W&M ScholarWorks

Dissertations, Theses, and Masters Projects

Theses, Dissertations, & Master Projects

1999

Causal distributed assert statements

Sharon J. Simmons
College of William & Mary - Arts & Sciences

Follow this and additional works at: https://scholarworks.wm.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Simmons, Sharon J., "Causal distributed assert statements" (1999). *Dissertations, Theses, and Masters Projects.* Paper 1539623962.
https://dx.doi.org/doi:10.21220/s2-fcrz-jw60

This Dissertation is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Dissertations, Theses, and Masters Projects by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

CAUSAL DISTRIBUTED ASSERT STATEMENTS

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William and Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Sharon J. Simmons

1999

UMI Number: 9974950

UMI®

UMI Microform 9974950

Copyright 2000 by Bell & Howell Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

Sharon J. Simmons

Sharon J. Simmons

Approved, May 1999

Phil Kearns
Thesis Advisor

William Lee By

William Bynum

Weizhen Mao

Weizhen Mao

Robert E Noon

Robert Noonan

Rex K. Kincaid

Rex Kincaid
Department of Mathematics

Table of Contents

List of Figures xi

Abstract xii

1 Introduction 2

1.1 Monitoring Sequential Programs 2

1.2 Monitoring Distributed Systems 3

1.2.1 System Model 4

1.3 The *Happens Before* Relation 5

1.3.1 Asynchronous Message Passing Library 6

1.3.2 Partial Order of Events 7

1.3.3 Multiple Partial Orders 7

1.4 Outline of the Dissertation 9

2 Distributed Programs 11

2.1 Set partition 11

2.2 Mutual Exclusion 13

2.3 Bubble Sort 14

2.4 Tree Sort 16

2.5 Positive Ack/Retrans Protocol 18

3 Monitoring Methods 22

3.1 Global State 22

3.2 Runtime Methods 23

3.3 Postmortem Methods 30

4 Causal Distributed Assert Statement 32

4.1 Model and Notation 35

4.2 Implementation 40

5 Optimization 46

5.1 Timing Results 46

5.2 Piggybacking messages 48

6 Static Analysis 57

6.1 Goals of Static Analysis 57

6.2 Static Analysis in the Distributed Domain 58

6.3 Communication Analysis for Asynchronous Message Passing 63

6.4 *LCP* and *LCP′* Events 105

6.5 *POG* and Taylor's Complete Concurrency History 121

6.6 Static Analysis in the Parallel Domain 121

7 Loops **124**

7.1 Control Flow Graphs 124

7.2 *H* Graph 135

7.3 POG 152

7.4 *LCP* and *LCP′* events 160

8 Static Analysis of Distributed Programs **169**

8.1 Set Partition 169

8.2 Mutual Exclusion 174

8.3 Bubble Sort 179

8.4 Tree Sort 187

8.5 Positive Acknowledgement/Retransmission 192

8.6 Prototype 197

9 Conclusions **210**

9.1 Communication Systems 211

9.2 Complexity issues of static analysis 216

9.3 Future Work 217

  9.3.1 Data Analysis 217

  9.3.2 Modifications to the Distributed Program 218

  9.3.3 Global Assert Statement 219

9.4 Concluding Remarks 221

A Grammar **222**

B Asynchronous Library Functions **237**

Bibliography **249**

List of Figures

1.1 Asynchronous program 8

1.2 Space-time diagram 8

1.3 Multiple Partial Orders 9

1.4 Space-time diagrams 9

2.1 Algorithm for Set Partitioning Program 12

2.2 MUTEX 13

2.3 Bubble Sort 15

2.4 Local Sort 16

2.5 Distributed Bubble Sort 16

2.6 Tree Sort 17

2.7 Distributed Processes 18

2.8 Distributed Tree Sort 19

3.1 Local Snapshot phase of Global Snapshot Algorithm 26

3.2 Set Partition 28

3.3 Two asynchronously communicating processes 30

3.4 Lattice of global states 30

4.1 Consistent Cuts of the Assert Statement 34

4.2 3 process distributed system 36

4.3 Partial orders 37

4.4 Causal cuts for event *e* 38

4.5 Causal Global State for an Assert 39

4.6 *Latest* State 42

4.7 Vector Time 42

4.8 Propagation Protocol 44

4.9 Update Causal State Buffer 44

5.1 Datagram experiment 47

5.2 LCP and LCP' events of the Assert Event 50

6.1 Flow graphs of a 2 process system 60

6.2 Flow graphs of a 4 process system 60

6.3 Successor sets 61

6.4 Complete Concurrency History of figure 6.1 63

6.5 if/else portion of control flow graph 68

6.6 if portion of control flow graph 69

6.7 Flow graphs for a simple 3 process system 69

6.8 P_i's source code 70

6.9 FG_i 71

6.10 `if` and `if/else` flow graphs 73

6.11 Tree H for simple 3 process system 81

6.12 Possible and impossible receives 91

6.13 Same partial orders 94

6.14 2 possible POGs 101

6.15 H tree 103

6.16 POG derived from H of figure 6.11 105

6.17 LCP and LCP' events 109

7.1 Control flow graph of the loop constructs 130

7.2 Control flow graph with a while loop 132

7.3 Control flow graph with a do - while loop 133

7.4 Example 1 136

7.5 Example 1 with back edges 137

7.6 Example 2 138

7.7 Example 2 with back edges 139

7.8 Case 1 142

7.9 Case 2 144

7.10 Detecting a loop 145

7.11 Detecting a loop 146

7.12 Assert in the loop body 162

7.13 Assert and receive in the loop body 163

7.14 Assert not in the loop body 164

7.15 *POG* with a back edge 165

8.1 Flow Graphs for Set Partition 171

8.2 *H* for Set Partition 172

8.3 *POG* for Set Partition 173

8.4 Flow Graphs for Mutual Exclusion 176

8.5 Graph *H* for Mutual Exclusion 177

8.6 *POG* for Mutual Exclusion 178

8.7 Time space diagram for Bubble Sort 181

8.8 Flow Graphs for Bubble Sort 182

8.9 Graph *H* for Bubble Sort 183

8.10 *POG* for Bubble Sort 185

8.11 *LCP* and *LCP'* events for Bubble Sort 186

8.12 Flow Graphs for Tree Sort 188

8.13 Graph *H* for Tree Sort 190

8.14 *POG* for Tree Sort 191

8.15 Time Space Diagram for Tree Sort 192

8.16 Flow Graphs for Positive Ack/Retrans 194

8.17 Graph *H* for Positive Ack/Retrans 195

8.18 *POG* for Positive Ack/Retrans 196

8.19 *LCP* and *LCP'* events for Positive Ack/Retrans 197

ABSTRACT

Monitoring a program's execution is fundamental to the debugging, testing and maintenance phases of program development. This research addresses the issue of monitoring the execution of a distributed program. In particular, we are concerned with efficient techniques for evaluating global state predicates for distributed programs. The global state of a distributed program is not well-defined, making the monitoring task complex compared to that of a sequential programs. Processes of a distributed program execute concurrently, and the events of the program cannot be totally ordered. Each process has its own local memory, and the local memories are physically separate.

Despite the difficulties of defining a distributed computation's states, monitoring a distributed program requires reasoning about constituent processes' execution as a single collective entity. We have extrapolated the semantics of the sequential program's assert statement into the distributed context. A distributed assert statement is a global predicate that is anchored at a control point of one processes, and that is evaluated when that process executes the assert.

We have developed a runtime method for monitoring both stable and unstable properties that does not disrupt the computation of the distributed system. A distributed assert statement is evaluated with that statement's *causal global state* which incorporates the state of the system as a whole as it may have causal impact upon the assert statement. A runtime protocol has been implemented that constructs the causal global state and evaluates the assert statement. No additional synchronization or message passing is imposed on the distributed application although some message sizes are increased to propaga te state information. The causal global state is immediately available providing real-time feedback.

CAUSAL DISTRIBUTED ASSERT STATEMENTS

Chapter 1

Introduction

1.1 Monitoring Sequential Programs

Observing a program's execution is fundamental to the debugging, testing and maintenance phases of program development. Debugging is premised on the ability to examine the value of a variable at chosen points during the execution of a program. Testing involves detecting erroneous threads of execution and invalid variable values. Maintenance relies on the ability to follow a program's execution and detect deviations from anticipated behavior.

The ability to observe a sequential program's execution is straightforward since a single thread of execution defines a total temporal order on the program's atomic operations. The execution of each atomic operation results in a new program state, where a program state is a function from variables to values [12]. An ordered sequence of states is defined while the program is executing, and at any point of execution the state of the program is immediately available since all variable values are stored in the same local memory.

Debugging, testing and maintenance examine a program's execution by comparing states with expected behavior. One common method of conveying the expected behavior of a program utilizes state predicates. A predicate used in this manner is a boolean function on a program state and is evaluated by replacing variables of the predicate with their state values [12]. Predicate evaluation is straightforward in a sequential program since a state is well-defined and immediately available.

Choosing appropriate predicates is dependent on the application and the activities monitored. Predicates can be chosen to detect program malfunction and, if skillfully designed, relay a strong clue about the location of the bug leading to the failure. Particular points of a program's execution may be crucial, and predicates should be designed for evaluation at these points. Evaluating a predicate after the execution of an identified atomic operation is consistent with Hoare-style axiomatic program verification techniques [14]. Complex verification statements such as loop invariants, upon which a proof of partial correctness is usually hinged, make obvious candidates for conversion into predicates. Debugging breakpoints and diagnostic print statements indicate positions for developing appropriate predicates. Independent of the application, predicates are a powerful monitoring tool throughout the program's life cycle.

1.2 Monitoring Distributed Systems

This research addresses the issue of monitoring the execution of a distributed program. In particular, we are concerned with efficient techniques for evaluating global state predicates for distributed programs. The global state of a distributed program is not well-defined, making the task of monitoring complex compared to sequential programs. Processes of a distributed program execute concurrently, and the events of the program cannot be totally ordered. Each process has its own local memory, and the local memories are physically separate from one another. A process is only immediately aware of its own local state. Access to the state of a remote process requires communication and incurs a delay which is usually substantial and often unpredictable.

1.2.1 System Model

A sequential program's execution and the execution of a single process of a distributed program are similar. The i^{th} atomic operation or event of a sequential program is represented by e_i, and the resulting state is represented by S_i. The execution of a sequential program is modeled as

$$\sigma = S_0 \xrightarrow{e_1} S_1 \xrightarrow{e_2} S_2 \ldots$$

The notation $S_{i-1} \xrightarrow{e_i} S_i$ denotes the execution of event e_i which causes a transition from state S_{i-1} to S_i.

A distributed system consists of a fixed number of distinct processes $\Pi = \{P_0, \ldots, P_{N-1}\}$. These processes share no memory and interact only via message passing. Each process consists of a totally ordered sequence of atomic events. The i^{th} event of P_j is represented by e_j^i, and the resulting local state is represented by S_j^i. The execution of P_j is modeled as

$$\sigma/j = S_j^0 \xrightarrow{e_j^1} S_j^1 \xrightarrow{e_j^2} S_j^2 \ldots$$

In both a process and a sequential program, it is possible to say which event or state happened before another event or state since the events of both are totally ordered. The execution of a distributed program is viewed as a set of events $E = E_0 \cup \cdots \cup E_{N-1}$ where E_i represents the events of P_i, and an irreflexive *partial* order is defined on these events [19]:

$$\rightarrow \subseteq E \times E.$$

The $\rightarrow$ relation is commonly referred to as *happened before*. For $e, f \in E$, $e \rightarrow f$ if and only if e has potential causal impact upon f.

1.3 The *Happens Before* Relation

Interprocess communication defines the happens before relationship among events on different processes. Asynchronous communication occurs when a process places a message "on the network," and continues execution. The process receiving the message blocks until it receives the message, then continues execution.

In an asynchronous communication regime, $\rightarrow$ is the smallest relation satisfying the following three conditions: (1) if e and f are events in the same process, and e happens before f, then $e \rightarrow f$; (2) if e is the sending of a message and f is the receipt of the same message, then $e \rightarrow f$; and (3) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

If $e \rightarrow f$, we say that e causally precedes f and that f causally succeeds e. If $e \not\rightarrow f$ and $f \not\rightarrow e$, then we say that e and f are causally unrelated or concurrent, denoted $e \| f$, and neither can causally affect the other.

1.3.1 Asynchronous Message Passing Library

We have developed a library of asynchronous communication functions for writing distributed programs that communicate asynchronously. Each process's program is written in the programming language C[16] with the addition of the asynchronous communication functions for message passing between processes. Appendix B covers in detail the asynchronous functions, but the two of primary interest are *async_send* and *async_recv*. The function async_send has the following format:

async_send(*i*, *msg*, *len*).

The message pointed to by *msg* of length *len* is sent to process *i*. If *i* is -1, the message is broadcast to all the processes of the distributed program. The function async_recv has the following format:

async_recv(*i*, *msg*, *len*, *waitsecs*).

A message from process *i* is copied into the address *msg*. The length of the received message is *len*. If a message does not arrive within *waitsecs*, async_recv returns with a value of -1. If *i* is -1, the message is accepted from any process of the distributed program. If *waitsecs* is 0, the process waits until the message is received. When presenting example programs, only the fields of *i* and *msg* for both async_send and async_recv will be indicated. The field *waitsecs* of async_recv is assumed to be 0 unless otherwise indicated.

The asynchronous library routines implement reliable FIFO (First In First Out) communication by default. Unreliable or non-FIFO communication can be configured by functions described in the appendix. The example asynchronous distributed programs that appear in this document are based on reliable and FIFO communication unless specified otherwise.

1.3.2 Partial Order of Events

When a distributed program executes, a *partial order* of the program events is defined. The order is not total because some events on different processes are causally unrelated. Figure 1.1 is a distributed program of two asynchronously communicating processes. The dots denote statements that are not relevant to the communication. A time-space diagram of the program's execution is given in figure 1.2. Each vertical line corresponds to a process's execution where the direction of the line indicates time increasing, and each tick on that execution line corresponds to an event. A diagonal arrow between two processes denotes a communication. The following are some of the concurrent ($\|$) and causal ($\rightarrow$) relationships that exist between the program's events:

Concurrent	Causal
$e_0^3 \parallel e_1^1$	$e_0^1 \rightarrow e_1^2$
$e_0^4 \parallel e_1^1$	$e_0^3 \rightarrow e_1^2$
$e_0^5 \parallel e_1^6$	$e_1^6 \rightarrow e_0^6$
$e_0^6 \parallel e_1^8$	$e_0^1 \rightarrow e_1^8$

1.3.3 Multiple Partial Orders

The communication of a distributed program is classified as defining either a *single* partial order or *multiple* partial orders. The classification is based on the control constructs and

```
   P0                        P1
1  ·                      1  ·
2  x=0;                   2  async_recv(0, &z);
3  async_send(1, &x)      3  ·
4  ·                      4  ·
5  ·                      5  ·
6  async_recv(1,&x);      6  ·
7  ·                      7  async_send(0, &z)
8  ·                      8  ·
```

Figure 1.1: Asynchronous program

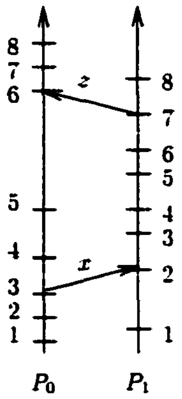


Figure 1.2: Space-time diagram

the communication functions they affect. The remaining statements of a process do not affect the partial order, and therefore are ignored.

If none of the processes have control constructs affecting the communication functions, the classification is a single partial order. If one or more of the processes have a control construct selecting among multiple communication functions, the classification is multiple partial orders. The partial order defined when the distributed program executes may differ according to which communication function is selected by the control construct.

Figure 1.3 is an example of a distributed program that is classified as defining multiple partial orders. The `if/else` control construct of P_0 selects one of the two groups of communications functions to execute. The two possible partial orders are shown in figure 1.4. The function **async_send(1,**w**)** is represented by i, function **async_send(2,**w**)** is presented by j, function **async_recv(0,**y**)** is represented by k, and function **async_recv(0,**z**)** is represented by l.

```
P0                      P1                      P2
⋮                       ⋮                       ⋮
w = i + 1               async_recv(0,y);        async_recv(0,z)
if (w > 0)              ⋮                       ⋮
  async_send(1, w);
  async_send(2, w);
else
  async_send(2, w);
  async_send(1, w);
⋮
```

Figure 1.3: Multiple Partial Orders

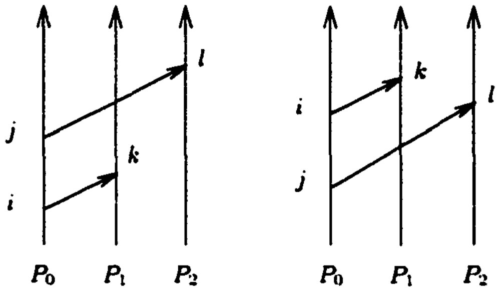


Figure 1.4: Space-time diagrams

1.4 Outline of the Dissertation

Chapter two presents several distributed programs that will be used in discussing distributed monitoring methods. The programs range from a single partial order program with repeating communication patterns to a multiple partial orders program with complex communication patterns.

In chapter three we review well-known monitoring methods that appear in the literature. Problems that these monitoring methods incur are discussed. Both runtime and postmortem methods are reviewed.

In chapter four our methodology for monitoring a distributed system is presented. The terminology and notation corresponding to our methodology is defined. This chapter also contains our initial algorithms.

Chapter five examines the affects of our initial algorithm to the execution of a distributed program and defines the messages that are sufficient for implementing our method of monitoring a distributed system. Chapters six and seven present algorithms for optimizing our initial results.

In chapter eight we apply our methodology for examining the execution of a distributed program to the programs of chapter two. Chapter nine concludes with possible avenues for continuing our research.

Chapter 2

Distributed Programs

Five distributed programs appear throughout this document to demonstrate and clarify concepts for monitoring distributed programs. These programs are described in detail in this chapter. The communication complexity of the programs varies greatly and is discussed with each program.

2.1 Set partition

SETPART, the set partition program, by Dijkstra [7] partitions disjoint integer sets S and T. SETPART exchanges an element of S with an element of T until the elements of S are less than the elements of T. The original sizes of S and T are maintained after each exchange. SETPART consists of two distributed processes, P_0 and P_1. P_0 maintains S, and P_1 maintains T. Processes P_0 and P_1 exchange an integer to determine if the sets are already partitioned correctly, then P_0 initiates an integer exchange with P_1 if there exists an element of S that is greater than the element previously received from P_1. For the exchange, P_0 sends the maximum element of S to P_1 and removes this value from its set.

P_1 receives the integer from P_0 and adds this integer to T, then P_1 sends the minimum element of T to P_0 and removes this minimum value from its set T. P_0 receives the integer from P_1 and adds this integer to S. P_0 continues to initiate an exchange until it determines that the sets are partitioned correctly. If the last value P_0 receives from P_1 is greater than or equal to the maximum of S, then no element of T is less than any element of S. And P_0 can conclude that partitioning is complete.

Set Partitioning's communication behavior exhibits *conversational continuity* [31], which is interactive communication between processes where a continuously repeating communication pattern is formed. The number of communications between the SETPART processes is dependent on the input data, but the communication pattern is static. Figure 2.1 is the distributed SETPART program for P_0 and P_1. The function **max** returns the maximum integer of the operand set, and the function **min** returns the minimum integer of the operand set.

```
P0::
mx = max(S)
async_send(1, mx)
S = S - {mx}
async_recv(1, x)
S = S ∪ {x}
mx = max(S)
while (mx > x)
    async_send(1, mx)
    S = S - {mx}
    async_recv(1, x)
    S = S ∪ {x}
    mx = max(S)
endwhile

P1::
while(true)
    async_recv(0, y)
    T = T ∪ {y}
    mn = min(T)
    async_send(0, mn)
    T = T - {mn}
endwhile
```

Figure 2.1: Algorithm for Set Partitioning Program

2.2 Mutual Exclusion

The circulating token mutual exclusion protocol can be embedded in distributed processes' application code if global mutual exclusion control is needed. The protocol defines a logical cycle through the processes, and the communication pattern is not influenced by the distributed system's application.

```
P($i$)::
do
    async_recv(($i + N - 1$) mod $N$, token, $waitsecs$)
    if message received
        if $want\_cs_i$
            $in\_cs_i$=true; $critsec_i$; $want\_cs_i$=false
        endif
        async_send(($i$ + 1)mod $N$,token)
    else /* async_recv timed out */
        do_other$_i$
    endif
enddo
```

Figure 2.2: MUTEX

MUTEX [21], shown in figure 2.2, is a token-based protocol for administering mutual exclusive critical section entry for a distributed system of N processes. The protocol allows only one process to enter its critical section at a time. Only one token exists in the system, and a process can neither create a token nor destroy the token. The processes are responsible for circulating the token around the system so that every process eventually receives the token. Process P_i receives the token from $P_{((i+N-1)\mathrm{mod}N)}$ and sends the token to $P_{((i+1)\mathrm{mod}N)}$. A process indicates that it wants to enter its critical section by setting *want_cs* to true. A process only enters its critical section when it receives the token and *want_cs* is true. Immediately before the process enters its critical section, *in_cs* is set to true. Process P_i passes the token to its neighbor $P_{(i+1)\bmod N}$ either when P_i completes its critical section or when P_i does not want to enter its critical section.

2.3 Bubble Sort

This distributed bubble sort algorithm is based on the odd-even transposition variation of the sequential bubble sort [43]. A total of q integers are sorted in ascending order with N processes where $N < q$. The processes are connected in a logical ring so that P_i's neighbors are P_{i-1} and P_{i+1}. Initially each process is assigned a list of q/N elements, and each list is sorted locally using a sequential sort.

The distributed sort consists of N phases, numbered 0 to $N-1$. If the phase number is even, each even numbered process sends its sorted list to its higher numbered odd neighbor, and each odd numbered process sends its sorted list to its lower numbered even neighbor. Each process merges the received list with its own list and sorts the resulting list. Each odd numbered process retains the last q/N elements of the list as its sorted list, and each even numbered process retains the first q/N elements of the list as its sorted list.

If the phase number is odd, similar steps are followed as for an even phase number. Each odd numbered process sends its sorted list to its higher numbered even neighbor, and each even numbered process sends its sorted list to its lower numbered odd neighbor. Each process merges the received list with its own list and sorts the resulting list. Each even numbered process retains the last q/N elements of the list as its sorted list, and each odd numbered process retains the first q/N elements of the list as its sorted list. Processes 0 and $N - 1$ do not participate in odd numbered phases.

After N phases are complete, all q numbers are sorted in ascending order where P_i has the elements $i \times q/N$ through $(i+1) \times q/N - 1$ of the sorted list. The bubble sort algorithm is shown in figures 2.3 and 2.4. Figure 2.5 shows the communication pattern for a bubble sort with a six process distributed system.

```
P_i::
    integer pid, phase;
    arrays list, recv_list
    pid = process's id
    read $q/N$ elements into list
    sort list
    for phase = 0 to $N - 1$
        if phase is even
            if pid is even
                async_send(pid + 1, list)
                async_recv(pid + 1, recv_list)
                list = merge_sort(list, recv_list, first)
            else
                async_send(pid - 1, list)
                async_recv(pid - 1, recv_list)
                list = merge_sort(list, recv_list, last)
            endif
        endif
        if phase is odd && pid != 0 && pid != $N - 1$
            if pid is even
                async_send(pid - 1, list)
                async_recv(pid - 1, recv_list)
                list = merge_sort(list, recv_list, last)
            else
                async_send(pid + 1, list)
                async_recv(pid + 1, recv_list)
                list = merge_sort(list, recv_list, first)
            endif
        endif
    endfor
```

Figure 2.3: Bubble Sort

```
merge_sort(list, recv_list, half)::
    array merge_list
    merge_list = merging of recv_list and list
    sort merge_list
    if half= first
        return first half of elements in merge_list
    else
        return last half of elements in merge_list
    endif
```

Figure 2.4: Local Sort

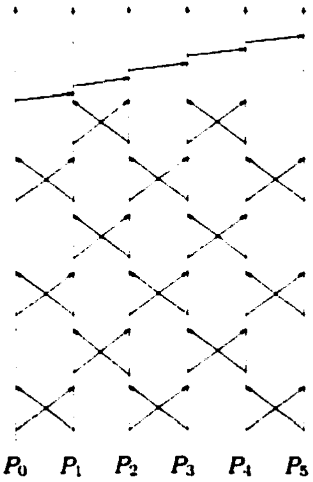


Figure 2.5: Distributed Bubble Sort

2.4 Tree Sort

The N processes of the tree sort distributed program are arranged in a binary tree. The number of processes required for this sort is $2^p - 1$, where $p > 1$. 2^{p-1} processes are leaf nodes. The process which is the root node of the tree initiates the sorting of q numbers, $q > N$. The root process splits the list in half and sends one half to each child process. If the receiving child process is not a leaf, it repeats the same steps as the root process. If the number of elements in the list is odd, the left child receives one more element than the right child. If the receiving child process is a leaf node, it sorts the list and sends the sorted list to its parent process. Once a parent process has received both of its children's sorted lists, the parent merges the two lists into one sorted list. If the parent node is not the root node, it sends this sorted list to its parent. The sort is complete when the root node receives two sorted lists from its children, and merges the two into one sorted list of q numbers.

