receive a \( Q \)-message with a larger time or a basic message; when this happens, we say that the engagement is broken. We denote the state broken by \( \text{EN}_t \).

**Lemma 7.1** If the message \( Q(t) \) has been sent along edge \( B-C \), then both processes \( B \) and \( C \) are \( \text{EA}_t \) or \( \Delta_t \).

Without loss of generality, assume that the message was sent from \( B \) to \( C \). Having received the message, \( C \) is engaged. (This is a consequence of our definition of receiving a \( Q \)-message.) Having sent the message, either \( B=R \) and by definition is engaged or \( B \) previously received the message and was engaged.
**Lemma 7.2** All neighbors of R are MA_t or A_t.

Because R is ready to quit, we know that the message Q(t) has been sent along all edges incident on R. By the previous lemma, all neighbors are MA_t or A_t.

**Lemma 7.3** If B and C are neighbors and B is NE_t, then C is either NE_t or MA_t.

Because B is NE_t, it has not received Q(t) and hence has not sent the message either. In particular, Q(t) has not been sent along edge B-C, which means that C has not yet accepted t.

**Theorem 7.4** If some process R is ready to quit, having become quiet at time T, then all processes have accepted T and not broken the engagement.

1) If some process X is NE_T, then some process other than R is MA_T.

By lemma 3, all neighbors of X are NE_T or MA_T. Some neighbor of R is on a minimum length path from X to R (and is closer to X than R is). From lemma 2, this neighbor is MA_T or A_T; and, by induction along this path, there is a process other than R that is MA_T.

2) If some process X is MA_T, then R is not ready to quit.

X is MA_T implies that there is a path from R to X along engagement edges. (These are engagement edges for T. While there is no guarantee that the processes along this path are still engaged, we nonetheless refer to these as engagement edges.) Because X is MA_T, its predecessor on this path has not received the accept message from X and is consequently also MA_T. By induction, some neighbor of R is MA_T and R is waiting for an accept message; i.e., R is not ready to quit.
3) If $R$ is ready to quit then all processes are $A_T$.
   
   By 1, if some process is $NE_T$ then some process is $NA_T$. And, by 2, if some process is $NA_T$ then $R$ is not ready to quit.

4) If all processes are $A_T$ and some process $X$ is $A_T$ and $Br_T$ then $R$ is not ready to quit.

   Let $X$ be the process that broke the engagement at the earliest time.
   
   To break the engagement after accepting $T$, it received either a basic message or $Q(T')$ where $T' > T$.

   If $X$ received a basic message, it received it from a process that was not quiet and hence was either $NE_T$ or $Br_T$. Similarly, if $X$ received $Q(T')$, the neighbor was engaged to $T'$ and $NE_T$ or $Br_T$. If the neighbor was $NE_T$ then lemma 3 states that $X$ was $NE_T$ or $NA_T$. Having broken the engagement, $X$ could not later become $A_T$. If the neighbor was $Br_T$, $X$ was not the first process to break with $T$. Both cases contradict assumptions.

5) All processes are $A_T$ and not $Br_T$.

   From 3 we conclude that if $R$ is ready to quit, all processes are $A_T$.

   From 4 we conclude that if all processes are $A_T$ and $R$ is ready to quit, no process is $Br_T$.

   **COROLLARY 7.5** If some process is ready to quit then all processes are quiet.

   This follows directly from the definitions of $A_T$ and $Br_T$.

7.6 The algorithm does not deadlock before some process is ready to quit.

   In section 7.4 we showed that if deadlock does not occur, eventually some process is ready to quit. Here we show that deadlock does not occur.
The only communication commands that are not in the guard of the repetition are the send and receive of reply's in the first two alternatives; there is exactly one send and one receive and we know that they match.

To be deadlocked, all time\(^i\) = max\(_t\^i\); in section 7.4 we showed that this implied that there was a tree of not-yet-accepted engagement edges with leaves all of whose reply's are ok. Hence the third guard is true. But we also showed that that the parent has reply = wait; therefore there is communication progress. When the tree of not-yet-accepted engagement edges is reduced to one node, the root is ready to quit; hence, there is no deadlock before some process is ready to quit.

2.2 Comments

Although we have not formally applied the rules of the previous chapters, we have used the concepts and the intuition that they support. The states corresponding to E\(_t\), A\(_t\), A\(_t\), and B\(_t\) can easily be modeled in the formal system.