The tree sort algorithm is shown in figure 2.6. Figure 2.7 is the binary tree formed by 15 processes ($p = 4$) $P_0 \ldots P_{14}$, and figure 2.8 shows the tree sort for the 15 processes.

```
$P_0$:: (root node)
    integer $child_1, child_2$
    arrays $list, list_1, list_2$
    read $q$ elements into $list$
    split list into two halves: $list_1, list_2$
    async_send($child_1, list_1$)
    async_send($child_2, list_2$)
    async_recv($child_1, list_1$)
    async_recv($child_2, list_2$)
    merge $list_1$ and $list_2$ into $list$
```

```
$P_i$:: (parent node)
    integer $child_1, child_2, parent$
    arrays $list, list_1, list_2$
    async_recv($parent, list$);
    split list into two halves: $list_1, list_2$
    async_send($child_1, list_1$)
    async_send($child_2, list_2$)
    async_recv($child_1, list_1$)
    async_recv($child_2, list_2$)
    merge $list_1$ and $list_2$ into $list$
    async_send($parent, list$)
```

```
$P_i$:: (leaf node)
    integer $parent$
    array $list$
    async_recv($parent, list$)
    sort $list$
    async_send($parent, list$)
```

Figure 2.6: Tree Sort

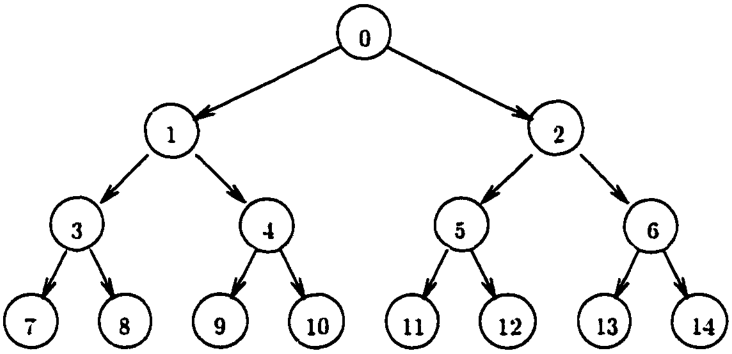


Figure 2.7: Distributed Processes

2.5 Positive Ack/Retrans Protocol

The positive acknowledgement/retransmission protocol presented by Tannenbaum [41] enforces reliable communication between two communication nodes, CN_s and CN_r, on an unreliable physical transmission line. The communication node CN_s only sends data messages, and the communication node CN_r only receives data messages. Associated with CN_s is at least one host that supplies the data for the outgoing messages, and associated with CN_r is at least one host that consumes the data of the incoming messages. Once CN_s has transmitted a message, it does not send another message until the message is received by CN_r without errors. The node CN_r informs CN_s with an acknowledgement message when it has received a message without errors. If CN_s does not receive an acknowledgement within a predetermined amount of time, it retransmits the data message.

Since the communication line is unreliable, the data message and the acknowledgement message can be lost or corrupted. There exists a problem with retransmitting the data message when the acknowledgement message is lost. Suppose CN_r has received an uncor-

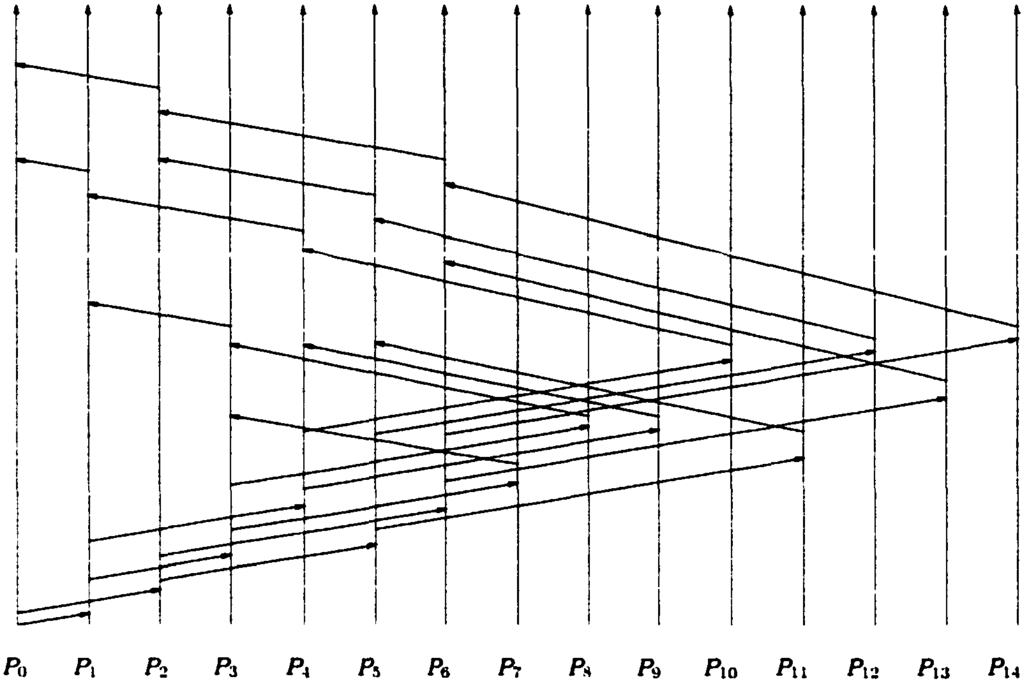


Figure 2.8: Distributed Tree Sort

rupted data message and sends an acknowledgement. If the acknowledgement is lost, CN_s retransmit the same data message. The node CN_r does not realize the data message is being retransmitted and interrupts the retransmitted message as a new message.

One bit appended to the data message provides the information for the receiver to distinguish between a retransmitted message and a new message. The node CN_s maintains a bit by alternating the bit when it receives an acknowledgement and appends the current value of the bit on data messages. The node CN_r maintains a bit by alternating the bit when it receives a valid data message. The receiver only accepts a data message as a new message if the bit on the message matches its bit value. Following is the described protocol:

```
Procedure $CN_s$::
    MsgBitSend : bit                              /* alternating bit */
    sbuffer: message                              /* buffer for outgoing data message */
    event: (MsgArrival, CksumErr, TimeOut)        /* different interrupt events */

    MsgBitSend = 0                                /* initialize alternating bit */
    FromHost(sbuffer)                             /* get the data message from host */
    repeat
        async_send( r,sbuffer, MsgBitSend)
        StartTimer;                               /* time to wait for acknowledgement */
        wait(event)                               /* possibilities MsgArrival, CksumErr, TimeOut */
        if event = MsgArrival
            async_recv(r, ack)                    /* receive the acknowledgement */
            FromHost(sbuffer)                     /* an acknowledgment has arrived intact */
            inc(MsgBitSend)                       /* increment by 1 then mod 2 */
        endif
    until doomsday
```

```
Procedure $CN_r$::
    MsgBitReceive : bit                           /* alternating bit */
    IncomingBit : bit                             /* incoming message's bit */
    rbuffer: message                              /* buffer for incoming data message */
    event: (MsgArrival, CksumErr)                 /* different interrupt events */

    MsgBitReceive = 0                             /* initialize alternating bit */
    repeat
        wait(event)                               /* possibilities MsgArrival, CksumErr */
        if event = MsgArrival                     /* a valid message has arrived */
            async_recv(s, rbuffer, IncomingBit)   /* accept the message */
            if IncomingBit = MsgBitReceive
                ToHost(rbuffer)                   /* pass the data to the host */
                inc(MsgBitReceive)                /* increment by 1 then mod 2 */
            endif
            async_send(s, acknowledgement)
        endif
    until doomsday
```

The **async_send** command transmits a message (data message and bit) over the communication channel, and the **async_recv** command accepts a message from the communication channel and assigns the data message to *rbuffer* and the bit to *IncomingBit*. Procedure

`StartTimer()` starts the timer and enables the Timeout Event. Procedure `Wait()` waits for an event to happen, and returns the event type when one occurs. The procedure `FromHost()` fetches a data message from the host, and the procedure `ToHost()` delivers a data message to the host.

Chapter 3

Monitoring Methods

3.1 Global State

A partial order is defined on a distributed system's events when the system executes. The notion of a system state is complicated by the lack of a total order among events. An additional complication is the difficultly of capturing a system state since local memories are physically separate from one another. Despite the difficulties of a distributed computation's states, monitoring a distributed program requires reasoning about constituent processes' execution as a single collective entity. Previous work [28, 4, 38, 37, 29, 33] has defined a global state for unified reasoning about the distributed processes. A global state is analogous to "gluing" together local states, one from each process, such that the local states can happen at the same "time". The "gluing" produces one possible state of the system.

Global states provide a means to monitor a distributed system's execution with global predicates. A global predicate for a distributed system is comprised of relationships among variables from different processes. Once a global state is constructed, a global predicate is evaluated on this state. Constructing a global state and evaluating a predicate on that state helps in any rational scheme for debugging and monitoring the distributed program.

3.2 Runtime Methods

Despite a global state's usefulness, problems exist with distributed system monitoring based on global states. A major problem is the difficulty of capturing a global state during the distributed system's execution.

Runtime methods of capturing a global state has been addressed by many researchers. Several papers that stand out in the literature are briefly described. Chandy and Lamport [4] were the first to define a global state as a global snapshot that could have occurred if all processes took a snapshot of their local states simultaneously. Their global snapshot algorithm assumes FIFO asynchronous communication, and each process has at least one incoming and outgoing unidirectional communication channel. Process P_i communicates directly with P_j if a channel exists from P_i to P_j, otherwise P_i communicates indirectly with P_j through intermediate processes and channels.

The snapshot algorithm consists of two phases. In the first phase, each process takes a snapshot of its state. In addition to the recorded local state information, the messages in-transit when the local snapshots are taken will be included in the global snapshot. The in-transit messages are flushed through the channels before the local snapshots are assembled into a global snapshot. A process initiates a global snapshot by (1) saving its local state, (2) sending a snapshot token message on each of its outgoing channels, and (3) beginning the recording of messages on each incoming channel. The token informs the receiving process that a snapshot is being taken, and it flushes the messages in-transit so they are included in exactly one process's local state. When a process receives a token. it performs the same three steps as the initiating process.

A process continues to record incoming messages for a channel until the process receives a snapshot token on the channel. Once a process has received a token on each channel, the process's local state is complete for the global snapshot.

In the second phase, each process disseminates its local state information to form a global snapshot. Each process must send its state information to each of its neighbors. and when a process receives other processes' states, it must relay this information to its neighbors. This type of dissemination ensures that the process requesting the snapshot eventually receives the global state.

Every process receives the global snapshot with Chandy and Lamport's algorithm. Kearns and Spezialetti [38] improve the efficiency of the global snapshot algorithm by reducing the message-passing load for disseminating the global state. Only the process or processes that initiate the global snapshot receive it. The process(es) that initiate the snapshot by passing snapshot tokens include their process identification with the tokens. The tokens continue with their original purpose of informing other processes to record their local states. Once a process has completed recording its local state, the local state is only sent to the process that prompted this process to take a snapshot. Once a non-initiating process has sent its local state to the initiating process, it has completed the global snapshot since it no longer has the responsibility of sending neighboring processes' state information through the network.

Lai and Yang [18] extend the original global snapshot algorithm by removing the FIFO restriction. One status bit is associated with each process and is piggybacked on all messages. Each process's bit is initially 0, and a process sets the bit to 1 when it initiates a snapshot. When a process receives a 1 status bit, its status bit is set to 1, and it takes a snapshot. Since the channels are non-FIFO, messages sent before the snapshot can still be in-transit after the snapshot is taken. These message must be incorporated into the global snapshot. Each process keeps a record of all messages it has received and sent for calculating the in-transit messages.

Mattern [28] develops an algorithm similar to Lai and Yang's for non-FIFO channels, but it does not require the processes to record messages. The algorithm ensures that the result of a process initializing a global snapshot is a consistent cut. A consistent cut is a set of events that are not causally related (concurrent), and each process has exactly one event in the cut. If an event e_i happens before P_i's cut event, and e_j happens before e_i, then e_j must happen before P_j's cut event for the cut to be consistent. This condition disallows messages sent after the cut to be received before the cut. The only messages in-transit after the cut are messages with a status bit of 0 being sent to processes with a status bit of 1. The global snapshot comprises the local states resulting from the cut events and the in-transit messages.

The global snapshot algorithms described share a common problem, they add causal dependencies to a distributed system's computation. To expose this problem, consider Chandy and Lamport's snapshot algorithm. The recording of P_i's local state and propagating the

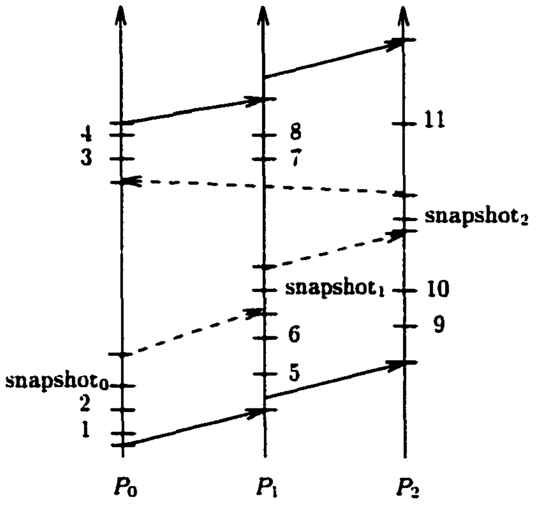


Figure 3.1: Local Snapshot phase of Global Snapshot Algorithm

snapshot token are events added to the distributed computation by the local snapshot phase of the algorithm. Figure 3.1 is a time-space diagram of a three processor system. The asynchronous messages of the computation (without the snapshot algorithm) are denoted with solid lines. The dashed lines represent snapshot token messages. The notation snapshot$_i$ indicates the local snapshot of P_i. Figure 3.1 shows both the local snapshots being taken and the propagation of the token, given that P_0 initiated a global snapshot after e_0^2. Assume no messages are in transit when the local snapshots are taken and the only communication channels are P_0's outgoing channel to P_1, P_1's outgoing channel to P_2, and P_2's outgoing channel to P_0. The global state obtained by this global snapshot is denoted by global_state, which is $\bigcup_{i=0\ldots2}$ snapshot$_i$.

The token messages add causality to the computation. For the events $e_0^1, e_0^2, e_0^3, e_0^4$ of P_0, events $e_1^5, e_1^6, e_1^7, e_1^8$ of P_1, and events $e_2^9, e_2^{10}, e_2^{11}$ of P_2, there exist no causal relationship between e_i^m and e_j^l for $i \neq j$, according to the distributed computation. For example, e_0^2 and

e_1^8 are concurrent in the underlying computation. The token messages add (false) causality to these events as defined by the happens before relationship. Event e_0^2 happens before e_1^8 according to the causal relationship added by the token transmission from P_0 to P_1. The concurrent execution of e_0^2 and e_1^8 is inconsistent with the causal order defined by the token messages. Events e_0^1 and e_0^2 are causally related to e_1^7, e_1^8 and e_2^{11}, e_1^5 and e_1^6 are causally related to e_0^3, e_0^4 and e_2^{11}, and e_2^9 and e_2^{10} are causally related to e_0^3 and e_0^4 due to the three token messages. For example, $e_0^2 \to e_1^7$ and $e_0^2 \to e_2^{11}$.

Adding causality to concurrent events invalidates legitimate global states of the underlying computation. For example, the cut consisting of events e_0^1, e_1^7 and e_2^{11} is consistent in the underlying computation, but is an inconsistent cut due to the causality added by the token messages. Since the cut of e_0^1, e_1^7 and e_2^{11} is not consistent, the global state consisting of the local states after the execution of e_0^1, e_1^7 and e_2^{11} is not a valid global state. The global state defined by e_0^1, e_1^7 and e_2^{11} is valid in the underlying computation.

Global snapshot algorithms require that obtaining a global state should not disrupt the computation of the distributed system, but these algorithms do interfere by imposing order on concurrent events. Distributed system monitors should be based on the uncorrupt computation of the system, and should not allow a method that invalidates legitimate global states.

An additional problem with global snapshots is their usefulness. Global snapshots are only adequate for detecting *stable* properties. Once a stable property occurs, it persists until the system is terminated. Examples include deadlock and termination. Predicates expressing stable properties are called global stable predicates. By taking global snapshots periodically, a stable property can be detected by a predicate evaluated on the sequence of snapshots.

Distributed monitoring and debugging properties are, in general, not stable. Predicates for detecting unstable properties are called unstable predicates. Repeated snapshots are inadequate for evaluating an unstable predicate, as the property expressed by the predicate may have occurred between snapshots, and gone undetected.

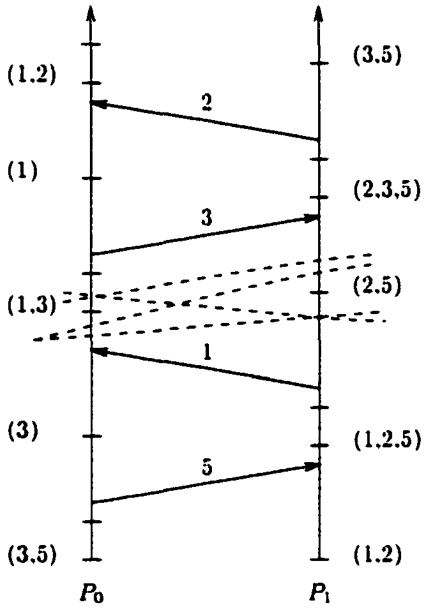


Figure 3.2: Set Partition

Consider the distributed program SETPART. A reasonable and informative global predicate to evaluate after each exchange of maximum and minimum datum values is $S \cap T = \emptyset$. If this predicate evaluate to true, SETPART is correctly updating the sets after an exchange. But many global states are possible after an exchange. A simple execution of SETPART is shown in figure 3.2. Each dashed line represents a possible global state after the first exchange. Figure 3.2 represents a valid execution of SETPART, but the evaluation of the global predicate may be either true or false. The predicate's evaluation does not provide insight into the correctness of the execution. If the global states are restricted by P_0 initiating the global snapshot after if receives x and adds x to S, two global states are possible: $S=\{1,3\}$, $T=\{1,2,5\}$ and $S=\{1,3\}$, $T=\{2,5\}$. One resulting in a false evaluation of the predicate, and the other resulting in a true evaluation. Although SETPART's communication has a simple repeating pattern, it exemplifies the deficiencies of monitoring unstable properties with existing runtime methods.

Cooper and Marzullo [5] propose an algorithm, *Currently*, for evaluating an unstable predicate while the system is executing. A process sends a monitor process, P_{mon}, its local state if the local state might affect the outcome of a known global predicate Φ. P_{mon} maintains the last received state of each process, and evaluates Φ each time it receives a process's state. If Φ evaluates to true, P_{mon} has detected an undesirable global state.

When a process enters a state that might falsify the evaluation of Φ, it freezes and sends a *block* message to P_{mon} before informing P_{mon} of its new state. The process remains blocked until P_{mon} has received all in-transit messages from the other processes. This flushing of messages allows P_{mon} to obtain in-transit states that might detect the predicate. Once the messages have been flushed, the blocked process sends P_{mon} its state and continues execution.

Although *Currently*'s objective is detecting unstable predicates, it is equivalent to taking snapshots periodically, and it can miss a state on which Φ evaluates to true. *Currently* incurs the same problem as the previously described algorithms, legitimate global states are invalidated by imposing causal relationships on concurrent events. When processes send state information to P_{mon} and receive acknowledgements from P_{mon}, order is imposed on concurrent events.

3.3 Postmortem Methods

Instead of capturing a global state while the system is executing, the postmortem algorithms *Definitely* and *Possibly* by Cooper and Marzullo [5] construct a lattice of all consistent global states based on trace data gathered during execution. *Possibly* Φ evaluates to true if there exists a global state which causes Φ to be true. *Definitely* Φ evaluates to true if for all total orders there exists at least one global state in each total order which causes Φ to be true. *Possibly* and *Definitely* provide a meaningful evaluation of unstable predicates since all global states are considered.

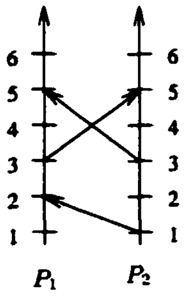

6
5
4
3
2
1
6
5
4
3
2
1
P_1
P_2

Figure 3.3: Two asynchronously communicating processes

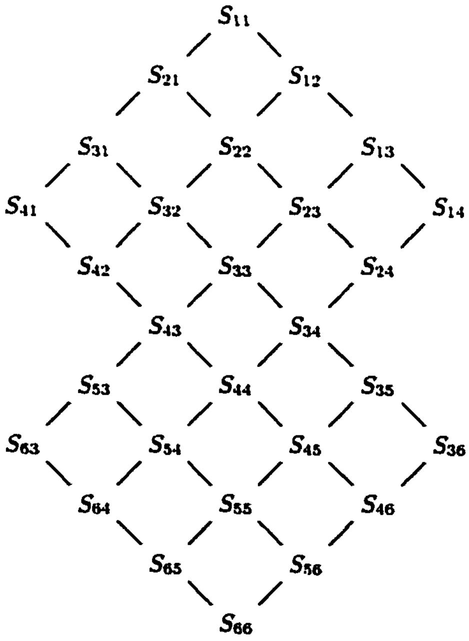

S_{11}
S_{21}
S_{12}
S_{31}
S_{22}
S_{13}
S_{41}
S_{32}
S_{23}
S_{14}
S_{42}
S_{33}
S_{24}
S_{43}
S_{34}
S_{53}
S_{44}
S_{35}
S_{63}
S_{54}
S_{45}
S_{36}
S_{64}
S_{55}
S_{46}
S_{65}
S_{56}
S_{66}

Figure 3.4: Lattice of global states

While the distributed system is running, each process informs P_{mon} of each local state it enters. P_{mon} maintains a FIFO list of these states for each process. Once the execution has completed, P_{mon} assembles the local states to construct a lattice of all consistent global states. Figure 3.4 shows the lattice constructed for the 2 processor distributed execution of figure 3.3. Point $S_{i,j}$ of the lattice is the global state where i events have occurred on P_1, and j events have occurred on P_2. The level of $S_{i,j}$ is $i+j$. A possible total ordering of states is a path starting at the level 1 global state, and each subsequent global state has a level increase of one. *Possibly* is true if at least one point in the lattice satisfies Φ. *Definitely* is true at least one point in every total ordering satisfies Φ.

Definitely and *Possibly* provide a meaningful predicate evaluation methodology by considering all global states. The outcome of evaluating Φ provides unambiguous information about the system's behavior. Although they provide meaningful results, the inability to monitor the system at runtime is a significant weakness of both algorithms. By waiting for the system to complete execution, on-line corrective actions such as recovery or abortion can not be made for invalid execution behavior. Real-time feedback is crucial for life- or mission- critical control applications.

We have developed a runtime method for monitoring a distributed system that is meaningful for both stable and unstable properties. Predicates are evaluated with all the processes' state information that may affect the evaluation. Any invalid system state, indicated by evaluation of the predicate, is detected. Evaluation is only with system states that can occur in the distributed computation, and legitimate global states are not invalidated. The following chapter describes our methodology, both in terms of design and implementation.

Chapter 4

Causal Distributed Assert Statement

Some sequential programming languages [39, 40] facilitate predicate evaluation with assert statements. An assert statement [30] (or library function, depending on it's implementation) generally has the form

$$\textbf{assert}(P)$$

where P is a predicate defined on the state of the program. The semantics of this assert statement are that P is evaluated, without side-effects, on the program state at the point at which the **assert** () is executed. If P is true then the program continues its execution. If P is false, however, the program is aborted, and a diagnostic message is produced.

We have extrapolated the semantics of the assert statement for sequential programs into the distributed context. A distributed assert statement is a global predicate that is anchored at a control point of one process, and that is evaluated when the process executes the assert. A distributed assert statement monitors a distributed system's execution, but only a subset of the system states of the execution are relevant for evaluating the assert. Two possibilities exist for which portion of the execution the distributed assert monitors. One possibility is the distributed assert statement monitors the global states that are defined by consistent cuts including the assert statement. This interpretation is in accord with the global predicate evaluation methods described in chapter 3. If the distributed assert monitors concurrent execution, then any consistent cut of the system that includes the assert event defines a valid global state for predicate evaluation. A simplistic three processor system is shown in figure 4.1. The broken lines represent all possible consistent cuts, and the × represents an assert statement.

The only previous work that resembles this interpretation of the distributed assert statement is Cooper and Marzullo's *Currently*[5]. *Currently* evaluates the global predicate Φ while the system is executing and is claimed to be appropriate for unstable predicates. But *Currently* is incomplete: global states can be missed that cause a true evaluation of Φ [33]. *Currently* is also intrusive of the system's execution since it introduces extra synchronization into the monitored computation, and it can cause a significant degradation in system performance. Every modification of a variable in Φ can be considered a possible invalidation of Φ, causing the network to be congested with block and acknowledgment messages and causing the process about to execute the modification to freeze until all in-transit messages to P_{mon} are received.

Another interpretation of the distributed assert statement is that it monitors the execution that has the most recent causal impact on the assert statement. We have developed

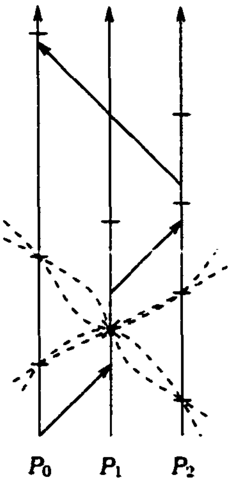


Figure 4.1: Consistent Cuts of the Assert Statement

a methodology for evaluating a distributed assert statement in accordance with this interpretation. Our methodology does not have the problems associated with global state reasoning. The state of the system necessary for evaluating the predicate is well-defined, and the evaluation result relays unambiguous information about the state of the system. Our distributed assert statement is characterized by two properties:

A1 The asserted predicate is evaluated during execution of the program. We do not generate and analyze traces *post mortem*.

A2 No additional synchronization or message passing is added to the original distributed application in support of the distributed assert statement. We do increase the size of some application messages.

4.1 Model and Notation

Recall that a distributed program consists of a fixed number of processes $\Pi = \{P_0, \ldots, P_{N-1}\}$, and the *happened before* relationship, $\rightarrow$, is a partial order on the program's events. For event e in P_i, $LCP(e, j)$ where $j \neq i$, denotes event e's *latest causally preceding event* in P_j. We define $LCP(e, j) = f$ if and only if f is an event in P_j, such that f happens before e, and there does not exist event f' in P_j such that f happens before f' and f' happens before e.

Definition 4.1 *For some event $e \in P_i$, the latest causally preceding event in P_j where $j \neq i$, denoted $LCP(e, j)$, is event f if and only if*

1. $f \in P_j$
2. $f \rightarrow e$
3. $\nexists f' \in P_j : f \rightarrow f' \rightarrow e$

One of possibly many partial orders is defined when a distributed system executes. This is due to branches in control of execution and to the fact that communication delays and process speeds are unpredictable. Hence sends and receives will "match up" unpredictably in general. Consider the source code of a three process distributed system shown in figure 4.2. One of possibly two partial orders is defined when this program executes. The two possible partial orders, PO_1 and PO_2, are shown in figure 4.3. Set $\mathcal{P}$ is the set of possible partial orders of a distributed system's execution. For the distributed system shown in

```
P0::

begin
   x = 1
   async_send(1,x)
   x =2
   async_send(1,x)
end
```

```
P1::

begin
   if
      async_recv(0,x)
      async_recv(2,y)
   else
      async_recv(2,y)
      async_recv(0,x)
   endif
   async_recv(0,x)
   assert(x = y)
end
```

```
P2::

begin
   y = 3
   async_send(1,y)
end
```

Figure 4.2: 3 process distributed system

figure 4.2. $\mathcal{P} = \{PO_0, PO_1\}$. For a given execution of the distributed system, one partial order, $\alpha \in \mathcal{P}$, is produced.

For a partial order, $\alpha \in \mathcal{P}$, at most one LCP event exists in each process for any event e. Each partial order may identify a different $LCP(e, j)$. The maximum unique LCP events of P_j for event e is bounded by the number of partial orders, i.e., the size of set $\mathcal{P}$.

Lemma 4.1 *For a partial order $\alpha \in \mathcal{P}$ of a distributed system and an event e of P_i, at most one $LCP(e, j)$ exists for $j \neq i$.*

Proof by contradiction. Assume two $LCP(e, j)$ events, e' and e'', exist for the one partial order α. According to the definition of LCP events (definition 4.1),

1. $e' \rightarrow e$, and there does not exist another event f such that $e' \rightarrow f \rightarrow e$
2. $e'' \rightarrow e$, and there does not exist another event f such that $e'' \rightarrow f \rightarrow e$

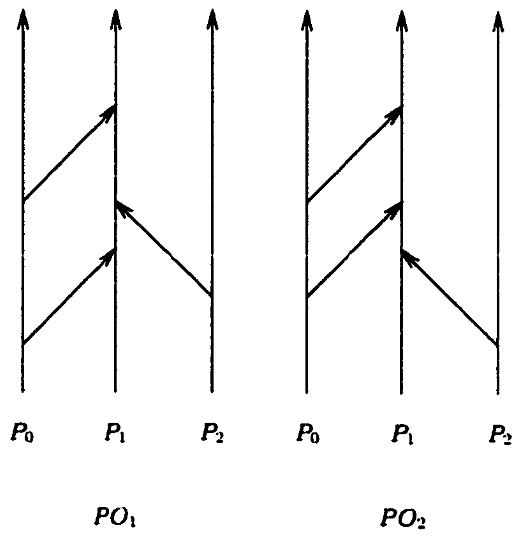


Figure 4.3: Partial orders

From 1. and 2., $e' \not\rightarrow e''$ and $e'' \not\rightarrow e'$. therefore $e' \| e''$. The concurrency of e' and e'' is a contradiction since both are events of P_j and the events of one process are totally ordered.

■

Consider event e in process P_i. A *causal cut* through e is the set of events consisting of e and the LCP event of e of each process for a partial order α.

Definition 4.2 *A causal cut through event e, denoted $CC(e)$, is defined as*

$$CC(e) = \{e\} \cup \left(\bigcup_{\substack{0 \le j < N \\ j \ne i}} \{LCP(e, j)\} \right).$$

Intuitively, $CC(e)$ is the "latest" set of events of Π which can have a causal impact upon e. In figure 4.4, the causal cut through P_0, P_1, and P_2 for event e is shown as a dashed line. An event f is said to be *before* causal cut $CC(e)$ if there exists event $g \in CC(e)$ such that

$f \rightarrow g$; f is *after* $CC(e)$ if there exists event $g \in CC(e)$ such that $g \rightarrow f$. Accordingly, we use CC to define a notion of global state to be used in the evaluation of a distributed assert statement.

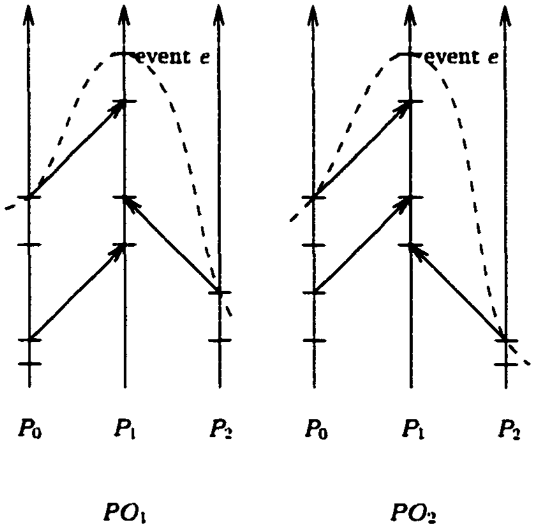


Figure 4.4: Causal cuts for event e

A causal cut does not necessarily include an LCP event from each process since each process may not have an event that occurs before an event e. For each $\alpha \in \mathcal{P}$, there is one causal cut for a given event. Also, the LCP events that comprise the causal cut for an event and one partial order may differ from the LCP events that comprise the causal cut for the same event and a different partial order.

Theorem 4.1 *For a partial order $\alpha \in \mathcal{P}$ of a distributed system and an event e of P_i, at most one $CC(e)$ exists.*

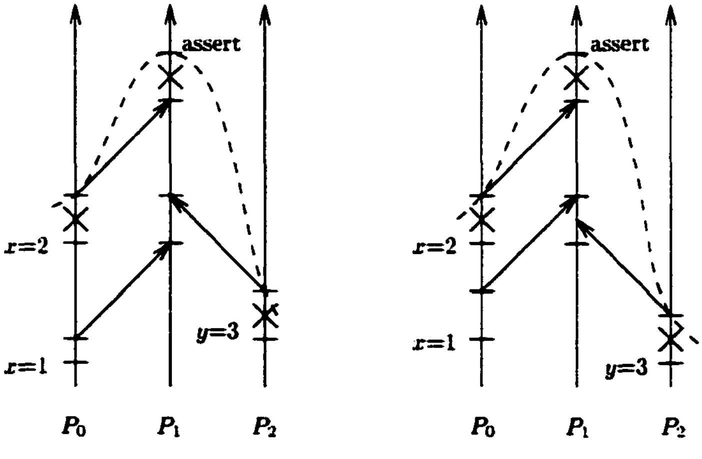
Proof. This follows directly from Lemma 4.1 and Definition 4.2. Since each process has at most one $LCP(e,j)$ for each $\alpha \in \mathcal{P}$ (lemma 4.1) and $CC(e)$ is comprised of the $LCP(e,j)$

from each process (definition 4.2), at most one $CC(e)$ can exist. ∎

For event e in process P_i, let $pre(e)$ denote the local state of P_i in which the execution of e is begun. Execution of e effectively terminates state $pre(e)$. If e is the execution of a causal distributed assert statement in P_i, then the causal global state, anchored on e, is simply

$$CGState(e) = \{pre(f) : f \in CC(e)\}.$$

CGState is the set of process states which immediately precede the causal cut through e, the execution of the assert statement. *CGState* thus incorporates the state of the system as a whole as it may have causal impact upon P_i at the point the assert statement is executed. Events which are after the causal cut through e cannot affect the execution of e. All events which happen before the causal cut will have their effect on e through transitivity.



PO_1 PO_2

Figure 4.5: Causal Global State for an Assert

Figure 4.5 shows a causal distributed assert statement being evaluated in process P_1. The horizontal lines across the process time lines represents events, and the dashed line represents $CC(\text{assert}(P))$. The individual process states compromising the causal global state anchored on the assert is denoted by x's on the process time lines. Partial orders PO_0 and PO_1 each have a corresponding causal cut and causal global state. Although in this example the causal cut and causal global state are identical, in other distributed systems they can be different. The causal global state is $P_0.x = 2$ and $P_2.y = 3$ for both partial orders.

4.2 Implementation

Our implementation of the causal distributed assert statement ensures that when an assert is executed, the relevant components of the causal global state are immediately available at the process executing the assert (Property A1). To that end, process P_i maintains its current view of the *CGState* in the causal global state buffer, $CGSBuffer_i$. Processes maintain their causal global state buffers independently. Buffer maintenance requires no message-passing or synchronization beyond that required by the underlying application (Property A2). Each causal state buffer consists of tuples of the following form

(process id, variable name, variable value, vector timestamp)

The meaning and use of vector timestamp is discussed below. A process maintains its causal state buffer to contain only the *latest* (causally speaking) state information for each process.

When an assert statement is executed in P_i, say at event e, $CGSBuffer_i$ will contain all components of $CGState(e)$.

A process receives state information from each process in the system by having the processes *piggyback* state information on application messages. When a process sends an application message, it piggybacks its *CGSBuffer* on the message. Process P_j acquires state information from P_i when P_i directly communicates with P_j or when P_i indirectly communicates with P_j. Process P_i directly communicates with P_j by sending a message to P_j. Process P_i indirectly communicates with P_j by sending a message to another process P_k and P_k either directly or indirectly communicates with P_j. If P_i does not directly or indirectly communicate with P_j, then P_i does not contribute to P_j's causal global state. In this case, $LCP(e, i)$ does not exist.

Consider the communication pattern shown in figure 4.6. P_2 receives state information for P_0 from two different sources: the message P_0 sends to P_2, and the message P_1 sends to P_2. When P_1 and P_2 communicate, P_2 requires a mechanism for determining the causally latest value of x. P_2 has one value of x in $CGSBuffer_2$ from its direct communication with P_0, and a new arriving value of x is piggybacked on P_1's message to P_2. In fact, the newly arriving value of x is stale and should not overwrite the tuple for x in $CGSBuffer_2$. Vector time [29] is the mechanism we adopt for determining the latest causal values associated with variables.

Timestamping a set of events with vector time has been shown to be isomorphic to the causal partial order on those events [33]. Each P_i maintains a vector V_i of N integers, $(V_i[0], \ldots V_i[N-1])$, where $V_i[i]$ is the counter of the number of events which have occurred on P_i. $V_i[i]$ is incremented before each event in P_i. $V_i(e)$ is the vector time of event e of process P_i. The vector time associated with event e is also associated with the state resulting from e. The rules for maintaining asynchronous vector time are:

1. Initially, for each P_i, $V_i[j] = 0$ for $0 \leq j < N$
2. $V_i[i] = V_i[i] + 1$ when an event occurs on P_i.
3. Suppose P_i sends a message to P_j, and e_i and e_j are the corresponding send and receive events, respectively. If $V_i = (V_i[0], \ldots, V_i[N-1])$ corresponds to e_i and $V_j = (V_j[0], \ldots, V_j[N-1])$ corresponds to e_j, then as a result of P_i and P_j communicating, P_j updates its vector clock to

$$V_j(e_j) = \mathrm{MAX}((V_i[0], \ldots, V_i[i]+1, \ldots, V_i[N-1]), (V_j[0], \ldots, V_j[j]+1, \ldots, V_j[N-1])),$$

where MAX designates component-by-component maximum.

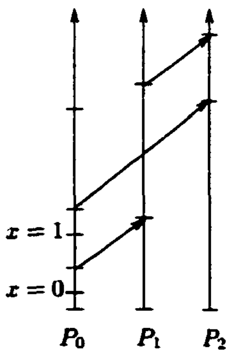


Figure 4.6: *Latest* State

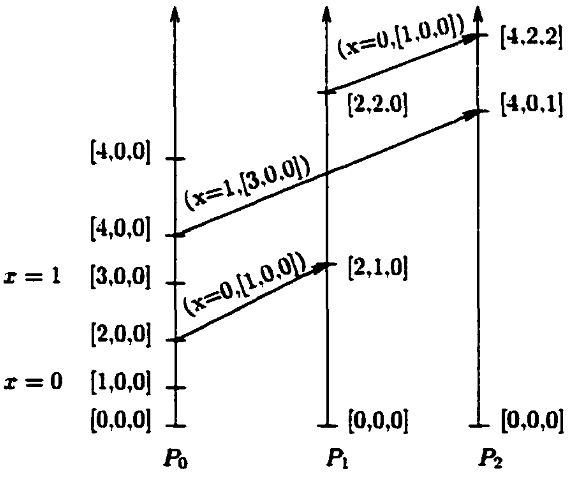


Figure 4.7: Vector Time

Vector time can be used to indicate the relative "causal timeliness" of state information. Suppose P_j propagates a state datum stamped with a vector time to P_k. If the datum is a variable of P_j, it will be timestamped with the vector time of P_j just before the communication with P_k. If, however, the datum is not local to P_j or P_k, then it must reside in $CGSBuffer_j$ (it is being propagated in order to handle the indirect communication), and the vector time of the component's tuple in $CGSBuffer_j$ will be used. Upon receipt of the vector timestamped datum (assume the datum resides at P_i), the ith component of its timestamp is compared with the ith component of the vector timestamp of the tuple in $CGSBuffer_k$ associated with the appropriate variable of P_i. If the ith vector component of the tuple in $CGSBuffer_k$ is greater than or equal to the ith component of the timestamp on the incoming datum, then the copy in $CGSBuffer_k$ is the valid latest causal value of the variable, and the tuple is not updated. Otherwise, the incoming datum is causally later than the value of the variable stored in $CGSBuffer_k$, and the tuple must be overwritten with the incoming datum.

Figure 4.7 is derived from figure 4.6 by adding vector time. Note that P_2 receives two copies of the datum for P_0's variable x. It receives x with value 1 and vector timestamp $[3,0,0]$ when P_0 sends a message to P_2. The tuple $(P_0, x, 1, [3,0,0])$ is inserted into $CGSBuffer_2$. When P_0 sent a message to P_1, the tuple $(P_0, x, 0, [1,0,0])$ was inserted into $CGSBuffer_1$. When P_1 sends a message to P_2, P_1 forwards a datum for x with value 0 and vector timestamp $[1,0,0]$ to P_2 to account for the indirect communication between P_0 and P_2. However, when P_2 receives the second datum for x, the first component of the datum's timestamp, 1, is compared to the first component of the vector timestamp for P_0's x in

CGSBuffer$_2$. 3. Process P_2 then knows to discard the second datum.

The causal state propagation is implemented by the protocol shown in figure 4.8 on each communication. This protocol is not intended as a final implementation but as a foundation for a more efficient result.

Protocol: Causal State Propagation

```
$P_i$ sends to $P_j$:
    $V_i[i] = V_i[i] + 1$
    send (msg, $V_i$.$CGSBuffer_i$) to $P_j$

$P_j$ receives from $P_i$:
    $V_j[j] = V_j[j] + 1$
    receive (msg$_{buffer}$, $V_i$, Tmp_Buffer) from $P_i$
    Update($CGSBuffer_j$,Tmp_Buffer)
    $V_j = MAX(V_i,V_j)$
    consume(msg$_{buffer}$)
```

Figure 4.8: Propagation Protocol

To simplify the presentation. the above pseudo-code assumes that each process keeps its local state in its causal state buffer along with remote state components it has acquired via message passing. The Update procedure in figure 4.9 is invoked to alter the local causal state buffer based on this communication.

Procedure Update(B1,B2)

Updates local state buffer B1 based on contents of remote buffer B2.
Recall that buffer tuples contain fields (Pid. var. value. V)

```
for all tuples $T$ in B2 do
    if ($T$.Pid,$T$.var,*,*) not in B1
        insert $T$ in B1
    else /* Let $T'$ be the tuple in B1 matching $T$. */
        if $T'$.V[$T'$.Pid] < $T$.V[$T$.Pid]
            replace $T'$ with $T$
endfor
```

Figure 4.9: Update Causal State Buffer

The asynchronous communicants piggyback state information on all messages to track the causal global state. Although this does guarantee that the causal global state is immediately available for the process evaluating the assert, we piggyback all state information on all messages. Optimizations of this naive approach are addressed in chapters 5, 6 and 7.

Chapter 5

Optimization

5.1 Timing Results

Our evaluation of an assert statement alters the distributed system by piggybacking data on existing messages, resulting in increased message sizes. Intuitively, one way message transmission time is linear in size of the message. To verify linear transmission time increases, we have conducted an experiment with datagram communication on real systems.

Two processes, P_{sender} and $P_{receiver}$, communicate with each other through UDP/IP datagrams. P_{sender}'s and $P_{receiver}$'s only function is communicating with each other. This provides an adequate environment to measure the full impact of increased message length on execution time. P_{sender} sends to $P_{receiver}$ 1,000 datagrams, and for each datagram sent, P_{sender} waits for an acknowledgment from $P_{receiver}$ before sending the next datagram. One thousand samples of P_{sender}'s execution time are gathered to obtain a sufficient number of samples to determined P_{sender}'s average execution time with 95% confidence. For the first 1,000 samples, the datagram size is 50 bytes. The datagram size is incremented by 50 bytes, and samples are gathered for each datagram size. This experiment is completed after the samples are gathered for a 3500 byte datagram. The experiment is conducted between two Sun workstations running SOLARIS 1.1. The average execution times and associated 95% confidence intervals are plotted in figure 5.1. The same experiment is conducted on two additional machine platforms and similar results are obtained. For one platform. the sender is an IBM RS6000 workstation running AIX 3.23. and the receiver is a Sun workstation. For the other platform. the sender is a DECstation 5000 workstation running Ultrix 4.2A. and the receiver is a Sun workstation.

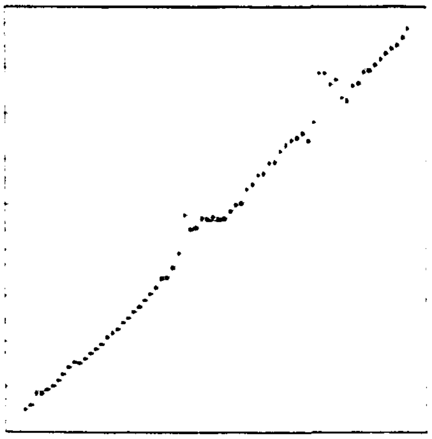


Figure 5.1: Datagram experiment

In all three datagram experiments, the execution times are roughly linear as message size increases. Common to all three experiments is a fluctuation in execution time when the message size is approximately 1500 and 3000. The significance of these numbers lies in the maximum transmission unit (MTU) for the Sun which is 1500 bytes, and datagram fragmentation into packets occurs for every MTU. The Internet protocol (IP) layer, or network layer, is responsible for fragmentation into packets and reconstruction of the datagram. The overhead of fragmentation occurs when the message size reaches an MTU multiple. The important conclusion gained from the datagram experiment for assert statement evaluation is that increasing the message size does increase the execution time of a distributed system, but the increase is linear in the size of the piggybacked data.

5.2 Piggybacking messages

The naive implementation described for a causal distributed assert statement constructs a causal state buffer consisting of each process's causally latest state information. Each process piggybacks its entire causal state buffer on the application messages. This does ensure that all data is available for assert statement evaluation, but one expects that a majority of the data is not necessary for the evaluation. The amount of state information gathered in the causal state buffers and piggybacked onto messages can be reduced by preprocessing with regard to the assert statement.

If the messages that are not necessary for delivering the *CGState* can be identified, the number of messages marked for piggybacking can be reduced. The *LCP events* are the means by which we reduce the number of messages piggybacking state information. The first step in achieving our reduction is showing that *LCP* events are communication events.

Lemma 5.1 *For event e of P_i, each $LCP(e,j), j \neq i$, is a communication event.*

Proof. According to the happens before relation and the definition of *LCP* events (definition 4.1), if there exists an $LCP(e,j)$, then there must exist a communication event f in

P_j such that $f \rightarrow e$, and there does not exist another communication event f' in P_j such that $f \rightarrow f' \rightarrow e$. Events in P_j either occur before or after f. Consider an event $g \neq f$ in P_j. There are two cases:

1. Assume $g \rightarrow f$. Since $g \rightarrow f \rightarrow e$ and an event that occurs before f is not $LCP(e, j)$, it follows that g is not $LCP(e, j)$.
2. Assume $f \rightarrow g$. Since there does not exist a communication event after f that happens before e, we know $g \not\rightarrow e$. Therefore g is not the $LCP(e, j)$.

We can conclude from 1. and 2. that $LCP(e, j)$ is the communication event f. ■

For asynchronous message passing, each LCP event is a send. We will be concentrating on results for asynchronous message passing, but our results can easily be extended to (the less practically significant) synchronous message passing.

Lemma 5.2 *For event e of P_i, each $LCP(e, j)$, $j \neq i$, is a send event.*

Proof. We know from lemma 5.1 that each LCP event is either a send or receive event. Assume that the event $e_{j \neq i} = LCP(e, j)$ is a receive event. For e_j to be the $LCP(e, j)$, $e_j \rightarrow e$ and there does not exist another event e'_j such that $e_j \rightarrow e'_j \rightarrow e$ (definition 4.1).

For an event of P_j to happen before an event of P_i process, there must exist a causal chain of communication events from P_j to P_i where the causal chain begins with P_j sending a message and ends with P_i receiving a message (definition of $\rightarrow$). For e_j to happen before e there must exist a send event e''_j in P_j that happens after e_j and that happens before the event e. Since $e_j \rightarrow e_j'' \rightarrow e$ and e_j'' is a send event, the receive event e_j can not be an LCP event. ■

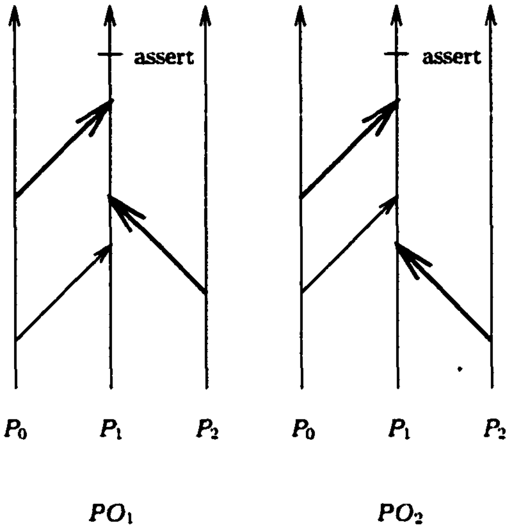


Figure 5.2: LCP and LCP' events of the Assert Event

Corresponding to each LCP send event is a receive event, denoted LCP'. A causal cut for event e consists of LCP send events. The LCP and LCP' events of the distributed program shown in figure 4.2 are shown in figure 5.2. The wider communication line indicates the message of the LCP and LCP' events. The LCP and LCP' events of a partial order comprise the communication events that are sufficient for delivering the *CGState* data to the process evaluating the assert. Before proving this property, the following definitions are necessary:

Definition 5.1 *A* communication path *of length $t+1$ from e_j^0 to e_i^t, where t is odd and $j \neq i$, is a series of communication events $e_j^0, \ldots, e_i^t$ such that*

1. *e_j^0 is the only communication event of P_j in the path.*

2. e_i^t *is the only communication event of* P_i *in the path.*
3. $e_k^r \rightarrow e_l^{r+1}$, *where* $k \neq l$ *or* $k = l$, *and there does not exist an event* e' *that is an event of the path such that* $e_k^r \rightarrow e' \rightarrow e_l^{r+1}$.
4. *for* e_k^r *and* e_l^{r+1}, *where* $k \neq l$ *and* r *is even,* e_k^r *and* e_l^{r+1} *are a send/receive pair (*e_k^r *being the send and* e_l^{r+1} *being the receive), and*
5. *for* e_k^r, e_l^{r+1}, *where* $k \neq l$, *the next event of the path (if it exists) must occur on* P_l, *denoted* e_l^{r+2}, *and the the event following* e_l^{r+2} *is not an event of* P_l.

If $e_j^0, e_k^1, e_k^2, e_l^3, e_l^4, e_i^5$ is a valid communication path of length 6, e_j^0 is a send to P_k, e_k^1 is the receive corresponding to e_j^0, e_k^2 is a send to P_l, e_l^3 the receive corresponding to e_k^2, and e_l^4 is a send to P_i, and e_i^5 is the receive corresponding to e_l^4.

Definition 5.2 *A* non-repetitive *communication path is a communication path such that when two communication events of* P_k *occur in the path* $\ldots, e_k^r, e_k^{r+1}, \ldots$, *no other events of* P_k *can occur in the path.*

A non-repetitive communication path differs from a communication path in that

- if P_k has events in the path, $k \neq j$, and $k \neq i$, then exactly one send and one receive of P_k occurs in the path.
- P_j has exactly one event in path, the send event e_j^0, and
- P_i has exactly one event in the path, the receive event e_i^t.

A non-repetitive communication path is a special case of a communication path.

Lemma 5.3 *If a communication path exists from e_j^0 to e_i^t, then at least one non-repetitive communication path exists from P_j to P_i consisting of a subset of the events of the communication path.*

Proof by contradiction. Assume a communication path exists from e_j^0 to e_i^t but a non-repetitive communication path does not exist from P_j to P_i consisting of a subset of the events of the communication path.

Consider the communication path from e_j^0 to e_i^t.

Case 1 .