The only time t for which a process is not B\(_t\) is t = max\(_t\) and then only when quiet; variable answered is then equivalent to A\(_max\_t\). Whenever a process B is broken from a time, the time can be added to one of two sets -- ENGAGED\(_B\) or ACCEPTED\(_B\) -- depending on whether max\(_t\) is accepted or not.

Therefore we have the relations (dropping the superscript)

\[
\begin{align*}
\text{E}_t &= t \land (\text{ENGAGED} \lor \text{ACCEPTED}) \land (\neg \text{quiet} \lor \text{max}_t = t) \\
\text{A}_t &= (\text{quiet} \land \text{max}_t = t \land \neg \text{answered}) \lor t \in \text{ENGAGED} \\
\text{A}_t &= (\text{quiet} \land \text{max}_t = t \land \text{answered}) \lor t \in \text{ACCEPTED} \\
\text{B}_t &= t \in (\text{ENGAGED} \lor \text{ACCEPTED})
\end{align*}
\]

The reader is invited to rewrite the lemmas in terms of these definitions and consider the formal proof.
Chapter 8
Discussion

8.1 Overview

We have shown how to extend a proof method for sequential programs to encompass communication. A satisfaction proof formalizes the intuitive argument that says that communication is distributed assignment. It encompasses the effects of synchronization and message passing.

We extended the notion of guarded command to include both input and output guards. This symmetry is possible because we do not distinguish between send and receive except in the definition of matching pairs.

Unlike repetition as described in [10], our repetitive command does not allow termination to occur because other processes are terminated; instead, termination only occurs when all boolean guards are false. This makes the termination conditions explicit, simplifies both the proof rule and the implementation, and improves the modifiability of programs.

The idea of synchronously altered variables turned out to be surprisingly useful when designing and explaining algorithms. It supports the idea that there is a global value that is to be approximated by local variables. In fact, we often design an algorithm in terms of global auxiliary variables, making certain that they are synchronously altered; then we add the local variables to make the program work within the constraints of CSP.

We have also presented a means of globally synchronizing a system. This is useful when each process is supposed to establish some local condition, but may be forced to invalidate the condition in response to communication from other processes.
8.2 The development of the communication axiom

Initially, we had separate axioms for send and receive. The receive axiom had the form it does now; there was no relation between the pre- and postconditions. This was in recognition of the command’s ability to change the value of its parameter in a way not determined by the command itself. The send axiom was the same as that for skip; after all, sending a message should not affect the state of the sender.

After a time, we found that the send axiom was not strong enough and, realizing that in terms of weak correctness one should be able to assert anything after a send executing in isolation, we changed it.\(^1\) We observed that in Hoare’s formal model [11] there is no real difference between send and receive. In his model there is only synchronization; receive is an abbreviation for a (possibly infinite) set of alternatives, the choice of which is determined by synchronization and determines the value received. This implies that synchronization provides an information flow into a sending process. This observation strengthened our opinion that the send axiom we now have is the correct one.

8.1 Logical Clocks

Lamport has discussed the notion of a logical clock in a system with no-wait sends [13]. A logical clock assigns unique integers (times) to events in such a fashion as to be consistent with causality.

Each process has a local approximation to the clock. Local events increment the clock. When a message is sent, a timestamp is included; on receipt, the receiving process updates its clock to a value greater than both the timestamp and the local clock value. This provides only a partial ordering, since there is no guarantee that local clocks assign unique \(^1\) We also liked the simplicity of one axiom for communication.
times. Any linear ordering, however, that is consistent with this partial ordering is also consistent with causality. An arbitrary but deterministic tie-breaking mechanism is then used to assign unique times. (One such mechanism would be to number the processes and use this numbering to break ties.)

Unfortunately, this does not correspond to causality in the synchronous message passing environment. Here, send and receive occur at the same time. To reflect this aspect of causality, it is necessary that corresponding send and receive be assigned the same time.

This ordering may be achieved by acknowledging send with the time of receive and using this as the time of send as well. Because of the overhead involved, it might be important to provide a logical clock as part of the system. All protocols with which we are familiar require that send be acknowledged; a timestamp could be included, allowing the overhead to be shared. It is our opinion that send must be acknowledged in any protocol in order to implement this causality.