The communication path from e_j^0 to e_i^t is not a non-repetitive communication path due to there existing at least two send commands and two receive commands of the same process, P_k, $k \neq j, k \neq i$, in the path. Let $p = e_j^0, \ldots, e_k^{r-1}, e_k^r, \ldots, e_k^{r+l}, e_k^{r+l+1}, \ldots, e_i^t$ represent such a path where P_k is the only process that has multiple send and receives in the communication path. The events e_k^{r-1} and e_k^{r+l} are receive events of P_k, and the events e_k^{r+l} and e_k^{r+l+1} are the send events of P_k. We know from the definition of a communication path that $e_k^{r-1} \rightarrow e_k^r \rightarrow e_k^{r+l} \rightarrow e_k^{r+l+1}$. We also know that $e_j^0, \ldots, e_k^{r-1}$ is a non-repetitive communication path and that $e_k^{r+l+1}, \ldots e_i^t$ is a non-repetitive communication path, therefore $e_j^0 \ldots e_k^{r-1}, e_k^{r+l+1}, \ldots e_i^t$ is a non-repetitive communication path.

Case 2 .

The communication path from e_j^0 to e_i^t is not a non-repetitive communication path due to there existing in addition to the send command e_j^0 at least one send and receive of P_j in the path. Let $p = e_j^0, \ldots, e_j^{r-1}, e_j^r, \ldots, e_i^t$ represent such path. The event e_j^{r-1} is a receive event of P_j and e_j^r is a send event of P_j. We know from the definition of a communication path that $e_j^0 \rightarrow e_j^{r-1} \rightarrow e_j^r$. We can conclude that $e_j^r \ldots e_i^t$ is a non-repetitive communication path.

Case 3 .

The communication path from e_j^0 to e_i^t is not a non-repetitive communication path due to there existing in addition to the receive command e_i^t at least one send and receive of P_i in the path. Let $p = e_j^0, \ldots, e_i^{r-1}, e_i^r, \ldots, e_i^t$ represent such path. The event e_i^{r-1} is a receive event of P_i and e_i^r is a send event of P_i. We know from the definition of a communication path that $e_i^{r-1} \rightarrow e_i^r \rightarrow e_i^t$. We can conclude that $e_j^0 \ldots e_i^{r-1}$ is a non-repetitive communication path.

■

If a non-repetitive communication path exists from event e_i to event e_j, then event e_i happens before e_j. Also, if event e_i happens before event e_j, then there exists a non-repetitive communication path from P_i to P_j where the first event of the path happens after e_i and the last event of the path happens before e_j.

Lemma 5.4 *Event e_j happens before e_i if and only if there exists a non-repetitive communication path from a send of P_j that happens after e_j and a receive of P_i that happens before e_i.*

Proof

If $e_j \rightarrow e_i$, then there exist a non-repetitive communication path from a send of P_j that happens after e_j and a receive of P_i that happens before e_i.

Assume $e_j \rightarrow e_i$ but there does not exist a non-repetitive communication path that starts with a send event of P_j that happens after e_j and ends with a receive event of P_i that happens before e_i.

For $e_j \rightarrow e_i$, there must be communication path, $e_j^0 \ldots e_i^t$, such that $e_j \rightarrow e_j^0$ and $e_i^t \rightarrow e_i$ (e_j can be e_j^0 and e_i can be e_i^t). From lemma 5.3 we know that there must also exist at least one non-repetitive communication path from P_j to P_i that consist of a subset of the communication path $e_j^0 \ldots e_i^t$.

If there exists a non-repetitive communication path from a send of P_j that happens after e_j and a receive of P_i that happens before e_i then $e_j \rightarrow e_i$.

Proof. Let e_j' be the send event that happens after e_j and e_i' be the receive event that happens before e_i. From definition 5.2, we know that $e_j' \rightarrow e_i'$and therefore $e_j \rightarrow e_i$.

■

Theorem 5.1 *For each $LCP(e_i, j)$ event of $CC(e_i)$, there exists a non-repetitive communication path from $LCP(e_i, j)$ to an LCP' of P_i such that each event of the path is either an LCP event or an LCP' event.*

Proof.

CASE 1 For each $LCP(e_i, j)$ event of $CC(e_i)$, there exists a non-repetitive communication path from $LCP(e_i, j)$ to a receive event e_i' such that $e_i' \to e_i$

Let $e_j = LCP(e_i, j)$. From lemma 5.2, we know e_j is a send event. From definition 4.1 we know $e_j \to e_i$. From lemma 5.4, we know there exists a non-repetitive communication path from e_j to some receive e_i' such that $e_i' \to e_i$.

CASE 2 The non-repetitive communication path that exists from $LCP(e_i, j)$ to receive event e_i' consists of LCP and LCP' events.

CASE 2.A The send events of the path are LCP events.

In order for every non-repetitive communication path that exists from $LCP(e_i, j)$ to event e_i' not to consist of LCP send events, in each path there must exist at least one send event in P_k, e_k, that is not an LCP event.

Since e_k is a send event of a non-repetitive communication path from e_j to e_i', we know from definition 5.2 that $e_k \to e_i$. For e_k to not be an LCP event, there must exist another event, e_k', of P_k such that $e_k \to e_k' \to e_i$; i.e., e_k' is $LCP(e_i, k)$. From this follows a contradiction. If e_k' exists then there does exist a non-repetitive communication that includes $LCP(e_i, k) = e_k'$ according to 5.4. If e_k' does not exist, then e_k is the $LCP(e, k)$.

CASE 2.B The receive events of the path are LCP' events.

We know from case 2.A that the sends of a non-repetitive communication path from e_j to e_i' are LCP events. And from definition 5.2, the receives of the path correspond to the sends, therefore the receives are LCP' events.

■

The following theorem is the basis for reducing the number of messages on which state information is piggybacked.

Theorem 5.2 *If the state data of the processes are piggybacked only on the messages of the* $LCPs$ *and* $LCP's$ *of the* $CC(e)$ *of the current execution, the process executing the assert statement is delivered exactly the CGState prior to the assert statement's execution.*

Proof. This follows directly from theorem 5.1. From theorem 5.1 we know there exists a non-repetitive communication path from each LCP event to an LCP' of P_i that consists of LCP and LCP' events. If a process only piggybacks its local state information, and the state information it has received from other processes, on the message corresponding to its LCP event, the data will be received by P_i's LCP' event(s). ■

Our first objective in reducing the amount of piggybacked data is to analyze the source code of the distributed processes to determine all possible partial orders and the LCP and LCP' events of each partial order. Chapters 6 and 7 explain our static analysis methods for achieving this objective.

Chapter 6

Static Analysis

6.1 Goals of Static Analysis

The causal state propagation protocol presented in chapter 4 satisfies the two properties of the distributed assert:

A1 The asserted predicate is evaluated during execution of the program. We do not generate and analyze traces *post mortem.*

A2 No additional synchronization or message passing is added to the original distributed application in support of the distributed assert statement. We do increase the size of some application messages.

This protocol can be improved by reducing the amount of data piggybacked. We know from the timing experiments in chapter 5 that these reductions will result in less interference with message transmission time. Hence, the "natural" time in the program can be preserved. The objective of static analysis is to determine which send and receive events are the *LCP* and LCP' events of the assert. By piggybacking data only on these messages, the assert statement is evaluated with the *CGState* and the amount of data piggybacked is reduced.

The first step in our static analysis is to examine the source code of each process and generate a flow graph. From the flow graphs, communication analysis matches send and receive events to generate a tree called a partial order graph (POG). We prove that the POG represents all partial orders of the distributed system (property 6.12) and that each path of the POG from root to a leaf node represents a unique partial order (property 6.13). After analyzing the source code and generating the POG, our technique detects the LCP and LCP' events for an assert statement. Properties 6.14 and 6.15 are our concluding properties of our analysis, and these properties establish that our technique for identifying LCP and LCP' events is valid.

By performing this analysis before execution, a reduction in the amount of piggybacked data is achieved by tagging the LCP and LCP' events as piggybacking events, and properties A1 and A2 are upheld. Before presenting algorithms for identifying the LCP and LCP' events, Taylor's static analysis technique is discussed.

6.2 Static Analysis in the Distributed Domain

Taylor [42] has developed an algorithm for statically analyzing the synchronous communication of a distributed program. Synchronous communication occurs when the sending process blocks until the message is received by the destination process. Effectively, the rendezvous of the send and receive appears as a distributed assignment, $var = expr$, that takes place in the context of both processes. The sender evaluates *expr*, and the receiver stores the value into *var*.

The transformation of the $\rightarrow$ relation into the synchronous communication regime only affects condition (2) of the three conditions stated in chapter one for asynchronous communication. All the conditions are repeated for completeness: (1) if e and f are events in the same process, and e happens before f, then $e \rightarrow f$; (2) if e and f are a send and receive pair which rendezvous, consider e/f as a single event (the rendezvous which effects the distributed assignment) on both the sending and receiving processes; and (3) if $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$.

Taylor's algorithm matches all possible synchronous communications for the programming language Ada [44]. The following is a discussion of Taylor's technique as modified (by us) to deal with communicating sequential processes (CSP)[15] . CSP is a well-defined language which supports strictly synchronous communication. The semantics of CSP constructs have been formalized, and sound and relatively complete verification methodologies for CSP are well-established [20, 21, 3]. Two message transmission operations are available in CSP. Process P_i sends a message, *msg_out*, to process P_j by a matching send/receive pair. P_i executes the send operation j!*msg_out*, and P_j executes the receive operation i?*msg_in*.

As part of the static analysis, each process is represented by an annotated flow graph G_i, which is a modification of a sequential program's flow graph derived from flow analysis [13]. A distributed program is represented by $\{G_0, G_1, \ldots, G_{N-1}\}$ such that $G_i = \{V_i, A_i, r_i\}$ where V_i is the set of nodes, A_i is the set of arcs, and $r_i \in V_i$ is the root node of G_i. In contrast to a flow analysis flow graph that usually represents all statements, nodes of

G_i represent only the statements necessary for communication analysis. In particular, the following commands are represented by nodes: send and receive communication commands, guards comprised of communication commands, and repetitive and selective constructs comprised of communication commands. In addition, the root node of G_i represents the beginning of P_i's execution (begin node), and the node whose out-degree is zero represents the completion of P_i's execution (end node). Arcs show the possible paths of execution between the nodes, and all paths of G_i are assumed to be executable. Figures 6.1 and 6.2 demonstrate two distributed programs' flow graphs. The horizontal lines of the flow graph represent the nodes.

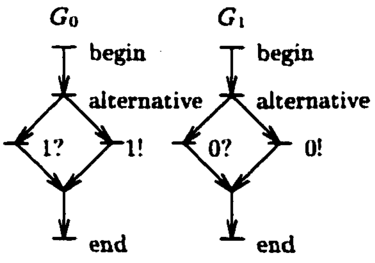

G_0
begin
alternative
1?
1!
end
G_1
begin
alternative
0?
0!
end

Figure 6.1: Flow graphs of a 2 process system

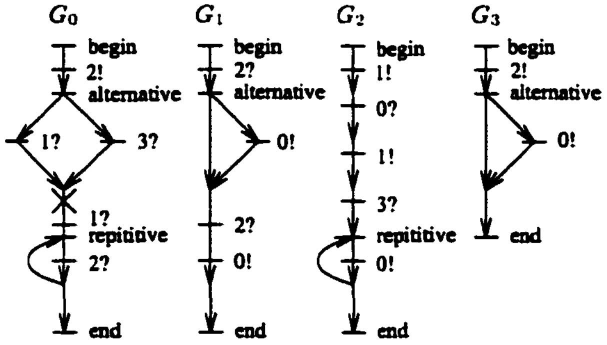

G_0
begin
2!
alternative
1?
3?
1?
repititive
2?
end
G_1
begin
2?
alternative
0!
2?
0!
end
G_2
begin
1!
0?
1!
3?
repititive
0!
end
G_3
begin
2!
alternative
0!
end

Figure 6.2: Flow graphs of a 4 process system

For any node v_i of G_i, the set of immediate successor nodes is the set of all nodes v_i' for which there exists a path p from v_i to v_i' in G_i such that there is no node v_i'' $(v_i'' \neq v_i$; $v_i'' \neq v_i')$ on the path from v_i to v_i'. Succ(v_i) denotes the set of immediate successors of v_i. Figure 6.3 list some of the successor sets for figures 6.1 and 6.2.

Taylor defines a concurrency state C as an ordered N-tuple $(v_0, v_1, \ldots v_{N-1})$ where each v_i is a node of G_i or is an inactive marker. Each v_i denotes the next node to be executed in P_i or indicates process inactivity. A concurrency state C has successor concurrency states

Successor sets of figure 6.1		Successor sets of figure 6.2	
G_0:	G_1:	G_0:	G_1:
succ(begin)=alternative	succ(begin)=alternative	succ(alternative)=1?,3?	succ(alternative)=0!,2?
succ(alternative)=1?,1!	succ(alternative)=0?,0!	succ(repetitive)=2?	
succ(1?)=end	succ(0?)=end	succ(2?)=repetitive,end	
succ(1!)=end	succ(0!)=end		

Figure 6.3: Successor sets

based on the successor sets of the nodes of C. A concurrency state $C' = (v'_0, v'_1, \ldots, v'_{N-1})$ is a successor of C, SUCC(C), if and only if

1. For all $i, 0 \leq i \leq N - 1$, either
    - (a) $v'_i \in \text{succ}(v_i)$,
    - (b) $v'_i = v_i$, or
    - (c) $v_i = \text{end}$ and $v'_i = \text{inactive}$
2. There exists at least one v'_i which represents application of case a or c.
3. Adherence to the communications semantics of CSP is reflected in the application of the three cases a-c. If v_i is a send or receive command, v_i can not be replaced by an element of succ(v_i) until the command's matching communication command occurs in the concurrency state. When a matching send/receive occur in the concurrency state, either both or neither are replaced by their respective successor nodes for the successor concurrency state.

A matching send (v_i) and receive (v_j) in a concurrency state indicates the CSP communication between P_i and P_j can occur. The communication between P_i and P_j is an *i/o rendezvous* between P_i and P_j.

A nonterminal concurrency state has at least one successor state, and a terminal concurrency state has no successor states. Taylor's *concurrency history* is a sequence of concurrency states $C_0, C_1, \ldots, C_m$ such that

1. C_0 =(begin$_0$, begin$_1$, ..., begin$_{N-1}$), C_0 represents the initial state of the distributed computation.
2. For $i, 0 \leq i \leq m-1. C_{i+1} \in \text{SUCC}(C_i)$

A *proper concurrency history* is a finite concurrency history such that C_m has no successors; i.e., C_m is a terminal state. A *complete concurrency history* of a distributed system is the collection of all possible proper concurrency histories. A directed graph provides a visual representation of a complete concurrency history, where each node of the graph represents a concurrency state. For the distributed program in figure 6.1, the complete concurrency history is shown in figure 6.4.

Relating Taylor's algorithm to previously defined distributed system terminology, we see that each proper concurrency history corresponds to a possible total order of the synchronous communications. A proper concurrency history where C_m does not contain all inactive markers represents an execution that does not allow all the processes to complete their execution. For example, if process P_i executes the receive $j?$, but P_j does not send a message to P_i, then P_i hangs on the receive and can not complete execution. The complete concurrency history corresponds to all possible communication patterns since all execution paths are considered possible.

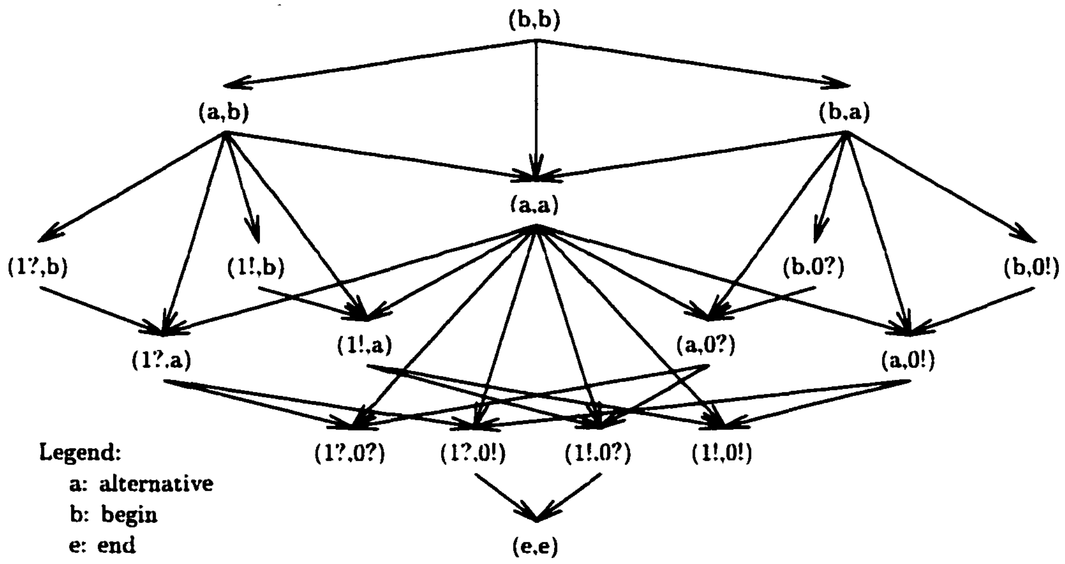


Figure 6.4: Complete Concurrency History of figure 6.1

Taylor's algorithm has been modified and expanded for various. distributed system's applications [27, 45, 22, 8, 26]. We have developed algorithms, motivated by Taylor's work, designed to identify the LCP and LCP' messages in each process for an assert statement.

6.3 Communication Analysis for Asynchronous Message Passing

In this work, the processes of a distributed program are written in the programming language C. The language has been augmented with three commands: async_send, async_recv, and assert. The statements async_send and async_recv are for asynchronous communication between processes and are described in detail in chapter 1. The assert command has the format assert(P) where P is a predicate. The predicate P is a boolean expression over the variables of the distributed program. Currently, the placement of async_send, async_recv and assert statements is restricted to the main function of the program. In this chapter, the language does allow nested if and if/else constructs, but it does not allow loops. This is done for ease of presentation. Loops are added to the language and handled by our analysis in chapter 7.

Each process P_i is represented by a control flow graph (FG_i). A distributed program is represented by { $FG_0, FG_1, \ldots, FG_{N-1}$ } such that FG_i={V_i, A_i, r_i } where V_i is the set of nodes, A_i is the set of arcs, and $r_i \in V_i$. The root node r_i represents the start of P_i's execution. The nodes of FG_i represent either computation statements or control constructs of the source code. Assignment, **async_send, async_recv**, and **assert** statements are classified as computation statements. The **if** and **else** constructs and begin and end delimiters are classified as control constructs. An end node represents the completion of P_i's execution. The arcs represent P_i's flow of execution. If an arc exists from node n to node n', n' can be executed following the execution of n. Although multiple branches may exist in the flow of execution, all flow of execution will terminate into a single end node.

Consecutive assignment statements that occur between control constructs and other types of computation statements are grouped into one node labeled ASSIGN. The commands **async_send, async_recv**, and **assert** are represented by SEND, RECEIVE, ASSERT nodes, respectively. The control constructs **if** and **else** are represented by nodes labeled IF and ELSE, respectively. The end of the **if** side of an **if/else** is represented by a END_IFSIDE node. The end of an **if** statement is represented by an END_IF node, and the end of the **else** side of an **it if/else** is represented by a END_ELSE node.

Each FG_i is generated by parsing the source code of P_i. First a lexical analyzer reads in the source code, and scans this code to recognize tokens. The software tool Lex has been used to produce the lexical analyzer.

The lexical analyzer passes the token to a parser. The tokens are parsed according to the ANSI C grammar that appears in Appendix A. This grammar is LR(1). The software tool Yacc helped produce the parser. The productions of the grammar that are relevant for describing the algorithm for generating the FG_is are `postfix_expression`, `unary_expression`, `assignment_expression`, and `selection_statement`.

Actions are embedded in these productions to call functions that collectively generate the control flow graphs. The algorithm, `Create_FG`$_i$`()`, implemented by these function calls, is described. For grouping consecutive assignment statements into one node, each assignment statement of the node is an entry in a linked list, and the assignment node references this linked list. A stack is employed to match the begin and end of control constructs. An entry in the stack is a pointer to a node of FG_i. The variable *TopStack* is a pointer to the node referenced by the top entry of the stack. The variable *CrtNode* is a pointer to the current node of FG_i. Associated with each node of FG_i are two fields that are for constructing the flow graph. The fields are *HoldPtr* and *AddEdgeFlag*. *HoldPtr* is a pointer to a node of FG_i and *AddEdgeFlag* is a boolean flag. The input for `Create_FG`$_i$`()` is the source code of P_i, and the output of `Create_FG`$_i$`()` is the flow graph FG_i.

```
Create_FG_i()                                        /* Input: P_i; Output: FG_i */
    Create the ROOT node of FG_i
    CrtNode = ROOT node
    if an assignment statement is recognized
        Add assignment statement to the tail of the linked list
```

```
if an async_send is recognized
    if the linked list is not empty
        AddNode(CrtNode, ASSIGN)                    /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, SEND)
if an async_recv is recognized
    if the linked list is not empty
        AddNode(CrtNode, ASSIGN)                    /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, RECEIVE)
if an assert is recognized
    if the linked list is not empty
        AddNode(CrtNode, ASSIGN)                    /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, ASSERT)
if an if statement is recognized
    if the linked list is not empty
        AddNode(CrtNode, ASSIGN)                    /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, IF)                                   /* for the if statement */
    Push CrtNode onto the stack
    TopStack = CrtNode
if an else is recognized
    AddNode(CrtNode, END_IFSIDE)
    if the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
    TopStack.HoldPtr = CrtNode                   /* Set HoldPtr of the IF node to the */
                                                     /* address of the END_IFSIDE */
    CrtNode = top entry of the stack
    CrtNode.AddEdgeFlag = true        /* Flag an edge needed from END_IFSIDE node */
                                      /* to the first node following END_ELSE node */
if the end of the else side of an if/else is recognized
    AddNode(CrtNode, END_ELSE)                 /* for the ending of the else side */
    if the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
    CrtNode.HoldPtr = TopStack.HoldPtr     /* Move the address of the END_IFSIDE */
                                                  /* node to the END_ELSE node */
    CrtNode.AddEdgeFlag = true      /* Flag an edge will be needed from END_IFSIDE */
                                   /* node to the first node following END_ELSE node */
    Pop the stack
if the end of an if statement is recognized
    AddNode(CrtNode, END_IF)                 /* for the ending of the if statement */
    if the linked list is not empty
```

```
            Set field in CrtNode to point to linked list
            linked list is set to empty
        CrtNode.HoldPtr = TopStack                /* Set the HoldPtr of END_IF node */
                                                      /* to the address of the IF node */
        CrtNode.AddEdgeFlag = true         /* Flag an edge will be need from the IF node */
                                           /* to the first node following the END_IF node */
        Pop the stack
    if the current control construct or statement is not recognized
        Generate an error and halt
    if the end of the source code is recognized
        AddNode(CrtNode, END)
        If the linked list is not empty
            Set field in CrtNode to point to linked list
            linked list is set to empty
end algorithm
```

The algorithm **Create_FG$_i$()** calls the algorithm **AddNode()**.

```
AddNode(CrtNode, type)
    NewNode = Allocate a node
    Create a directed edge from CrtNode to NewNode
    if CrtNode.AddEdgeFlag
        Create a directed edge from the node CrtNode.HoldPtr to NewNode
                        /* An edge is added either from END_IFSIDE or IF node to NewNode */
    if type = ASSIGN
        Set field in NewNode to point to assignment linked list
    CrtNode = NewNode
end algorithm
```

When a node is added to FG_i, if the previously added node is the end of the else side of an `if/else`, the END_IFSIDE and END_ELSE nodes must both have an edge to this newly added node. Figure 6.5 shows the adding of *NewNode*. The dashed lines indicate the edges **AddNode()** creates to *NewNode*. The END_ELSE is *CrtNode* so the edge from END_ELSE to *NewNode* is added by the second line of **AddNode()**. But creating the edge from END_IFSIDE to *NewNode* is more complicated. When END_IFSIDE is added to FG_i, the address of this node is stored in the IF node. This is accomplished with the following line from **Create_FG$_i$()**:

TopStack.HoldPtr = CrtNode

When the END_ELSE node is added, the address of the END_IFSIDE node is moved from the IF node to the END_ELSE node. This is accomplished with the following line from **Create_FG$_i$()**:

CrtNode.HoldPtr = TopStack.HoldPtr

By moving the address of the END_IFSIDE, when a new node is added and *CrtNode* is equal to END_ELSE, the address of the END_IFSIDE node is available in *CrtNode* to add the edge from END_IFSIDE to *NewNode*. The flag *AddEdgeFlag* of the END_ELSE node is set to true to indicate that function **AddNode()** should add an edge from the END_IFSIDE node to *NewNode*.

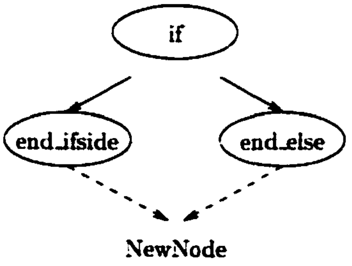


Figure 6.5: if/else portion of control flow graph

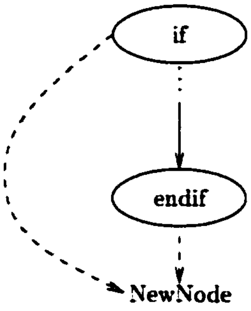

if
endif
NewNode

Figure 6.6: if portion of control flow graph

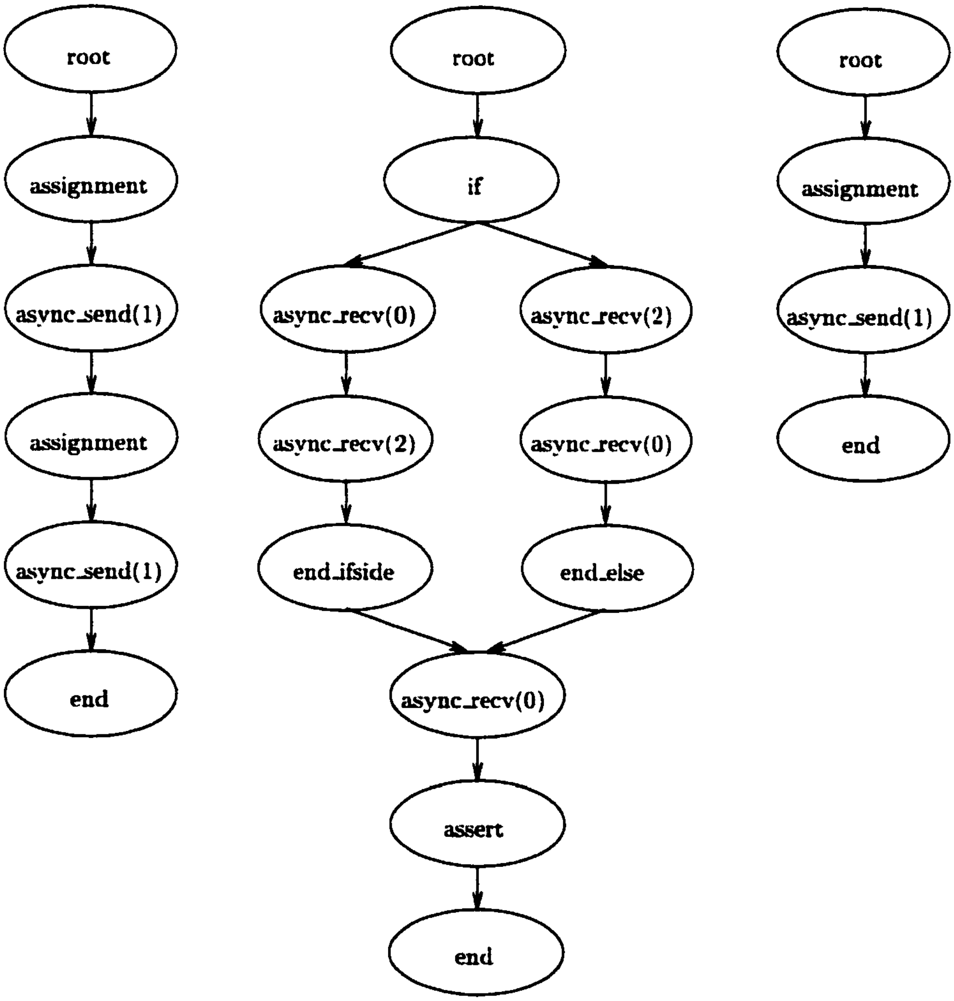

root
assignment
async_send(1)
assignment
async_send(1)
end
root
if
async_recv(0)
async_recv(2)
async_recv(2)
async_recv(0)
end_ifside
end_else
async_recv(0)
assert
end
root
assignment
async_send(1)
end

Figure 6.7: Flow graphs for a simple 3 process system

Figure 6.6 shows the adding of a new node when *CrtNode* is a END_IF node. When the END_IF node is created, the address of the IF node, which is available on top of the stack, is stored in the END_IF node. This is accomplished with the following line from **Create_FG$_i$()**:

CrtNode.HoldPtr = TopStack

When *NewNode* is added to FG_i, the address of the IF node is available in *CrtNode* so that an edge from the IF node to *NewNode* can be created by **AddNode()**. The flag *AddEdgeFlag* of the END_IF node is set to true to indicate that function **AddNode()** should add an edge from the IF node to *NewNode*. Figure 6.7 is the resulting control flow graphs for the source code of figure 4.2. Another example of a flow graph is figure 6.9 which is the result of one process's source code with nested if constructs shown in figure 6.8.

```
P_i::
{
    a = random number
    b = a -1
    if (a > 1) {
        if (b > 1) {
            async_send(0, a)
            b = b * 2
        }
        else {
            async_recv(0, b)
            a = b * 2
        }
    }
    a = b
}
```

Figure 6.8: P_i's source code

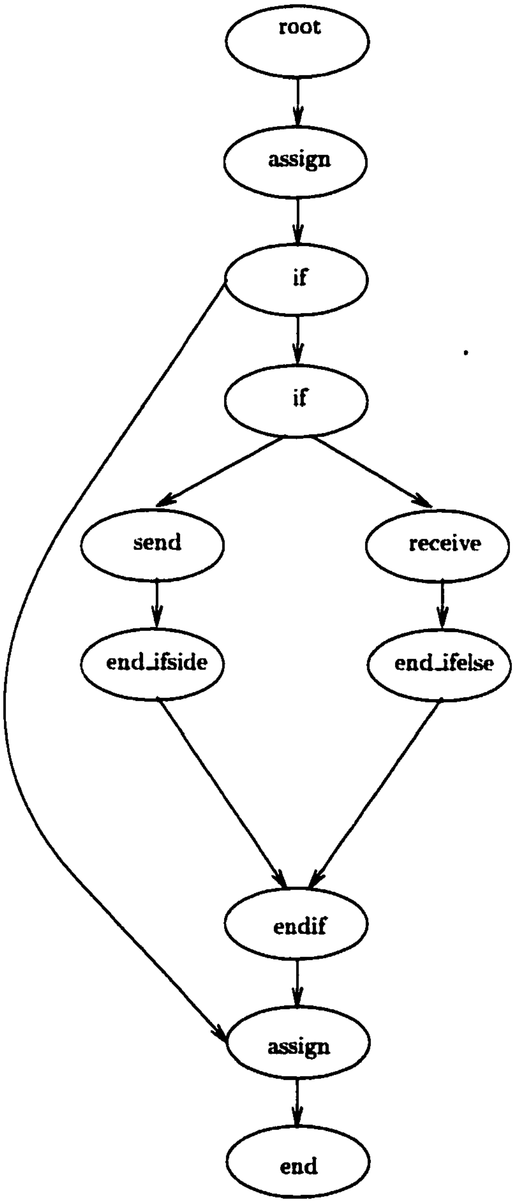

root
assign
if
if
send
receive
end_ifside
end_ifelse
endif
assign
end

Figure 6.9: FG_i

As we know from the definition of FG_i, the nodes of FG_i represent syntactic constructs in the source code of P_i. The execution of P_i may be viewed as a traversal of FG_i, starting at the root node and ending at the end node. An event in the execution of P_i corresponds to the locus of control passing through a node of FG_i. In the remaining discussion of the flow graphs, the symbol representing a node of FG_i is also used to represent the event corresponding to the execution of the source associated with that node. The context of the use of the symbol determines whether it is representing a node of FG_i or an event. For example, if the context is $a \rightarrow b$, the symbols a and b represent events.

We make use of the following properties of a FG_i.

Property 6.1 *A path exists from node a to node b in FG_i if and only if $a \rightarrow b$ when both a and b are executed.*

Proof.

PART 1. If a path exists from node a to node b, then $a \rightarrow b$ when both a and b are executed.

CASE 1. First consider a process's source code in which no `if` or `if/else` statements exist. The resulting control flow graph contains only nodes of type ROOT, ASSIGN, SEND, RECEIVE, ASSERT and END, and one path exists from the ROOT node to the END node. Since execution must follow the edges inFG_i, a path from a to b implies $a \rightarrow b$.

CASE 2. Now consider the case in which `if` and `if/else` constructs exist. According to the construction algorithm, flow graphs of the form shown in figure 6.10 are generated for an `if` control construct and an `if/else` control construct.

For the if control construct, the branch of control resulting from the falsifying of the `if` statement is the edge from the IF node to S2. When the condition of the `if` statement evaluates to false, the statements represented by S2 are executed next, and therefore IF → S2. Let node a occur before the IF node in FG_i, and let node b occur after S2 as are shown in figure 6.10. Two paths exist from node a to node b. Independent of which path is followed in an execution P_i, $a \rightarrow b$.

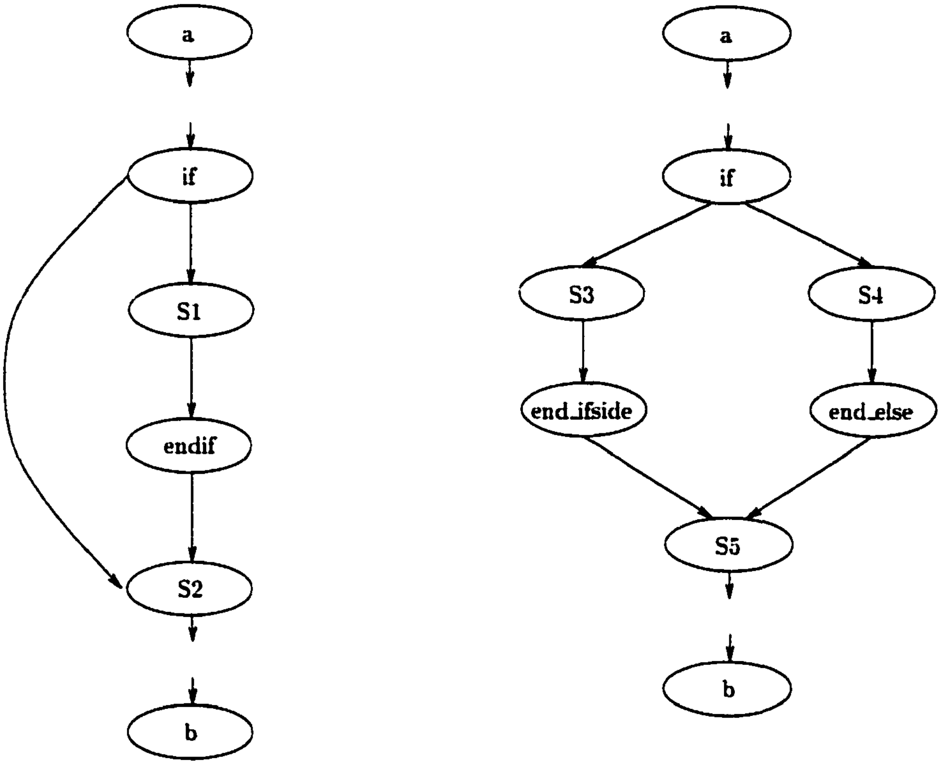


Figure 6.10: `if` and `if/else` flow graphs

Next consider the `if/else` control construct. For the branch resulting from a true evaluation of the condition of the `if/else`, a path is created by `Create_`FG_i`()` from the IF node to the END_IFSIDE and from the END_IFSIDE to S5. If the condition evaluates to true, the statements represented by S3 are executed then the statements represented by S5 are executed. Therefore, IF → S3 → END_IFSIDE → S5. For the branch resulting from a false evaluation of the if statement, a path exists from the IF node to the END_ELSE and from the END_ELSE to S5. If the condition evaluates to false, IF → S4 → END_ELSE → S5. Let node a occur before the IF node in FG_i, and let node b occur after S5 in FG_i as are shown in figure 6.10. Two paths exist from node a to node b. Independent of which path is followed in an execution P_i, $a \rightarrow b$.

PART 2. If $a \rightarrow b$ when both a and b are executed, then a path exists from node a to node b in FG_i.

Assume $a \rightarrow b$ but that a path does not exist from node a to node b in FG_i. Two cases can exist in FG_i such that a path does not exist from node a to node b.

1. there exists a path from node b to node a, or
2. node a occurs in one branch of a `if/else` and node b occurs in the other branch of the `if/else`.

If a path exists from node b to node a, we know from part 1 of this proof that $b \rightarrow a$ when both b and a are executed. This contradiction stands in to our assumption that $a \rightarrow b$, therefore a path cannot exist from b to a. Now consider case 2. Only one branch of the `if/else` will be executed for any execution of P_i. Therefore, $a \not\rightarrow b$. So we can conclude that if $a \rightarrow b$, a path exists from node a to node b.

■

Property 6.2 *Each path of FG_i from ROOT node to the END node represents an execution of P_i.*

Proof. Assume there exists a path from the root node to the end node that does not represent an execution of P_i. For such a path to exist, there must exist at least two nodes v and v' where v is a parent of v' , and it is not possible that $v \rightarrow v'$ for any execution of P_i. This is a contradiction of property 6.1. ■

Property 6.3 *For each path, the occurrence of the nodes in the path represents the total order of events if this path is executed.*

Proof: For each statement and control construct of the source code, a node is generated in FG_i (algorithm `Create_`FG_i`()`). From this observation of `Create_`FG_i`()` and properties 6.1 and 6.2, it follow that this property is true. ■

Property 6.4 *FG_i represents all execution paths of P_i.*

Proof. This property may be falsified under two conditions:

CONDITION 1. Flow graph FG_i only represents a subset of execution paths of P_i. We know from `Create_`FG_i`()` that every statement and control construct is represent in FG_i. For a path not to be represented in FG_i, one or more directed edges between nodes are omitted. Three cases exist when an edge can be omitted:

1. an edge from current node to new node is not added,

2. an edge from END_IFSIDE node to first node following the END_ELSE node is not added, or
3. an edge from IF node to first node following the END_IF node is not added.

For any of these cases to occur, the `AddNode()` algorithm is contradicted.

CONDITION 2. FG_i represents an invalid execution of P_i. For this to be true, at least one path from the ROOT node to the END node represents an invalid execution of P_i. This contradicts property 6.2.

■

For each communication node, v, of FG_i, an immediate successor set $S(v)$ is determined from FG_i. Node v' is an immediate successor of node v if

1. there exists a path from v to v',
2. v' is a communication node or END node, and
3. there does not exist a communication node v'' on the path from v to v' such that $v'' \neq v'$.

Concurrency communication states (CCSs) are generated from the flow graphs $\{FG_0, FG_1, \ldots FG_{N-1}\}$ of the constituent processes of the distributed system. Each CCS is an ordered N-tuple $(v_0, v_1, \ldots, v_{N-1})$ where v_i is the root node of FG_i, a communication node of FG_i, or the END node of FG_i. In the examples, an underscore denotes the END node. If v_i is a communication node, v_i denotes the next communication command to be executed in P_i. The communication commands of a CCS represent the events that may occur concurrently. Not all communication commands are ready to be executed: i.e., a receive is not ready if its corresponding send has not been executed. All the communication commands of a CCS that are ready to execute are concurrent. A series of CCSs are generated, as described shortly, to mimic the execution of the distributed system represented by $\{FG_0, FG_1, \ldots, FG_{N-1}\}$. Collectively, a tree, H, of CCSs is generated that represents all the possible partial orders $\mathcal{P}$ of the distributed system. Figure 6.11 is an example of an H tree where each node of the tree represents a CCS. The concurrency among the communication events is preserved in H by not imposing a total order on the concurrent events.

Associated with each send command in a CCS is a counter. If v_i is a send to P_j, the counter associated with v_i is how many messages have been sent to P_j including this send. Assume we have a four process system, and $v_1 \in CCS$ is equal to 5:async_send(0). This five means four messages have been sent collectively to P_0 from P_1, P_2 and P_3 prior to this message. Associated with each receive command whose matching send command has already been executed is also a counter. If v_j is a receive command and has an associated counter, the counter is how many messages have been received by P_j including the message received with v_j.

The initial concurrency communication state, CCS_0, contains the root node of each flow graph $\{FG_0, FG_1, \ldots, FG_{N-1}\}$, $CCS_0 = (r_0, \ldots, r_{N-1})$. Successor CCSs of CCS_0 are determined from $\mathrm{S}(r_i), 0 \leq i < N$. The successors of CCS_0 are a set of concurrency communication states denoted by SUCC(CCS_0). The following steps determine SUCC(CCS_0):

1. Generate a successor of CCS_0 by replacing each r_i with an element of $\mathrm{S}(r_i)$; i.e.,

$CCS = (v_0, \ldots, v_{N-1})$ is an element of SUCC(CCS_0) if each v_i is an element of S(r_i).

2. Generate SUCC(CCS_0) by repeating step 1 until all unique CCSs are generated from the root nodes' immediate successor sets. The number of successor CCSs of CCS_0 is $|\mathrm{S}(r_0)| * \cdots * |\mathrm{S}(r_{N-1})| = |\mathrm{SUCC}(CCS_0)|$

A CCS, where each v_i is a communication node or an inactive marker, has at least one successor, $CCS' = (v'_0, v'_1, \ldots, v'_{N-1})$, if CCS has at least a send command or a *ready* receive command. If node n of H represents the concurrency communication state CCS, the successors of CCS are represented in H as the children nodes of n. The predecessor of CCS is represented in H as the parent (immediate ancestor) of n. A ready receive means that the necessary send command for this receive command occurred in the predecessor of the CCS or in a ancestor of CCS. A message queue, *Msg_Q*$_i$, is maintained for each process. If $v_j \in CCS$ is a send command to P_i, the entry j is added to the queue *Msg_Q*$_i$ following the generation of SUCC(CCS). If v_i is a receive from P_j and *Msg_Q*$_i$ contains a j, the receive is *ready* and the first j in *Msg_Q*$_i$ is removed.

Associated with each *Msg_Q*$_i$ is a counter that is incremented each time an entry is placed in the queue. The current value of the counter is appended to an entry when it is added to *Msg_Q*$_i$. An entry in *Msg_Q*$_i$ has the format <counter, process id>. The value of counter is also appended to the send entry of the CCS node of H that generated the entry in *Msg_Q*$_i$. Send commands that are syntactically identical in a process's source code are distinguished in the CCS nodes of H by their associated counter. When a receive v_i from P_j is ready, the counter associated with the first j entry in the queue *Msg_Q*$_i$ is appended to the receive entry in the CCS node of H. Not only are syntactically identical receives distinguishable, the counter provides a method to match sends with corresponding receives. The use of this counter for matching sends and receives will be seen in a later algorithm.

A *CCS* may contain multiple sends and ready receive commands. For example, *CCS* = (async_send(1), async_send(0), async_recv(3),-) has the two sends, and a possible ready receive. If Msg_Q_2 has the entry <*counter*, 3> to indicate that P_3 has sent a message to P_2 but the message has not been received by P_2, v_2 (v_2 = async_recv(3)) is a ready receive. If v_2 is a ready receive, the value of *counter* is appended to async_recv(3) in the *H* node. If *CCS* has no sends and no ready receives, *CCS* has no successor states. The successor concurrency communication states of *CCS*, SUCC(*CCS*), are determined from the immediate successor sets of *CCS*'s send and ready receive commands. The following steps determine SUCC(*CCS*):

1. In *CCS*, find the send and ready receive commands.

2. Generate a successor of *CCS*, *CCS'*, by replacing each v_i of *CCS* that is either a send or ready receive command with an element of S(v_i). If the element of S(v_i) chosen is the end node, replace v_i with the inactive marker.

3. Generate SUCC(*CCS*) by repeating step 2 until all unique *CCS'*s are generated from the send and ready receive immediate successor sets. For example, if *CCS* has two sends, v_0 and v_1, and one ready receive, v_3, then the number of successor states of *CCS* is |S(v_0)| * |S(v_1)| * |S(v_3)| = |SUCC(*CCS*)|

A *CCS* containing more than one send and/or ready receive commands signifies these commands happen concurrently. If a *CCS* consists of no send commands and one or more receive commands, where the receive commands are not ready, the CCS has no successors and is an *invalid terminal state* of the distributed system. A CCS comprised of all inactive markers is a *valid terminal state.*

A *proper CCS history* is a sequence of concurrency states $CCS_0, CCS_1, \ldots, CCS_m$ such that

1. $CCS_0 = (r_0, r_1, \ldots, r_{N-1})$,
2. For all $i, 0 \leq i \leq m-1$, $CCS_{i+1} \in \mathrm{SUCC}(CCS_i)$, and
3. CCS_m has no successors (CCS_m is a valid or invalid terminal state).

A *complete CCS history* of a distributed system is a collection of all possible proper CCS histories. The complete CCS history is represented by a directed graph $H = (N, A, r)$ where N is the set of nodes, A is the set of arcs, and $r \in N$ is the root node of the graph. The nodes represent the CCSs, r represents CCS_0, and an arc exists from the node that represents CCS to the node that represents CCS' if $CCS' \in \mathrm{SUCC}(CCS)$. A path from the root node to a node of the graph that has no successors (out-degree is 0) is a proper CCS history. Figure 6.11 is a complete CCS history for the distributed system shown in figure 6.7. The underlined communication events are the sends and ready receive events. The number preceding the communication event is the counter associated with the event.

The following algorithm, `Crt_H()`, generates the graph H to represent the complete concurrency history. The graph H is built breath first, that is, one level of the tree is created before the next level is begun. A node of H consists of two entries, CCS and *FGnode[0...N-1]*. The entry CCS is the CCS this node represents. The array entry

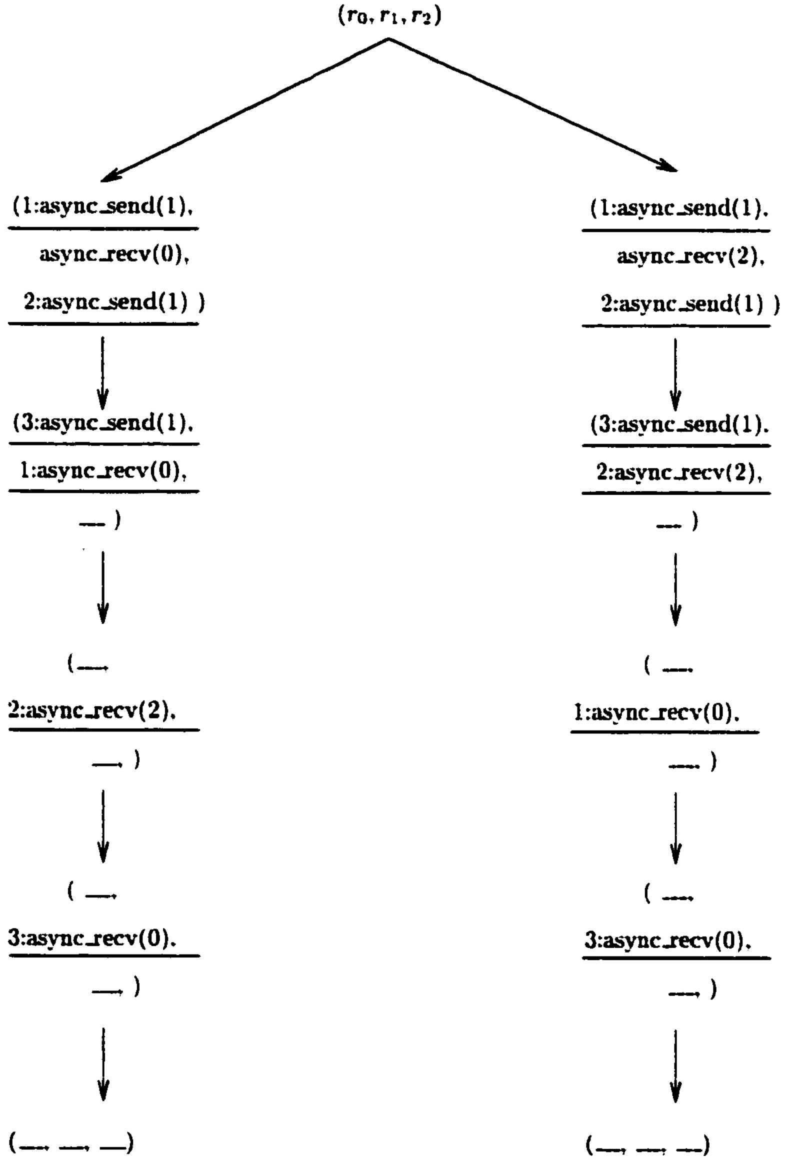

(r0, r1, r2)
(1:async_send(1),
async_recv(0),
2:async_send(1))
(3:async_send(1),
1:async_recv(0),
—)
(—,
2:async_recv(2),
—,)
(—,
3:async_recv(0),
—,)
(—, —, —)
(1:async_send(1),
async_recv(2),
2:async_send(1))
(3:async_send(1),
2:async_recv(2),
—)
(—,
1:async_recv(0),
—,)
(—,
3:async_recv(0),
—,)
(—, —, —)

Figure 6.11: Tree *H* for simple 3 process system

FGnode[i] is the node of FG_i that v_i of CCS represents. The array *FGnode* is set to the appropriate values by algorithm `Crt_H()` and is used later by algorithm `Crt_POG()`.

An array of size N of integers is maintained, *counter*[0], ..., *counter*[$N-1$], by algorithm `Crt_H()` for counting the number of messages that have been sent to each process. The value of *counter*[i] is the number of messages that have been sent to P_i and the number of entries that have been placed in Msg_Q_i. In addition to the Msg_Q_i queues, another queue *CCS_Q* is maintained for recording the CCSs that are to be added next to H. An entry in *CCS_Q* consists of four parts, a CCS, a linked list representing the set SUCC(CCS), the values of the queues $Msg\_Q_0, \ldots, Msg\_Q_{N-1}$ that correspond to CCS after SUCC(CCS) has been determined, and the value of array *counter* that corresponds to CCS after SUCC(CCS) has been determined. The format of an object in the queue is <node, list,$Msg\_Q_0, \ldots, Msg\_Q_{N-1}$, *counter*>. An entry in the linked list *list* consist of two two values, CCS and the variable *FGnode* corresponding to this CCS. The input to `Crt_H()` is $\{FG_0, FG_1, \ldots, FG_{N-1}\}$, and the output is the tree H.