8.4 Related work

The system presented views communication as a means to an end; that is, processes are sequential programs with communication providing external information. In other proof systems (Chandy and Misra [3], Hauser [9], Hoare [11]), the sequence of messages produced is the purpose of the process; sequential programs provide a means of controlling the communications. Such proof systems are based on the history of communication and introduce variables that record each send and receive. Rather than include this in our proof rules we allow auxiliary variables, which can be used to record as much or as little history as is needed. (Compare the auxiliary variables IN and OUT from the bounded buffer example
with set\textsubscript{i} from Min.\)

Chandy and Misra use global relations on sequences of messages to specify networks of processes. Their system requires that all output commands preserve the global relations assuming that all inputs do the same.

In [11], Hoare represents processes by sets of strings over an alphabet of messages; each process has a specific alphabet. The notions of input and output have been abstracted; instead, if one process is to communicate a message, all other processes that include that message in their alphabet must communicate that message at the same time. Input then consists of a set of alternatives; the message "received" is determined by which alternative is chosen to synchronize with the "sender".

Hauser extends the work of Hoare. A process is specified by the set of strings it can communicate. However, instead of algebraically combining these sets as Hoare did (which we feel is inappropriate for proving the correctness of programs), Hauser extends the assertion language to allow references to the specification of the language and then couches the proof in an annotated program. A major advantage of Hauser's work is that it can handle problems of liveness, like starvation. To do this, he introduced proof rules that allow one to prove facts about what will happen as opposed to what has happened.

While preparing this thesis, we learned of similar research being done by Apt, Francez, and de Roever [1]. Their system has the same axiom for send and receive as we do, but replaces a satisfaction proof by a cooperation proof. The cooperation proof is different in that it is derived from the forward assignment rule and that global invariants are used to relate auxiliary variables, rather than allowing shared references
to auxiliary variables. This global invariant is then used to eliminate matching communication pairs that cannot synchronize. The idea of extending matching communication pairs to allow assignments to auxiliary variables during communication, as a means of reducing the work involved in proving non-interference, was not present in their early work. Instead they allow sections of code to be considered atomic actions; this is their means of changing local variables in two processes synchronously. The global invariant must be true outside of these atomic actions; it must be shown that commands in other processes do not interfere with assertions inside of the atomic actions.

Our algorithm for global synchronization is similar to the "largest finder" algorithm of Chang [4] in that it finds the process whose time of becoming quiet is largest. The important difference is that Chang's algorithm requires that the value associated with a process remain constant and, of course, the time a process last became quiet can change.

8.5 Research

Further research needs to be done regarding deadlock and starvation. The suggested approach to deadlock requires much in the way of case analysis, a common source of error. The problem of starvation is ignored; instead all processes are required to terminate. The problem of dealing with non-terminating processes is an area for future research.

In dealing with starvation, one must begin to consider time as a factor. Whereas deadlock is a static condition, starvation is a dynamic one. To formally study the problem will entail the use of a logic that can discuss time or sequences of execution. Temporal logics, dynamic logics, and trace logics are all candidates. However, as has been stated earlier, we feel that these logics are not suitable for the design of algorithms.
The problem, as we see it, is to use temporal logic, say, to prove that an axiomatic system is sufficient to ensure the absence of starvation. Although we did it informally, this was the approach taken in chapter 6 when we proved that all processes terminate.

Another important area for research is the design of new constructs and better understanding of old ones. For example, how does the Rule of Satisfaction extend to buffered message passing? In this case, the postcondition of a receive can only assert that it has read a prefix of what has been sent; the postcondition of a send must be implied by its precondition because there is no information flow back to a send when messages are buffered. Consequently, output guards make little sense. This is only true for infinite buffers. With bounded buffers, the problem is even more complex. Still, buffered systems are important and more work needs to be done.

Of new constructs, consider adding a guard called otherwise to alternation. (This is similar to the default case in Ada accept statements.) A simple definition would be that the alternative guarded by otherwise is selected only when the other guards are not ready. Unfortunately, this is a time dependent definition. On single processor systems, where parallelism is simulated by interleaving, some implementations could assume that otherwise could always be taken, because no other process is "ready" at the same time; this is clearly not what is desired. Another possibility is to consider a guard to have aborted if the process being waited for has terminated and only take otherwise when all guards abort. This sounds reasonable, but it does not allow the process containing otherwise to continue immediately. What is actually wanted? Perhaps some form of time-out mechanism or a distributed
equivalent of first-come-first-serve scheduling. What type of proof rules will capture these ideas?

The dynamic properties of programs are becoming more important as we see more work being done in real time systems. We must fully understand our tools, if we are to make good use of them.
Chapter 9

Bibliography