Algorithm `Crt_H()` calls function **`Determine_SUCC()`** to determine the successors of a CCS and to place the appropriate entries in the *Msg_Q* queues and *CCS_Q* queue. Function **`Determine_SUCC()`** calls function **`Generate_SUCC()`** to generate all the successors of a CCS. The variables employed by function **`Generate_SUCC()`** to generate the successors are S_v_i and *index*. Corresponding to each send and receive node of FG_i is an array S_v_i that contains the successors of node v_i, S(v_i), in FG_i. If v_i is an entry in a CCS, array S_v_i is the successor nodes of v_i. The maximum number of successors of a node is *MAXKIDS*, and the dimension of each S_v_i is *MAXKIDS*+1. Each S_v_i array is filled with -1 for unused entries. Variable *index* is an array of N integers. Function `Permute()` determines a successor of CCS by selecting an index into each S_v_i array for each $v_i \in CCS$ that is a send or ready receive. The array *index* contains indexes into each S_v_i. If $v_i \in CCS$ is a send or ready receive, *index[i]* is an index into the array S_v_i. If $v_i \in CCS$ is neither send nor ready receive, *index[i]* is a -1 meaning this v_i should not be changed in the successors of CCS. Function `Generate_SUCC()` calls function `Permute()` to obtain the indexes for a successor of a CCS and continues to call function `Permute()` until all successors of a CCS are generated.

```
Crt_H()
    Initialize queues $Msg\_Q_0, \ldots, Msg\_Q_{N-1}$, $CCS\_Q$ to empty
    Initialize array $counter[0] \ldots counter[N-1]$ to 0
    Create root node $r$
    $r.CCS = CCS_0$
    Determine_SUCC($r, CCS_0, Msg\_Q_0, \ldots, Msg\_Q_{N-1}, CCS\_Q$)
    while $CCS\_Q$ is not empty
        $item$ = behead($CCS\_Q$)    /* format of item is <node,list,$Q_0, \ldots, Q_{N-1}$, counter> */
        $Parent = item.node$
        $LL = item.list$
        $Msg\_Q_0, \ldots, Msg\_Q_{N-1} = item.Q_0, \ldots, item.Q_{N-1}$
        $counter = item.counter$
        for each < $CCS$,$FGnode$> entry in $LL$
            Create a node $n$ in $H$
            $n.CCS = CCS$
            $n.FGnode = FGnode$
            Create edge from $Parent$ to $n$
            Determine_SUCC($n, CCS, Msg\_Q_0, \ldots, Msg\_Q_{N-1}, counter, CCS\_Q$)
        end for
    end while
end algorithm
```

```
Determine_SUCC($n, CCS, Msg\_Q_0, \ldots, Msg\_Q_{N-1}, counter, CCS\_Q$)
    $Msg\_Q'_0, \ldots, Msg\_Q'_{N-1} = Msg\_Q_0, \ldots, Msg\_Q_{N-1}$
    $counter' = counter$
    if ($n$ = root node)
```

```
        SUCC($CCS$) = Generate_SUCC($n$)    /* < CSS.FGnode> is entry in SUCC(CCS) */
        if (SUCC($CCS$) $\neq$ NULL)
            Add <$n$.SUCC($CCS$),Msg_Q'$_0$,... Msg_Q'$_{N-1}$,counter'> to the tail of CCS_Q
        end if
    else
        for $i = 0$ to $N-1$
            if ($v_i$ of $CCS$ = async_recv($j$))
                if (Msg_Q'$_i$ has entry < counter, $j$ >)
                    /* $v_i$ is a ready receive */
                    item = behead first < counter, $j$ > entry in Msg_Q'$_i$
                    append item.counter to $v_i$ in $CCS$        /* item.counter:async_recv($j$) */
                end if
            end if
        end for
        for $i = 0$ to $N-1$
            if ($v_i$ of $CCS$ = async_send($j$))
                counter[j]'++
                Add <counter[j]', $i$ > to Msg_Q'$_j$
                Append counter[j]' to $v_i$ in $CCS$
            end if
        end for
        SUCC($CCS$) = Generate_SUCC($n$)
        if (SUCC($CCS$) $\neq$ NULL)
            Add <$n$.SUCC($CCS$),Msg_Q'$_0$,... Msg_Q'$_{N-1}$,counter'> to the tail of CCS_Q
        end if
    end if
end function
```

```
Generate_SUCC($n$)
    SUCC($CCS$) = NULL
    index[0] ... index[$N-1$] = -1
    for $i = 0$ to $N-1$
        if ($v_i \in CCS$ = send OR $v_i \in CCS$ = ready receive OR $v_i \in CCS = r_i$)
            index[i] = 0
        end if
    endfor
    do
        $CCS' = n.CCS$
        FGnode' = n.FGnode
        for $i = 0$ to $N-1$
            if (index[i] $\neq$ -1)
                $v_i \in CCS'$ = commo command or inactive marker for node S_$v_i$[index[i]]
                FGnode[i] = S_$v_i$[index[i]]
            endif
        endfor
        Add < $CCS'$,FGnode> to the tail of linked list SUCC($CCS$)
    while (Permute(index) = true)
    return(SUCC($CCS$))
end function


Permute(index)
    current = $N$ -1
    while (index[current] == -1) AND (current $\geq$ -1)
        current = current --
    endwhile
    if (current $\geq$ 0)
        index[current]++
    else
        return(false)                                    /* index is all -1's */
    endif
    while (current $\geq$ 0) AND ($S\_v_{current}$[index[current]] = -1)
        index[current] = 0
        current --
        while (current $\geq$ 0) AND (index[current] = -1)
            current --
        end while
        if (current $\geq$ 0)
            index[current]++
        endif
    end while
    if (current < 0)
        return(false)                     /* have been through all permutations */
    else
        return(true)
    endif
end function
```

The following are useful properties of H. In proving these properties, the function ρ maps an event e to the process of the distributed system in which the event occurs.

$$\rho(e) = i \in \Pi \text{ if } e \in P_i$$

Property 6.5 *If*

- *v_i and v_j are events in the execution of a distributed system,*
- $v_i \rightarrow v_j$,
- *$v_i \in CCS$ and v_i is a send or ready receive, and*
- *$v_j \in CCS'$ and v_j is a send or ready receive,*

then CSS is an ancestor of CCS'.

CASE 1 **For** $\rho(v_j) = \rho(v_i)$.

Proof by induction.

BASIS. If

- $v_i \in CCS$.
- v_i is a send or ready receive.
- CCS occurs on level l of H,
- $v_j \in$ S(v_i), and
- $v_j \in CCS'$

then CCS' occurs on level $l + 1$.

Proof. We know that the SUCC(CCS) are children of CCS in H. According to the construction of H, SUCC(CCS) is determined with the S(v_i) for each v_i that is send or ready receive. Node v_j is represented in at least one $CCS' \in$ SUCC(CCS) which occurs on the next level, $l + 1$, of the tree H.

INDUCTIVE HYPOTHESIS. If

- $v_i \in CCS$,
- v_i is a send or ready receive.
- CCS occurs on level l of H.
- $v_i \rightarrow v_k$.
- $\rho(v_i) = \rho(v_k)$, and
- $v_k \in CCS''$.

then CCS'' occurs on level $l + n$ for $n > 2$.

INDUCTIVE STEP. If

- v_k is a send or ready receive.
- $v_j \in \mathrm{S}(v_k)$, and
- $v_j \in CCS'$

then CCS' occurs on level $l + n + 1$.

Proof. We know from the inductive hypothesis that CCS'' occurs on level $l + n$ and that CCS'' is an ancestor of CCS. Since $v_j \in \mathrm{S}(v_k)$, we know from the basis that CCS' occurs on level $l + n + 1$. We can conclude that CCS' is an ancestor of CCS.

CASE 2. **For** $\rho(v_j) \neq \rho(v_i)$.

Proof. Since $v_i \rightarrow v_j$, we know from lemma 5.4 there exists a non-repetitive communication path from P_i to P_j from a send of P_i that happens after v_i (or v_i is this send) and a receive of P_j that happens before v_j (or v_j is this receive). Let $NCP = e_i^0, \ldots, e_j^t$

be this non-repetitive communication path. Two possibilities exist for v_i: either v_i and e_i^0 are the same event, or v_i occurs before e_i^0. Two possibilities exist for v_j, either v_j and e_j^t are the same event or v_j occurs after e_j^t. In the remaining proof we assume without loss of generality that v_i and e_i^0 are the same event, and v_j and e_j^t are the same event.

The events of the path NCP correspond to one or more messages. Consider the following 2 cases:

CASE 2.A. NCP corresponds to one message.

Event v_i is the sending of a message to P_j, and v_j is the corresponding receive of the message from P_i. Let the following be true for the nodes CCS and CCS' of H: $v_i \in CCS$ and $v_j \in CCS'$. According to the construction of H, when v_j is ready, the i entry in Msg_Q_j corresponds to v_i. For the i that corresponds to v_i to be in Msg_Q_j, CCS must be an ancestor of CCS'.

CASE 2.B. NCP defines two or more messages.

Let $NCP = e_i^0, \ldots, e_k^m, e_k^{m+1}, e_l^{m+2}, \ldots, e_j^t$, where $m+2 < t$, and $e_i^0 \in CCS$ and is a send, $e_k^m \in CCS''$ and is a receive, $e_k^{m+1} \in CSS'''$ and is a send, $e_l^{m+2} \in CCS''''$ and is a receive, and $e_j^t \in CCS'$ and is a receive. We know from case 1 that for events e_k^m, e_k^{m+1} of NCP, where $e_k^m \in CCS''$ and $e_k^{m+1} \in CCS'''$, that CCS'' is an ancestor of CCS''' We know from case 2.a that for events e_k^{m+1}, e_l^{m+2} of NCP, where $e_k^{m+1} \in CCS'''$ and $e_l^{m+2} \in CCS''''$, that CCS''' is an ancestor of CCS''''. Therefore, CCS'' is an ancestor of CCS''''. If e_k^m is the receive event immediately following e_i^0 in NCP, then from case 2.a we know that CCS is an ancestor of CCS''. Therefore, CCS is an ancestor of CCS''''. If e_l^{m+2} is the send immediately preceding e_j^t in NCP, then from case 2.a we know CCS'''' is an ancestor of CCS'. We conclude that CCS is an ancestor of CCS'.

■

Property 6.6 *The sends and ready receives of a CCS are concurrent.*

Proof. Assume for $v_i, v_j \in CCS$ that $v_i \rightarrow v_j$. This contradicts property 6.5. ■

Property 6.7 *If CCS is an ancestor of CCS', $v_i \in CCS$ and $v_j \in CCS'$, and v_i and v_j are either sends or ready receives, then $v_i \rightarrow v_j$ if one of the following is true:*

CASE 1. $\rho(v_i) = \rho(v_j)$

CASE 2. *v_i is send to P_j, v_j is a ready receive from P_i, and the next i entry in Msg_Q_j corresponds to v_i.*

CASE 3. *$v_i \rightarrow v_k$ and $v_k \rightarrow v_j$ where $v_k \in CCS''$ such that v_k is either a send of ready receive. CCS is an ancestor of CSS'', and CCS'' is an ancestor of CCS'.*

Proof.

CASE 1.

For v_j to occur in CCS' that is a descendant of CCS, $v_j \in S(v_i)$ or $v_j \in S(S(\ldots S(v_i)\ldots))$ where the nesting of immediate successor sets is two or greater. Therefore $v_i \rightarrow v_j$.

CASE 2.

According to the algorithm for constructing H, for v_j to be a ready receive and the next i entry in Msg_Q_j to correspond to v_i, v_i must happen before v_j.

CASE 3.

This follows directly from the transitive property of the happens before relationship.

■

We know from property 6.6 that the sends and ready receives of a CCS are concurrent. We can deduce concurrent sends and ready receives that occur in different CCSs. Entries v_i and v_j are concurrent if $v_i \in CCS$, $v_j \in CCS'$, v_i and v_j are either sends or ready receives, CCS is an ancestor of CCS', and $v_i \not\to v_j$.

Before stating and proving the next property, lemma 6.1 is established. The execution of a communication event in P_i represented by node n in FG_i is *possible* if there exists at least one path from the root node to n such that the communication events occurring in the path prior to n are either sends or ready receives in H. In other words, the communication event of node n has a possibility of being executed if the communication events that occur prior to it are executed. If a receive is possible, its execution is then dependent on a message being sent, and the receive is labeled as ready when the necessary message is sent. If the necessary message is not sent, the receive does not become ready and does not execute. If a send is possible, it executes since a send's execution is not dependent on the occurrence of a communication event in another process.

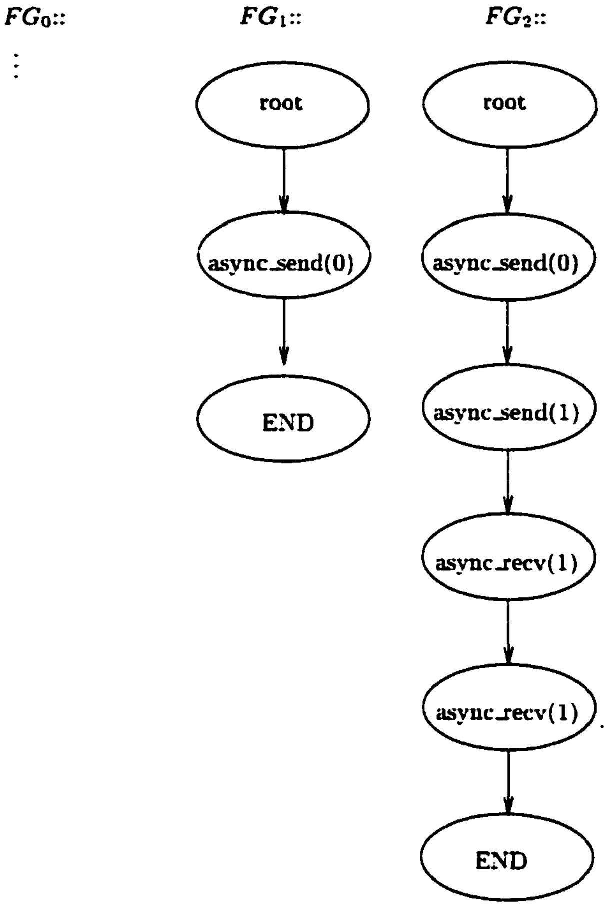

FG_0::
FG_1::
FG_2::
root
async_send(0)
END
root
async_send(0)
async_send(1)
async_recv(1)
async_recv(1)
END

Figure 6.12: Possible and impossible receives

An example of a possible receive event and an impossible receive event is shown in figure 6.12. In FG_2 there exists a path from the root node to the first **async_recv(1)**. We know from the construction of H that **async_send(0)** will be an element of a node of H, and **async_send(1)** will be an element of a node of H. The first **async_recv(1)** of FG_2 will occur in a node of H as a receive, but this receive will not be ready since the sending of a message from P_1 to P_2 does not exist. This receive occurs as an entry in an H node to represent the receive waiting to execute. The communication commands of P_2 prior to the first **async_recv(1)** are executed, and **async_recv(1)** is possible although it will not execute. Since the first **async_recv(1)** of P_2 can not execute, the second **async_recv(1)** of P_2 will not occur in a node of H and is therefore impossible.

Lemma 6.1 *If node n of FG_i is a communication node and the execution of n is possible, then n is a send or receive in at least one node of H.*

Proof.

BASIS.

> If node n is a successor of the root node of FG_i, $n \in \mathrm{S}(r_i)$, then $v_i = n$ for at least one $CSS \in \mathrm{SUCC}(CCS_0)$. CCS_0 occurs on level 0 of H, therefore each $CCS \in \mathrm{SUCC}(CCS_0)$ occurs on level 1 of H.
>
> Proof. According to the construction of H, the $\mathrm{SUCC}(CCS_0)$ is determined by $\mathrm{S}(r_i)$ for all i. Node n of FG_i is represented in at least one $CCS \in \mathrm{SUCC}(CCS_0)$.

INDUCTIVE HYPOTHESIS.

> If node n' is a communication node of FG_i, n' is an immediate predecessor of node n in FG_i, and the execution of n' is possible, then node n' is represented in CCS' on level i of H.

INDUCTIVE STEP.

> If node $n \in \mathrm{S}(n')$ and the execution of node n is possible, then node n is represented in at least one $CCS \in \mathrm{SUCC}(CCS')$ on level $i + 1$ of H.
>
> Proof. From the inductive hypothesis, we know n' is represented in node CCS' on level i of H. For the execution of node n to be possible, node n' is either a send or ready receive element of CCS'

In the construction algorithm, SUCC(CCS') is determined by S(v_i) for all i of CCS' that are sends or ready receives. Let $v_i' = n'$ in node CCS'. Since v_i' is a send or a ready receive of CCS' and $n \in$ S(v_i'), we can conclude that node n is represented in at least one $CCS \in$ SUCC(CCS') on level $i + 1$ of H.

■

Property 6.8 *The tree H derived from $\{ FG_0, \ldots, FG_{N-1} \}$ represents all partial orders of the distributed system represented by $\{ FG_0, \ldots, FG_{N-1} \}$.*

Proof.

1. From properties 6.1, 6.3, and 6.4, we know each FG_i represents all execution paths of P_i, and the occurrence of the nodes of a path of FG_i represents the total order of events of P_i.
2. From properties 6.5 and 6.7, we know all the happens before relationship among local and non-local events of the distributed system are correctly represented in H.
3. From lemma 6.1, we know that if the execution of a communication node of FG_i is possible, then the communication event is represented in H.

From (1), (2) and (3), we can conclude all possible executable events of each process are represented in H, and all happens before relationships among these events are correctly represented in H. Therefore all partial orders of the distributed system are represented in H. ■

In some cases, two or more branches of H represent the same partial order. Consider the portion of tree H for a four processor system in figure 6.13. In this example, the receives of the CCSs are not ready. The sends of each CCS are replaced in the child CCSs with an inactive marker. Both leaf node branches indicate that P_0 does not complete execution. The two branches shown represent the same partial order. From the tree H, a partial order graph, POG, is constructed that combines branches that represent the same partial order into one branch. Also, only the sends and receives that are executed in a partial order are represented in the POG. In other words, the sends and ready receives are presented in the POG.

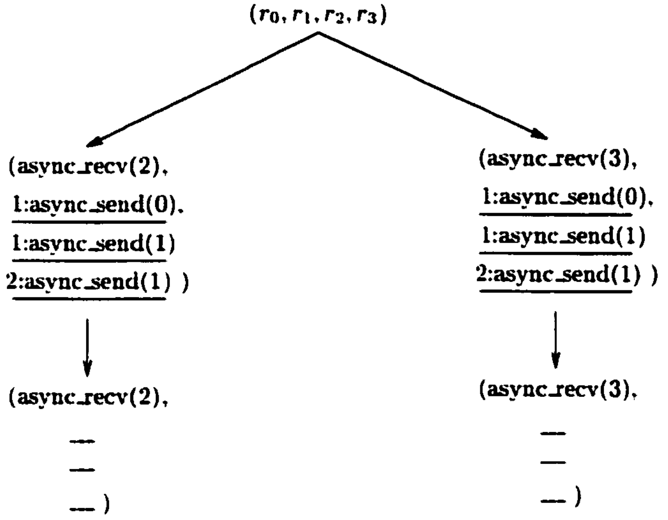

(r_0, r_1, r_2, r_3)
(async_recv(2),
1:async_send(0),
1:async_send(1)
2:async_send(1))
(async_recv(3),
1:async_send(0),
1:async_send(1)
2:async_send(1))
(async_recv(2),
—
—
—)
(async_recv(3),
—
—
—)

Figure 6.13: Same partial orders

A POG is a directed graph (N, A, s) where N is the set of nodes, A is the set of arcs, and $s \in N$ is the root node of POG. The nodes of the POG are generated from H's nodes such that the POG nodes represent the sends and ready receives command of the H nodes.

In the remaining discussion of *POG* nodes, the following format of an entry is a *POG* node is adapted for conciseness. A send entry has the format $c : iSj$ where c is the counter, i is the process executing the send and j is the destination process. A ready receive entry has the format $c : iRj$ where c is the counter, i is the process executing the receive and j is the sender. The *POG* is constructed by traversing H breath first, starting at the the root node of H, and generating the nodes of the *POG* in breath first order. The algorithm for constructing the *POG* determines whether CCSs have *equivalent* send and ready receive communication entries. $CCS_1, \ldots, CCS_t$ have equivalent communications if the following conditions are true:

1. If at least one $CCS_{q:1 \leq q \leq t}$ contains one or more send and/or ready receive commands.
2. If v_i is a send command of $CCS_{q:1 \leq q \leq t}$, then each v_i in all $CCS_{r:1 \leq r \leq t}$ is the same[1] send command.
3. If v_i is a ready receive command of $CCS_{q:1 \leq q \leq t}$, then each v_i in all $CCS_{r:1 \leq r \leq t}$ is the same receive command and is a ready a receive.

If CCS_i and CCS_j have equivalent communication commands, the equivalent communication commands of CCS_i and CCS_j are all the send and ready receive commands that occur in CCS_i and CCS_j.

The algorithm for constructing the *POG* relies on the function **EQUIV()**. The input to **EQUIV()** is a set of H nodes, *node_set*, and the return value is a subset of *node_set*. If *node_set* contains two or more nodes that have equivalent communication commands, **EQUIV()** returns these nodes, else **EQUIV()** returns $\emptyset$. Nodes of H that have equivalent communication commands are called equivalent nodes. If **EQUIV()** finds a subset of *node_set* that have equivalent communication commands, **EQUIV()** removes these nodes from *node_set*.

---

[1] Same meaning each v_i represents the same node of FG_i and the counters are equal.

If *node_set* contains two or more equivalent node subsets, **EQUIV()** nondeterministicly returns only one of these subsets. For example, let the CCSs of *node_set* equal $\{(2{:}0S1, 3{:}1R2, 2R0, 3R0), (2{:}0S1, 3{:}1R2, 2R1, 3R1), (1{:}0R1, 1R3, 2{:}2S0, 3R0), (1{:}0R1, 1R0, 2{:}2S0, 3R0)\}$. The first and second entries in *node_set* are equivalent and the third and forth entries in *node_set* are equivalent. **EQUIV()** will return either the nodes corresponding to $\{(2{:}0S1, 3{:}1R2, 2R0, 3R0), (2{:}0S1, 3{:}1R2, 2R1, 3R1)\}$ or $\{(1{:}0R1, 1R3, 2{:}2S0, 3R0), (1{:}0R1, 1R0, 2{:}2S0, 3R0)\}$. To select a node from a set of H nodes for testing if a subset of the nodes are equivalent, function **EQUIV()** calls function **Select()**. Function **Select()** randomly picks a node element from a set of nodes, removes the element from the set, and then returns this element.

If the return value of **EQUIV()** is not NULL, the returned nodes are represented with one node in the *POG*. This *POG* node is labeled with the sends and ready receives of the returned nodes.

The *POG* construction algorithm, **Crt_POG()**, places information about the newly added nodes of the *POG* in the queue data structure *VisitNodes*. An entry in the *VisitNodes* queue has the format <*node_ptr*, *node_set*>. The entry *node_ptr* points to a node of the *POG*, and *node_set* is a set of one or more H nodes. The set *node_SuccSet* is a set of H nodes that is built from the successors of equivalent nodes. The string *Commos* is set to the sends and ready receives of an H node and is for labeling the nodes of the *POG*. For example if the

CCS of a node is (1:0*S*1, 3:1*S*0), then *Commos* = "1:0*S*1, 3:1*S*0". The following algorithm constructs the *POG* from *H*:

```
Crt_POG()
    Initialize queue VisitNodes to empty
    Create root node S (labeled root)
    Add <S.SUCC(root node of H)> as the first entry in the queue VisitNodes
    while ( VisitNodes not empty )
        item = behead( VisitNodes)            /* format of item is <node_ptr, state_set> */
        POG_ptr = item.node_ptr
        node_set = item.node_set              /* state_set= {$CCS_1, \ldots, CCS_m$}, $m \geq 1$ */
        while ((EQUIV_set = EQUIV(node_set) $\neq \emptyset$)
            Commos = the sends and ready receives of the CCSs of EQUIV_set
            Create POG node N and label with Commos
            Create an arc from node of POG_ptr to N
            node_SuccSet = $\emptyset$
            for each node of EQUIV_set
                node_SuccSet = node_SuccSet $\cup$ SUCC(node)
            end for
            Add the entry <N, node_SuccSet> to the tail of VisitNodes
            node_set = node_set - EQUIV_set
        end while
        for each node $\in$ node_set
            if ((Commos = sends and ready receives of the CCS of node) $\neq$ NULL)
                Create POG node N and label with Commos
                Create an arc from node of POG_ptr to N
                Add the entry <N.SUCC(node)> to the tail of VisitNodes
            else
                Create POG node N and label as END node
                Create an arc from node of POG_ptr to N
            endif
        endfor
    end while
end algorithm
```

```
EQUIV(node_set)
    node_set' = node_set
    EQUIV_found = false
    while (node_set' $\neq \emptyset$) AND (EQUIV_found = false)
        Node_1 = Select(node_set')
        EQUIV_set = {Node_1}
        Commos = the sends and ready receives of Node_1.CCS
        FGnode = Node_1.FGnode
        local_set = node_set'
        while (local_set $\neq \emptyset$)
            Node_2 = Select(local_set)
            Commos_2 = the sends and ready receives of Node_2.CCS
            FGnode2 = Node_2.FGnode
            if (Commos = Commos_2) AND (FGnode = FGnode2)
                EQUIV_found = true
                Add Node_2 to EQUIV_set
            end if
        end while
    end while
    if (EQUIV_found=true)
        return(EQUIV_set)
    else
        return($\emptyset$)
end function
```

The *POG* represents the causal and concurrent relationship among the communication events. The first four properties of the *POG* are derived directly from the properties of H.

Property 6.9 *If $e_i \rightarrow e_j$, where e_i and e_j are communication events, and e_i is an entry in node N of the POG and e_j is an entry in node N' of the POG, then N is an ancestor of N'.*

Property 6.10 *The communication events represented in a node of the POG are concurrent.*

Property 6.11 *If POG node N is an ancestor of POG node N' and $e_i \in N$ and $e_j \in N'$, then $e_i \rightarrow e_j$ if one of the following is true:*

1. $\rho(i) = \rho(j)$

2. *e_i is a send to P_j, e_j is a receive from P_i, and e_j is the corresponding receive for this send.*

3. *$e_i \rightarrow e_k$ and $e_k \rightarrow e_j$ where $e_k \in N''$ such that N is an ancestor of N'' and N'' is an ancestor of N'.*

Property 6.12 *The POG represents all partial orders.*

The construction of the *POG* prunes the tree H with the `EQUIV()` function so that one branch of the *POG* from root to leaf node represents an unique partial order $\alpha \in \mathcal{P}$. The nodes of the *POG* are minimized from the nodes of H to represent only the communication commands that occur in an execution of the distributed system. The properties of H remain true in the *POG* since the construction does not eliminate or create new information about the occurrence of the communication events.

Lemma 6.2 *The construction of the POG from H preserves the causal and concurrent relationships represented in H.*

Proof.

CASE 1. Nodes of H with equivalent communication commands do not exist.

Function `EQUIV()` always returns $\emptyset$ for nodes of tree H; i.e., there exists no nodes of H that have equivalent communication commands.

Algorithm `Crt_POG()` traverses H in a breath-first order with the use of queue *VisitNodes*. The next entry in *VisitNodes* represents the next group of nodes in H to be represented in the *POG*. Consider creating the nodes and edges of the *POG*.

NODES.

If a node of H, h, has at least one send or ready receive, the node is represented in the *POG* by creating a *POG* node and labeling it with the corresponding sends and ready receives of $h.CCS$.

If a node of H does not have at least one send or ready receive, a node is not created in the *POG* to represent this node. A node of H, h, that does not have at least one send or ready receive means no communication commands are executed after the sends and ready receives of h's parent, and therefore node h does represent any causal or concurrent relationships among events.

EDGES.

If a node of H, h, is represented in the *POG* by node n and if a child of h is represented in the *POG* with node n', then an edge is created from node n to node n' of the *POG*. Therefore, causal and concurrent relationships among nodes of H are preserved in the *POG*. Since all nodes of H that have at least one send or ready to receive are represented in the *POG*, all causal and concurrent relationships are preserved.

CASE 2. Nodes of H with equivalent communication commands do exist.

Function `EQUIV()` finds nodes of H that have equivalent communication commands.

The nodes that are input to `EQUIV()` are nodes that occur in the same level of H. If the nodes of H, $\{ h_1 \ldots h_t \}$, are equivalent (the *CCS*s have equivalent communication commands) one node n is created in the *POG* to represent these t nodes and is labeled with the equivalent communication commands. Then set *node_SuccSet* is built so that *node_SuccSet* = SUCC(h_1) $\cup \cdots \cup$ SUCC(h_t). Set *node_SuccSet* is placed in the queue *VisitNodes* for generating the children of node n. Therefore nodes of H that represent the same causal and concurrent relationships are represented as one node in the *POG*. and all causal and concurrent relationships that are represented by the successor nodes of $\{ h_1 \ldots h_t \}$ will be represented in the *POG* as children of n.

■

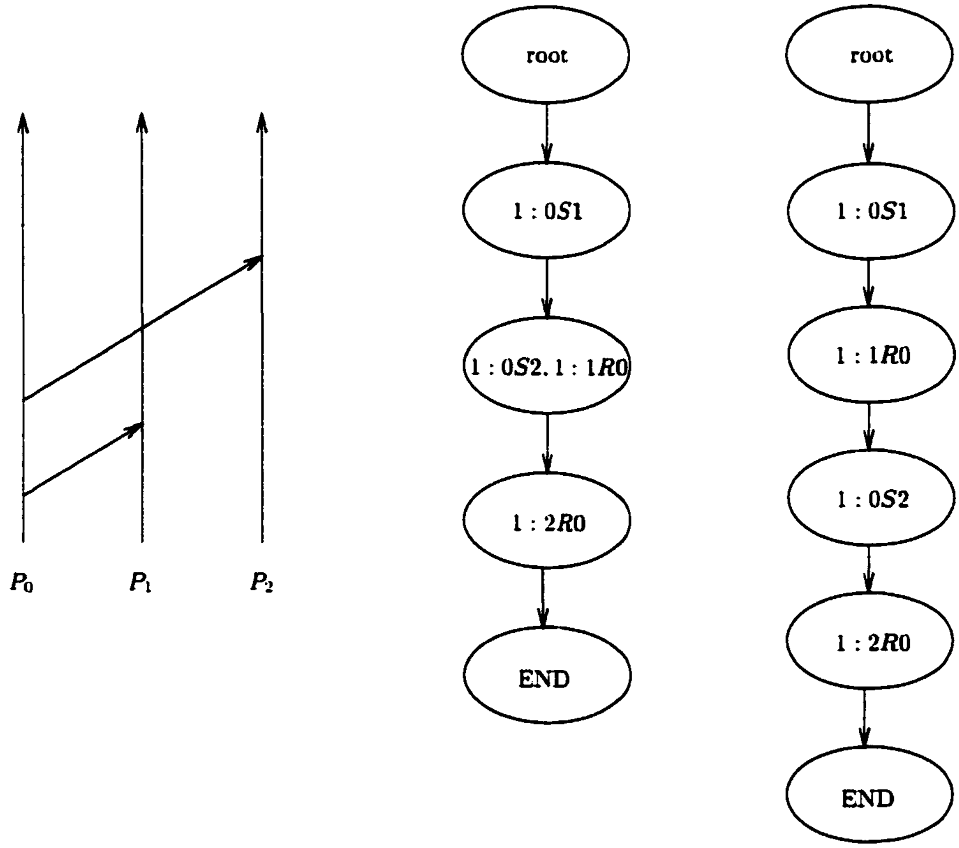


Figure 6.14: 2 possible *POG*s

A partial order $\alpha \in \mathcal{P}$ is represented in the *POG* by a path beginning at the root node and ending at a leaf node of the tree. The process of generating the *POG* guarantees that there exists only one possible representation of a partial order in the *POG*. Figure 6.14 is the time-space diagram of a distributed system's execution and the two possible unique path representations of the partial order defined by the system's execution. From property 6.11 we can determine from either of the two paths the following relationships: 1:0S1 → 1:0S2, 1:0S1 → 1:1R0, 1:0S2 → 1:2R0, and 1:0S1 and 1:0S2 are concurrent. Of the two paths shown in figure 6.14, only the path to the left is generated by the `Crt_POG()` algorithm. Since the *POG* is derived from H, algorithm `Crt_H()` dictates the path that will occur in the *POG* for a partial order. The H generated by algorithm `Crt_H()` is shown in figure 6.15 for the execution shown in figure 6.14. The left path in 6.14 is generated from this H.

Lemma 6.3 *For partial order $\alpha \in \mathcal{P}$, there exists one possible representation of α in the POG.*

Proof.

A partial order is represented in the *POG* by a path beginning at the root node and ending at a leaf node of the tree.

Assume there exist two different representations of α in the *POG*, thus there must exist two differing paths from the root node to a leaf node that correspond to α. For this to occur, H must have at least one path from the root to a leaf node that corresponds to each path of α in the *POG* (according to algorithm `Crt_POG()` and lemma 6.2). Let p be one such path of H, and let p' be the other path of H. The nodes of paths p and p' must differ in the order that the sends and ready receives occur in the path to generate two different

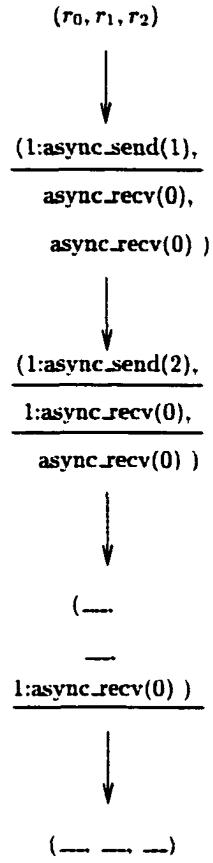

(r0, r1, r2)
(1:async_send(1),
async_recv(0),
async_recv(0))
(1:async_send(2),
1:async_recv(0),
async_recv(0))
(—
—
1:async_recv(0))
(— — —)

Figure 6.15: *H* tree

representations of α in the *POG* (according to algorithm `Crt_POG()` and function `EQUIV()`).

For p and p' to differ in this manner, there must exist a node n of H that is common to both paths that has at least two children that mark the differing of paths p and p'. Let c be a child of n that corresponds to path p and let c' be a child of n that corresponds to path p'. For nodes c and c' to correspond to different paths in the *POG*, nodes c and c' must consist of different send and ready receives (according to algorithm `Crt_POG()` and function `EQUIV()`).

For node n to have children, node n must have at least one send or ready receive. Let $v_i \in n$ where v_i is a send, iSj. The children of n, SUCC(n), are determined by the successors of iSj, S(iSj). For SUCC(n) $= \{c, c'\}$, S(iSj) must have two entries. For S(iSj) to have two entries, there must exist two branches in FG_i from the node of FG_i that corresponds to iSj such that each branch includes a successor of iSj. In FG_i, a branch indicates a different total order of events of P_i. Therefore c and c' of H mark the beginning of two different partial orders, and the *POG* paths that are derived from p and p' represent two different partial orders. A contradiction to our assumption has been reached.

Let $v_i \in n$ where v_i is a ready receive, iRj. Since SUCC(n) $= \{c, c'\}$ occurs under the same conditions as when $v_i = iSj$, the same contradiction is reached for $v_i = iRj$. ■

Property 6.13 *Each path of the POG from root node to leaf node represents a unique partial order*

Proof.

Assume two paths of the *POG* represent the same partial order. Two cases are possible for this to occur.

Case 1 . The two paths are identical.

For this to occur, there must exist a node of the *POG* that has two children that are identical. This contradicts function `EQUIV()`.

Case 2 . The two paths differ but represent the same partial order.

This contradicts lemma 6.3.

■

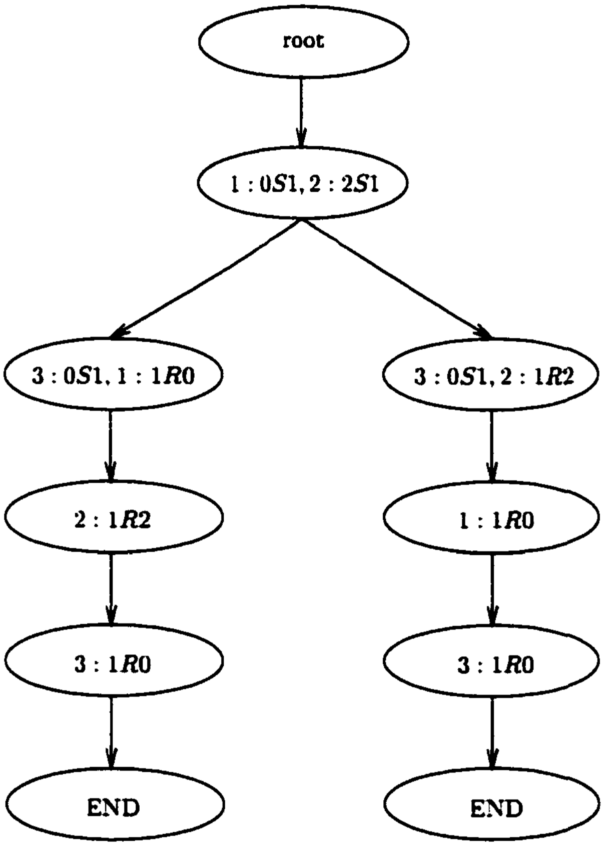


Figure 6.16: POG derived from H of figure 6.11

Figure 6.16 is the *POG* of the distributed program in figure 4.2. and this *POG* is generated from H shown in figure 6.11. Notice that the two partial orders of figure 4.3 are each represented as a path from root to a leaf node in the *POG*. In particular. the left path of the *POG* represents $P0_1$. and the right path of the *POG* represents PO_2.

6.4 *LCP* and *LCP'* Events

For an event e_i, each process's *LCP* and *LCP'* events can be determined from the *POG*. From theorem 4.1 and lemma 5.1 , we know that for a partial order α and event e_i, at most one $CC(e_i)$ exists and this $CC(e_i)$ consists of LCP events which are communication events. The causal global state for event e_i is identified by $CC(e_i)$. From theorem 5.2. we know that by piggybacking state data on the LCP and LCP' events. the $CGState(e_i)$ is available in P_i for event e_i.

Before determining the LCP and LCP' events of the assert statement e_i. the *last* LCP' receive event that occurs in P_i must be identified for each execution path of P_i that includes e_i. "Last" means the receive event corresponding to the last of the latest causal messages that will piggyback state information to P_i for evaluating the assert statement. Since the assert statement and all possible executions of P_i are represented in FG_i. the last LCP' event(s) of P_i is(are) identified from FG_i.

The algorithm **Bound_Assert()** determines the last LCP' event(s) of an event. Referring to figure 6.7. note that an async_recv(0) of P_1 has two parents. Since a node of FG_i can have more than one parent. the parents of each node are maintained as a linked list of node pointers. The variable *current_list* is set to this linked list. The variable *NextBranch* is a stack. and an entry in the stack is a linked list of FG_i node pointers. The variable *Local_LCPs* is a linked list of FG_i node pointers. and at the completion of the algorithm the entries in this linked list are the last LCP' receive events of an event in P_i.

The input to **Bound_Assert()** is FG_i and *assert_node*. The variable *assert_node* is a pointer to the assert node in FG_i. Algorithm **Bound_Assert()** begins the search for the last LCP' events of *assert_node* with the first parent node in *assert_node*'s *current_list*. The search continues by traveling up the tree until a receive event is found or the root node is reached. Each possible path from *assert_node* to the root node of FG_i is searched for a receive event. In the case that multiple paths exist from the *assert_node* to the root node, a different receive.event may be found on each path. If a receive event is found on the path, this receive event is a last LCP' event and is placed in the linked list *Local_LCPs*, and the search is stopped on this path. The output of the algorithm is *Local_LCPs*.

```
Bound_Assert()
                                                        /* input: $FG_i$ and assert_node */
    current_list = the parent nodes of assert_node
    NextBranch = NULL
    Local_LCPs = NULL
    crnt_node = first entry in current_list
    Remove crnt_node from current_list
    receive_found = false
    do
        while (receive_found=false) AND (crnt_node $\neq$ root node of $FG_i$)
            if (current_list $\neq$ NULL)
                Push current_list on the stack NextBranch
            endif
            if crnt_node = receive
                Add crnt_node to Local_LCPS
                receive_found = true
            else
                current_list = parent nodes of crnt_node
                crnt_node = first entry in current_list
                Remove crnt_node from current_list
            endif
        endwhile
        if (NextBranch $\neq$ NULL)
            receive_found = false
            current_list = Pop(NextBranch)
            crnt_node = first entry in current_list
            Remove crnt_node from current_list
        endif
    while (NextBranch $\neq$ NULL)
end algorithm
```

From **Bound_Assert()** we have identified the last LCP' events in FG_i. The next step is to identify these same events in the *POG*. Each entry in *Local_LCPs* is represented in the *POG* at least once if the execution of the receive is possible. To access the *POG* node that corresponds to an entry in *Local_LCPs*, it is necessary to know which send and receive commands of the control flow graphs each *POG* node represents. When creating a *POG* node, a linked list of pointers is built that identifies the send and/or receive nodes of the control flow graph that the *POG* node represents. Also, each send or receive node of FG_i has a linked list of pointers to the *POG* nodes that represent this communication event. For each entry in *Local_LCPs*, which is actually a pointer to the appropriate receive node in FG_i, the *POG* node(s) that represent the receive can be accessed.

If an entry in *Local_LCPs* is represented by a *POG* node, then this receive is a LCP' event of *assert_node* in P_i. If an entry in *Local_LCPs* is not represented by a POG node, then this receive can not be executed and therefore is not an LCP' event.

Continuing with the distributed program shown in figure 4.2, we find the last LCP' events of P_1 from figure 6.7 using algorithm **Bound_Assert()**. Process P_1 has only one such message, **async_recv(0)**. This is the **async_recv(0)** that immediately precedes the assert statement in FG_1. Two nodes of the *POG* represent this communication command, one for partial order PO_1 and the other for partial order PO_2. These two *POG* nodes are shown in figure 6.17 with double circles.

From theorem 5.2 we know for assert event e_i there exist a non-repetitive communication path from each LCP event to an LCP' event of P_i that consists of LCP and LCP' events. The algorithm **Find_LCPs()** accesses the *POG* to find these LCP' and LCP events for the assert event e_i. For each partial order branch of the *POG* corresponding to an entry in *Local_LCPs*, the algorithm traverses the branch in an upward direction beginning with

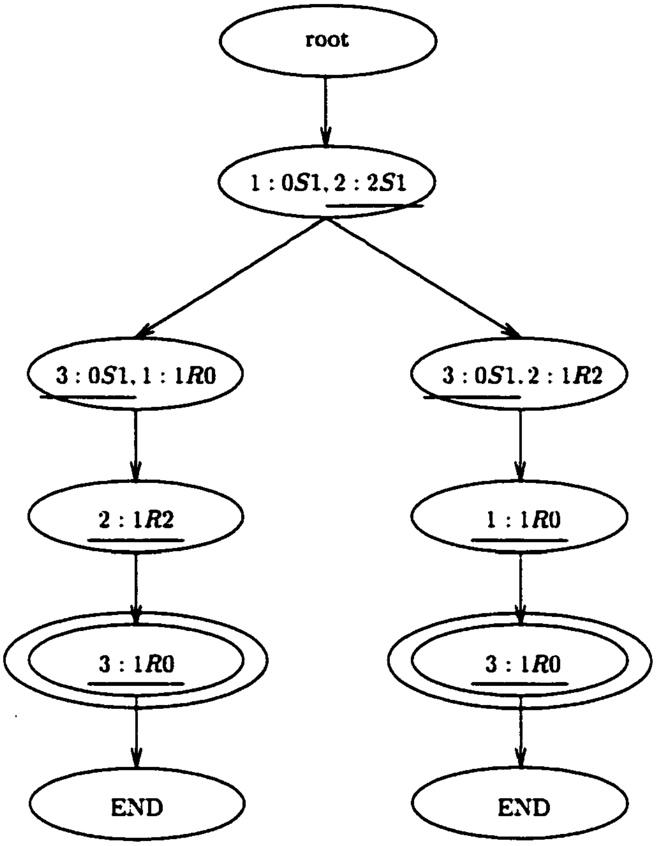

root
1 : 0S1, 2 : 2S1
3 : 0S1, 1 : 1R0
3 : 0S1, 2 : 1R2
2 : 1R2
1 : 1R0
3 : 1R0
3 : 1R0
END
END

Figure 6.17: *LCP* and *LCP'* events

the receive event of *Local_LCPs* up to possibly the root node to find these non-repetitive communication paths. Since the branch is traversed upward, the receives (LCP's) of the messages are encountered before the matching sends (LCPs).

When a receive event, $c{:}jRk$, is encountered in a *POG* node, it is a *candidate LCP'* event if:

1. a non-repetitive communication path has been found from P_j to P_i that occurs after $c{:}jRk$ and a non-repetitive communication path from P_k to P_i has not been found, or

2. the receive event is an event of P_i and a non-repetitive communication path from P_k to P_i has not been found.

The reason for *candidate* is the receive event *c:jRk* is not an LCP' event of P_j if another non-repetitive communication path from P_k to P_i is found before[2] the matching send of *c:jRk* is encountered in the *POG*.

When a send event, *c:jSk*, is encountered in the *POG*, it is an *LCP* event if:

1. the matching receive, *c:kRj*, has been encountered, and
2. receive event, *c:kRj*, is a candidate LCP' event.

Six data structures are employed by algorithm `Find_LCPs()` to find the *LCP* and LCP' events when traveling up a branch of the *POG*. Three of the six data structures are sets of process numbers. These sets are *FoundProcs*, *Sends*, and *Rec_wo_Sends*. The set *FoundProcs* contains the entry j if the piggybacking message for P_j, consisting of the send event of P_j and the matching receive event, has been determined from the *POG*. Set *Sends* contains the entry j if the send event for piggybacking data from P_j has been found. Set *Rec_wo_Sends* contains the entry j if the receive end of a piggybacking message has been found for P_j but the matching send has not. The other three data structures are queues: *RwoSQ*, *SendQ* and *RecvQ*. The queue *RwoSQ* contains entries for receive commands whose matching send command has not been found in the *POG*. An entry in *RwoSQ* has the format <c, i, j, *POGnode*> where c is the counter, i is process number of the receiver,

---

[2] Before in this context meaning the path happens after the matching send since the *POG* is traversed upward

j is the process number of the sender, and *POGnode* is a pointer to the *POG* node that contains the receive. The queue *SendQ* contains an entry for each *LCP* send event, and an entry has the format <c, j, *POGnode*> where c is the counter, j is the process number of the sender and *POGnode* is a pointer to the *POG* node containing the send event. The queue *RecvQ* contains an entry for each LCP' receive event, and an entry has the format <c, j, *POGnode*> where c is the counter, j is the process number of the receiver and *POGnode* is a pointer to the *POG* node containing the receive event.

```
Find_LCPs()                                /* Input: Local_LCPs Output: SendQ, RecvQ */

  for each entry in Local_LCPs where the event format is $c{:}iRj$
    for each POG node that contains $c{:}iRj$
      POGnode = POG node that contains $c{:}iRj$
      FoundProcs = Sends = $\emptyset$
      Rec_wo_Sends = $\{i\}$
      RwoSQ = NULL
      Insert <c, i, j, POGnode> in RwoSQ
      POGnode = ParentOf(POGnode)
      while (POGnode $\neq$ root node) AND (FoundProcs $\neq$ ({ 0, ... ,N-1 } - $i$))
        for each receive, $c{:}jRk$, in POGnode
          if (($j \in$ Sends) OR ( $j = i$)) AND ($k \notin$ FoundProcs)
            AND (Rec_wo_Sends does not have entry $jRk$)
            Insert <c, j, k, POGnode> in RwoSQ
            Rec_wo_Sends = Rec_wo_Sends + $j$
          endif
        endfor
        for each send, $c{:}jSk$, in POGnode
          if ($k \in$ Rec_wo_Sends)AND((Recv_POGnode = SearchQ(c, k, j)) $\neq$ NULL)
            if (RwoSQ does not have an entry with $k$ as the receiver)
              Rec_wo_Sends = Rec_wo_Sends - $k$
            endif
            Sends = Sends + $j$
            FoundProcs = FoundProcs + $j$
            Insert <c, j, POGnode> in SendQ
            Insert <c, k, Recv_POGnode> in RecvQ
          endif
        endfor
```

```
            POGnode = ParentOf(POGnode)
        endwhile
    endfor
endfor
end algorithm
```

The if statements of the algorithm are complex and require explanation. When a receive event *c:jRk* occurs in a *POG* node, the following check is made:

$$\text{if } ((j \in \text{Sends}) \text{ or } (j = i)) \text{ and } (k \notin \text{FoundProcs}) \text{ and } (Rec\_wo\_Sends \text{ does not have entry } jRk)$$

The value j being in the set *Send* indicates a non-repetitive communication path has been found from P_j to P_i that occurs after this receive. Any data received by P_j from receive event *c:jRk* can then be piggybacked on the messages of the path to P_i. If $j = i$, then the receive is a local event of the process evaluating the assert. The data piggybacked on the message of this receive event will be available to the assert statement without having to piggyback the data on additional messages. The value k being in *FoundProcs* indicates the *LCP* and *LCP′* events for piggybacking the state information of P_k have been found, and the message associated with this receive is not needed for piggybacking data from P_k to P_i. If the if statement evaluates to true, the receive event is a candidate *LCP′* event.

Assume P_j has two or more *jRk* receive events, and one *jRk* is already inserted in *RwoSQ*. If the other *jRk* receive events are encountered by the algorithm, they should not be considered as *LCP′* events since there execution occurs before the *jRk* that is represented in *RwoSQ*. The last condition of the if statement prevents these events from being considered.

When a send event $c{:}jSk$ occurs in a *POG* node, the following check is made first:

$$\text{if } (k \in \textit{Rec\_wo\_Sends}) \text{ and } ((\textit{Recv\_POGnode} = \texttt{SearchQ}(c,k,j)) \neq \text{NULL})$$

The value k being in the *Rec_wo_Sends* set indicates P_k has a receive event that is a candidate LCP' event and the matching send event has not be found. For this send to be the matching send event, the receive for P_k must be expecting a message from P_j. The function `SearchQ()` searches the queue *RwoSQ* for the occurrence of the entry $<c, k, j,$ *POGnode*$>$. If found, the entry is deleted from *RwoSQ* and *POGnode* is returned. If not found, NULL is returned. The `if` statement evaluating to true indicates this send, $c{:}jSk$, is an *LCP* event and the matching receive pointed to by *Recv_POGnode* is an LCP' event. The nested `if` statement checks whether k should be removed from *Rec_wo_Sends*.

$$\text{if } (\textit{RwoSQ} \text{ does not have an entry with } k \text{ as the receiver})$$

If, after `SearchQ()` removes the entry corresponding to $c{:}jSk$, *RwoSQ* has an entry where P_k is the receiver of a message, then there is a possibility that P_k has additional LCP' events. The value k should remain in *Rec_wo_Sends* to indicate that receives of P_k are candidate LCP' messages. If *RwoSQ* does not have an entry where P_k is the receiver of a message, then the value k is removed from *Rec_wo_Sends*.

Since we have identified the last LCP' events of the distributed program shown in figure 4.2, we next identify the *LCP* and LCP' events. For each partial order, the *LCP* and LCP' events are determined with algorithm `Find_LCPs()`. The steps taken by `Find_LCPs()` to find the *LCP* and LCP' events of partial order PO_1 are given. For each iteration of the algorithm's loop, the variables values are shown. The values of the variables before executing the loop are:

$Rec\_wo\_Sends = \{1\}$
$FoundProcs = \emptyset$
$Sends = \emptyset$
$i = 1$
$RwoSQ$ = (<3,1,0,3:1R0>)
$SendQ$ = NULL
$RecvQ$ = NULL
$POGnode$ = 2:1R2

For the first iteration of the loop, the if statement (($j \in Sends$) OR ($j = i$)) AND ($k \notin$ FoundProcs) AND (Rec_wo_Sends does not have entry jRk) evaluate to true for event 2:1R2. The values of the variables after this iteration are:

$Rec\_wo\_Sends = \{1\}$
$FoundProcs = \emptyset$
$Sends = \emptyset$
$i = 1$
$RwoSQ$ = (<3,1,0,3:1R0 >, <2,1,2,2:1R2 >)
$SendQ$ = NULL
$RecvQ$ = NULL
$POGnode$ = 3:0S1,1:1R0

For the second iteration of the loop, the if statement ($k \in$ Rec_wo_Sends AND ($Recv\_POGnode$ = **SearchQ**(c, k, j)) $\neq$ NULL) evaluates to true for event 3:0S1. The values of the variables after this iteration are:

Rec_wo_Sends = $\{1\}$
FoundProcs = $\{0\}$
Sends = $\{0\}$
$i = 1$
RwoSQ = (<2,1,2,2:1*R*2>)
SendQ = (<3,0,3:0*S*1>)
RecvQ = (<3,1,3:1*R*0>)
POGnode = 1:0*S*1, 2:2*S*1

For the third iteration of the loop, the if statement ($k \in$ Rec_wo_Sends AND (*Recv_POGnode* = `SearchQ`(c, k, j)) $\neq$ NULL) evaluates to true for event 2:2*S*1. The values of the variables after this this iteration are:

Rec_wo_Sends = $\emptyset$
FoundProcs = $\{0,2\}$
Sends = $\{0,2\}$
$i = 1$
RwoSQ = NULL
SendQ = (<3,0,3:0*S*1>, <2,2,2:2*S*1>)
RecvQ = (<3,1,3:1*R*0>, <2,2,2:1*R*2>)
POGnode = root

The condition of the while loop evaluates to false, and the *LCP* and *LCP'* events for PO_1 are identified in *SendQ* and *RecvQ*. The *LCP* events are 3:0*S*1 and 2:2*S*1, and the *LCP'* events are 3:1*R*0 and 2:1*R*2. For PO_2, algorithm `Find_LCPs()` identifies the *LCP* events 3:0*S*1 and 2:2*S*1, and the *LCP'* events 3:1*R*0 and 2:1*R*2. These events are underlined in figure 6.17. In this particular example, the *LCP* and *LCP'* events are the same for both partial orders, but this is not always the case. Notice that the send and receive of the first message from P_0 to P_1 are not identified as LCP and LCP' events. This message need not be used for piggybacking data.

The properties resulting from this algorithm are:

1. Event e_j is an LCP event if and only if event e_j is an entry in *SendQ*.
2. Event e_j is an LCP' event if and only if event e_j is an entry in *RecvQ*.

These two properties establish that our technique for identifying LCP and LCP' events is valid. Two lemmas are prerequisites for proving these properties.

Lemma 6.4 *If* `Find_LCPs()` *adds send event* e_k *to SendQ,* e_k *is an event of a non-repetitive communication path, and* e_k *is an* LCP *event.*

Proof.

Event e_i is the assert event of P_i.

BASIS.

> If $e_k = c{:}kSi$, $k \in$ *Rec_wo_Sends* and *RwoSQ* has the entry <*c, i, k, POG_node*>, then *c:kSi* is an event of the non-repetitive communication path *c:kSi, c:iRk* and e_k is an LCP event.
>
> Proof: By definition 5.2, *c:kSi, c:iRk* is a a non-repetitive communication path. The send event *c:kSi* is the LCP event of P_k if $c{:}kSi \rightarrow c{:}iRk$ and there does not exist another send event e'_k such that $c{:}kSi \rightarrow e'_k \rightarrow c{:}iRk \rightarrow e_i$. Since *c:kSi* is the corresponding send to *c:iRk* (*RwoSQ* has the entry <*c, i, k, POGnode*>) then $c{:}kSi \rightarrow c{:}iRk \rightarrow e_i$, and since $k \in$ *Recv_wo_Send*, e'_k does not exist.

INDUCTIVE HYPOTHESIS.

The events $jRk, jSl, lRj, \dots mSi, iRm$ form a non-repetitive communication path of length n, and $iRm \to e_i$.

INDUCTIVE STEP.

If $e_k = c{:}kSj$, then

1. the send event $c{:}kSj$ is added to the non-repetitive communication path jRk, jSl, lRj, $\dots, mSi, iRm$ to form the non-repetitive communication path kSj, jRk, jSl, lRj, $\dots$, mSi, iRm of length $n+1$, and
2. the send event $c{:}kSj$ is an *LCP* event.

Proof:

The event $c{:}kSj$ is the corresponding send of $c{:}jRk$, and the relationship $c{:}kSj \to c{:}jRk$ is true Therefore, $kSj, jRk, jSl, lRj, \dots, mSi, iRm$ is a non-repetitive communication path (definition 5.2) of length $n+1$. Event $c{:}kSj$ is added to *SendQ* by algorithm `Find_LCPs()` when it is found to be part of the non-repetitive communication path.

We know $c{:}kSj \to e_i$ since $c{:}kSj$ is an event of $kSj, jRk, jSl, lRj, \dots, mSi, iRm$ and $iRm \to e_i$. From the basis and $c{:}kSj \to e_i$, we can conclude that $c{:}kSj$ is an *LCP* event.

■

Lemma 6.5 *If* `Find_LCPs()` *adds receive event* jRk *to RecvQ,* jRk *is an event of a non-repetitive communication path and* jRK *is an* LCP' *event.*

Proof.

For event jRk to be added to *RecvQ*, k must first be an entry in *Rec_wo_Sends* and *RwoSQ* contains the entry for c:jRk. For these to exist, we know from algorithm `Find_LCPs()`

- $j \in Sends$ or $j = i$, and
- $k \notin FoundProcs$

Consider the two possibilities:

1. $j \in Sends$ and $k \notin FoundProcs$.

   From lemma 6.4, if $j \in Sends$, a non-repetitive communication path exists from a LCP send of P_j, jSl, to a LCP' event of P_i, iRm: $jSl, \ldots, iRm$. And for $k \notin FoundProcs$, the LCP and LCP' of P_k have not been found in the POG. We can also conclude that $jRk \rightarrow jSl$. For jRk to be added to *RecvQ*, the send kSj must have been found in an ancestor node of the of jRk (from algorithm `Find_LCPs()`). Therefore, $kSj \rightarrow jRk$. From this we can conclude the send event corresponding to jRk, kSj, is found and is an LCP event (Lemma 6.4). jRk is an LCP' event, and jRk is an event of the non-repetitive communication path $kSj, jRk, jSl, \ldots, iRm$.

2. $j = i$ and $k \notin FoundProcs$. Event jRk is a receive event of the process evaluating the assert, and the LCP event of P_k has not been found. Then for jRk to be added to *RecvQ*, the send kSj has been found in the POG, is the LCP event of P_k and forms the non-repetitive communication path kSj, jRk (lemma 6.4). Since jRk is the corresponding receive of kSj, jRk is an LCP' event.

■

Property 6.14 *Event e_j is an LCP event if and only if event e_j is an entry in SendQ.*

Proof.

PART 1. If e_j is an LCP event, then e_j is in *SendQ*.

Proof by contradiction.

Assume $e_j = jSk$ is an LCP event but is not in *SendQ*. Since jSk is an LCP event, there exists a non-repetitive communication path $jSk, kRj, kSl, lRk, \ldots, mSi, iRm$ that consists of LCP and LCP' event where $jSk \rightarrow kRj \rightarrow kSl \rightarrow lRk \rightarrow \cdots \rightarrow mSi \rightarrow iRm$ (theorem 5.1 and definition 5.2). For jSk to not be in *SendQ*, kRj is not in *RwoSQ* and k is not in *Rec_wo_Sends*. For this to occur either

1. $j \in$ *FoundProcs* or
2. $k \notin$ *Sends*.

1. For j to be in *FoundProcs*, another e'_j exists where e'_j is in *SendQ* and e'_j is an LCP event of P_j. But since P_j can have only one LCP event (lemma 4.1) a contradiction has been reached.
2. For $k \notin$ *Sends*, kSl, the LCP event of P_k, is not in *SendQ*. The same reasoning holds as to why each LCP event of the non-repetitive communication path $jSk, kRj, kSl, lRk, \ldots, mSi, iRm$ is not in *SendQ* except for mSi. For mSi to not be an LCP event, iRm is not recognized as an LCP' event. For iRm to not a LCP' event, m must be in *FoundProcs*. For m to be *FoundProcs*, a send event e_m and a receive event e'_i exist where $mSi \rightarrow iRm \rightarrow e_m \rightarrow e'_i$. Thus e_m is the

LCP event of P_m (definition 4.1). A contradiction has been reached since mSi is the *LCP* event of P_m.

PART 2. If e_j is in *SendQ*, then e_j is an *LCP* event.

Proof. This follows directly from lemma 6.4.

■

Property 6.15 *Event e_j is an LCP' event if and only if event e_j is an entry in RecvQ.*

Proof.

PART 1. If e_j is an LCP' event, then e_j is in *RecvQ*.

Proof.

If $e_j = jRk$ is a receive LCP'event, then jRk is part of a non-repetitive communication path to an LCP' event of P_i that consists of LCP and LCP' events, $jSk, kRj, kSl, lRk, \ldots, mSi, iRm$ (theorem 5.1). We know the LCP sends are entries in *SendQ* (property 6.14). If the sends are entries in *SendQ*, then the corresponding receives are also entries in *RecvQ* according to algorithm `Find_LCPs()`.

PART 2. If e_j is in *RecvQ*, then e_j is an LCP' event.

Proof. This follows directly from lemma 6.5.

■

6.5 *POG* and Taylor's Complete Concurrency History

Taylor's work motivated our static analysis to generate the *POG* for representing the possible executions of a distributed system, but our static analysis algorithms have been developed independent of Taylor's work. The only portion of our static analysis that is a derivation of Taylor's static analysis is representing each process with a flow graph and the successor relationship between nodes of the graph.

A path of the POG has a different meaning from a path in Taylor's complete concurrency history. A path of Taylor's history represents a possible total ordering of i/o rendezvous and does not represent the concurrent execution of i/o rendezvous. Each path of the *POG* represents a partial order of the distributed system, and a path does represent the concurrency of the communication commands. One or more of Taylor's paths can correspond to one path of the POG since one or more total orders can correspond to the same partial order.

6.6 Static Analysis in the Parallel Domain

Work in the parallel domain that is most closely related to ours is the automated parallelization of sequential code. Parallelizing compilers collect data flow information for a source program and use this information to detect potential parallelism, determine an appropriate grain size, and then transform the program into a functionally equivalent parallel program that can exploit the underlying architecture. These compilers also aim at automating the selection of data distributions and reducing nonlocal data accesses in distributed memory systems.

The majority of the data flow analysis performed by these compilers is dependence analysis. Two computations that have a dependence relationship means that constraints on their execution order are present. By identifying these constraints with dependence analysis, it can be determined whether transformations of the source code will alter the semantics of the computation.

Two types of dependencies that can be identified with data flow analysis are data and control. Consider two statements, *a* and *b*, of a sequential program. Statement *b* is control dependent on statement *a*, if *a* determines whether *b* executes. Statements *a* and *b* have a data dependence if they cannot be executed simultaneously because of conflicting uses of the same variable.

Dependence analysis performed at the procedure and function level is useful for identifying coarse grain parallel transformations [35, 17, 24, 23, 36]. Dependence analysis performed at the loop level is useful for identifying fine grain parallelism [6, 10, 11, 9, 25, 36]. Languages, such as Fortran D [34], provide commands the programmer uses to annotate the sequential program with data decompositions. The compiler then performs dependence analysis to determine the computation decomposition [2]. Other languages [32, 2] exist in which the compiler determines both data and computation decompositions with the aid of dependence analysis.

The objective of the compiler is to produce parallel code in which the execution is maximally parallel and nonlocal data accesses are minimized. Dependency analysis provides information for achieving this objective.

Initially our static analysis appeared similar to the static analysis performed by parallelizing compilers. By comparing the two more closely, the similarities are only superficial. A parallelizing compiler generates control flow graphs of the sequential program and performs sequential data flow analysis. The compiler uses these results to create a functionally equivalent parallel program and decompose the sequential program's data. As part of this process, the necessary communication commands are also created. Our work generates control flow graphs for the source code of the distributed processes to analyze the communication. The source code is already comprised of communication commands. We do not perform dependence analysis and we do not add communication to the distributed system.

In the next chapter, the analysis of distributed programs with the addition of loops is described. The distributed programs in chapter 2 are analyzed in chapter 8, and the LCP and LCP' events determined. These programs further demonstrate the benefits of identifying LCP and LCP' events for reducing the number of messages that piggyback data.

Chapter 7

Loops

Chapter 6 presented algorithms for creating the FG_i, H and POG graphs. Algorithms where also presented for determining the LCP and LCP' events of an assert statement from the POG. These algorithms did not support loops in the source code of the distributed processes. In this chapter we make the additions to the algorithms to allow loops, and the algorithms are modified so all properties and lemmas of chapter 6 are preserved. By concluding with the preservation of properties 6.14 and 6.15, we demonstrate that our technique remains valid for identifying LCP and LCP' events.

7.1 Control Flow Graphs

Three loops constructs can occur in the source code of a process: `do - while`, `while`, and `for`. Each loop has one unique entry point and one unique exit point. Nesting of loops are allowed, but each loop has its own entry and exit point. Neither `goto` nor `break` statements are allowed in the source code since either can create additional entry or exit points for loops.

Algorithm **Create_FG$_i$()** requires additions for representing loops in FG_i. Each loop in a process's source code is represented as a cycle in the process's corresponding flow graph. The cycle is accomplished with a *back edge* from the exit point of the loop to the entry point of the loop. The concept of a *dominating* node is necessary to define a back edge. A node a of flow graph FG_i *dominates* node b of FG_i if every path from the root node of FG_i to node b passes though a. If (a, b) is an edge, then a is the initial node and b is the terminal node. An edge is a back edge if its terminal node dominates its initial node. An edge of a flow graph that is not a back edge is referred to as either a forward edge or an edge.

The control flow graph for a process, FG_i, requires additional node types for representing loops. The entry point of a loop is represented with a head node, and the exit point of a loops is represented with a tail node. The head and tail of a `while` loop are nodes labeled WHILE and END_WHILE, respectively. The head and tail of a `do - while` loop are nodes labeled DO and END_DO, respectively. The head and tail of a `for` loop are nodes labeled FOR and END_FOR, respectively. The nodes that occur between the head and tail nodes make up the body of the loop.

The `while` and `for` loop are similar in that the loop condition is evaluated at the head of the loop. The loop body is executed zero or more times. This type of loop is referred to as a *precondition* loop. The loop condition of the `do` loop is evaluated at the tail of the loop so the loop body is executed one time before testing the condition. This type of loop is referred to as a *postcondition* loop.

Algorithms **Create_FG$_i$()** and **AddNode()** are repeated from chapter 6 with the additions required for the loop constructs. Figure 7.1 shows the three loop constructs represented by a portion of a control flow graph. The dashed edges between nodes indicate the edges added by algorithm **AddNode()** when *NewNode* is added to the flow graph. A back edge is added by **Create_FG$_i$()** for any one of the loop constructs from the tail of the loop (e.g., END_WHILE node) to the head of the loop (e.g., WHILE node). The back edge creates a cycle in the graph.

Three additional stacks and three additional variables are required to handle loops in algorithm **Create_FG$_i$()**. The stacks are *WhileStack*, *DoStack* and *ForStack*. The three pointer variables are *TopDoStack*, *TopWhileStack*, and *TopForStack*. Each pointer references the top entry of its respective stack. The stacks are initially empty, and the pointers are initially NULL. The stacks are used to match the begin and end of the loop constructs.

```
Create_FG_i()                                            /* Input: P_i; Output: FG_i */
    Create the ROOT node of FG_i
    CrtNode = ROOT node
    if an assignment statement is recognized
        Add assignment statement to the tail of the linked list
    if an async_send is recognized
        if the linked list is not empty
            AddNode(CrtNode, ASSIGN)             /* for the assignment statements */
            linked list is set to empty
        AddNode(CrtNode, SEND)
    if an async_recv is recognized
        if the linked list is not empty
            AddNode(CrtNode, ASSIGN)             /* for the assignment statements */
            linked list is set to empty
        AddNode(CrtNode, RECEIVE)
    if an assert is recognized
        if the linked list is not empty
            AddNode(CrtNode, ASSIGN)             /* for the assignment statements */
            linked list is set to empty
        AddNode(CrtNode, ASSERT)
    if an if statement is recognized
        if the linked list is not empty
```

```
            AddNode(CrtNode, ASSIGN)                /* for the assignment statements */
            linked list is set to empty
        AddNode(CrtNode, IF)                        /* for the if statement */
        Push CrtNode onto the stack
        TopStack = CrtNode
    if an else is recognized
        AddNode(CrtNode, END_IFSIDE)
        if the linked list is not empty
            Set field in CrtNode to point to linked list
            linked list is set to empty
        TopStack.HoldPtr = CrtNode                  /* Set HoldPtr of the IF node to the */
                                                    /* address of the END_IFSIDE */
        CrtNode = top entry of the stack
        CrtNode.AddEdgeFlag = true                  /* Flag an edge needed from END_IFSIDE node */
                                                    /* to the first node following END_ELSE node */
    if the end of the else side of an if/else is recognized
        AddNode(CrtNode, END_ELSE)                  /* for the ending of the else side */
        if the linked list is not empty
            Set field in CrtNode to point to linked list
            linked list is set to empty
        CrtNode.HoldPtr = TopStack.HoldPtr          /* Move the address of the END_IFSIDE */
                                                    /* node to the END_ELSE node */
        CrtNode.AddEdgeFlag = true                  /* Flag an edge will be needed from END_IFSIDE */
                                                    /* node to the first node following END_ELSE node */
        Pop the stack
    if the end of an if statement is recognized
        AddNode(CrtNode, END_IF)                    /* for the ending of the if statement */
        if the linked list is not empty
            Set field in CrtNode to point to linked list
            linked list is set to empty
        CrtNode.HoldPtr = TopStack                  /* Set the HoldPtr of END_IF node */
                                                    /* to the address of the IF node */
        CrtNode.AddEdgeFlag = true                  /* Flag an edge will be need from the IF node */
                                                    /* to the first node following the END_IF node */
        Pop the stack
    if a while statement is recognized
        if the linked list is not empty
            AddNode(CrtNode, ASSIGN)                /* for the assignment statements */
            linked list is set to empty
        AddNode(CrtNode, WHILE)                     /* for the if statement */
        Push CrtNode onto WhileStack
        TopWhileStack = CrtNode
    if a for statement is recognized
        if the linked list is not empty
            AddNode(CrtNode, ASSIGN)                /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, FOR)                                        /* for the if statement */
    Push CrtNode onto ForStack
    TopForStack = CrtNode
if a do statement is recognized
    if the linked list is not empty
        AddNode(CrtNode, ASSIGN)                          /* for the assignment statements */
        linked list is set to empty
    AddNode(CrtNode, DO)                                         /* for the if statement */
    Push CrtNode onto DoStack
    TopDoStack = CrtNode
if the end of a while loop is recognized
    AddNode(CrtNode, END_WHILE)                          /* for the end of the while loop */
    if the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
    Add back pointer from CrtNode to TopWhileStack         /* create cycle in the graph */
    CrtNode.HoldPtr = TopWhileStack             /* Set the HoldPtr of END_WHILE node */
                                                   /* to the address of the WHILE node */
    CrtNode.AddEdgeFlag = true      /* Indicate an edge will be needed from the WHILE */
                                  /* node to the first node following the END_WHILE node */
    Pop WhileStack
if the end of a for loop is recognized
    AddNode(CrtNode, END_FOR)                              /* for the end of the for loop */
    if the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
    Add back pointer from CrtNode to TopForStack            /* create cycle in the graph */
    CrtNode.HoldPtr = TopForStack                   /* Set the HoldPtr of END_FOR node */
                                                     /* to the address of the FOR node */
    CrtNode.AddEdgeFlag = true           /* Indicate an edge will be need from the FOR */
                                    /* node to the first node following the END_FOR node */
    Pop ForStack
if the end of a do loop is recognized
    AddNode(CrtNode, END_DO)                                /* for the end of the do loop */
    if the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
                                                 /* create cycle in the graph for loop */
    Add back pointer from CrtNode to node reference by TopDoStack
    Pop DoStack
if the current control construct or statement is not recognized
    Generate an error and halt
if the end of the source code is recognized
    AddNode(CrtNode, END)
    If the linked list is not empty
        Set field in CrtNode to point to linked list
        linked list is set to empty
end algorithm
```

The only valid exit point of a postcondition loop is from the tail of the loop. Algorithm **AddNode()** creates an edge from the END_DO node to the first node added to the graph after the END_DO node (*NewNode*). The only valid exit point of a precondition loop is from the head of the loop. Algorithm **AddNode()** creates an edge from the WHILE node and the FOR node to the first node that occurs after the loop's end node.

```
AddNode(CrtNode, type)
    NewNode = Allocate a node
    if CrtNode $\neq$ END_WHILE, END_FOR
        Create a directed edge from CrtNode to NewNode
    if CrtNode.AddEdgeFlag
        Create a directed edge from the node CrtNode.HoldPtr to NewNode
    if type = ASSIGN
        Set field in NewNode to point to assignment linked list
    CrtNode = NewNode
end algorithm
```

Algorithm **AddNode()** does not require additional code or modification to create the edge from the exit point of a **do-while** loop. Additional code is required for the exit point of the **while** and **for** loops. To create an edge from a WHILE or FOR node to *NewNode*, the same steps are taken when an edge is added from an IF node to the END_IF node. The description of this process is presented in terms of the **while** loop, but is generalized to any precondition loop. When the END_WHILE node is added, the address of the WHILE node, available on top of *WhileStack*, is stored in the END_WHILE node. This is accomplished with the following line from **Create_FG$_i$()**:

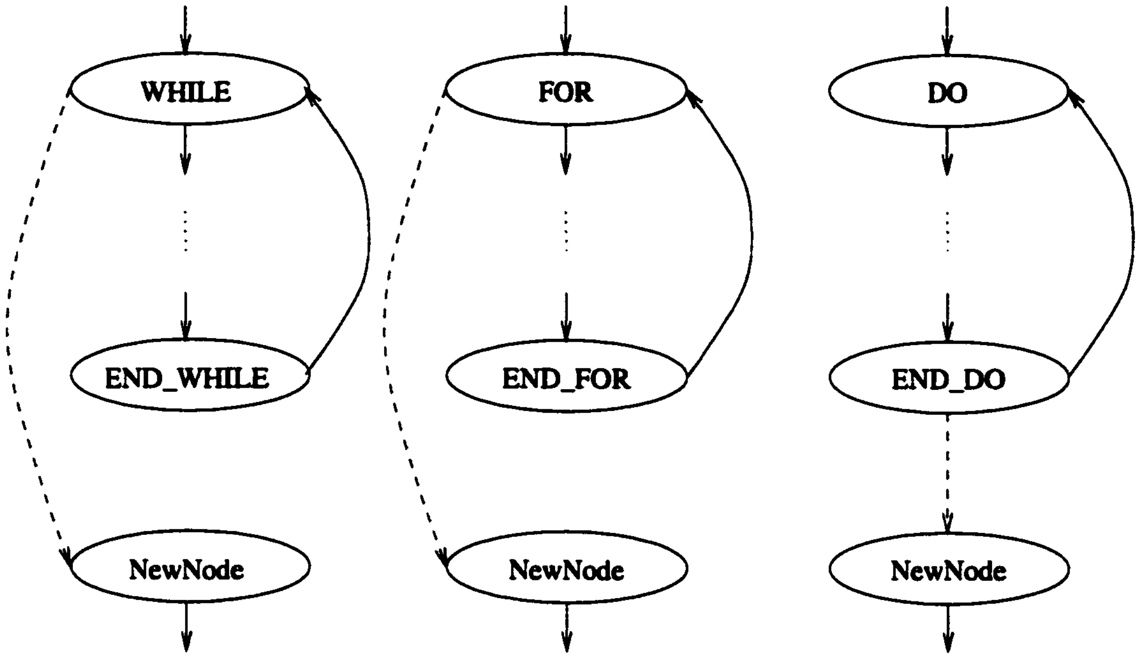


Figure 7.1: Control flow graph of the loop constructs

CrtNode.HoldPtr = TopWhileStack

When *NewNode* is added to FG_i, the address of the WHILE node is available in *CrtNode* so that **AddNode()** can create an edge from the WHILE node to *NewNode*. The flag *AddEdgeFlag* of the END_WHILE node is set to true to indicate that function **AddNode()** should add an edge from the WHILE node to *NewNode*.

Properties 6.1 through 6.4 correspond to the control flow graphs. Properties 6.2 and 6.4 are not affected by loops, but property 6.1 requires some modification when loops occur in the source code.

First we will consider precondition loops. Consider the `while` loop shown in figure 7.2. If the loop is executed zero times, the happens before relationship among the nodes is:

WHILE → S2

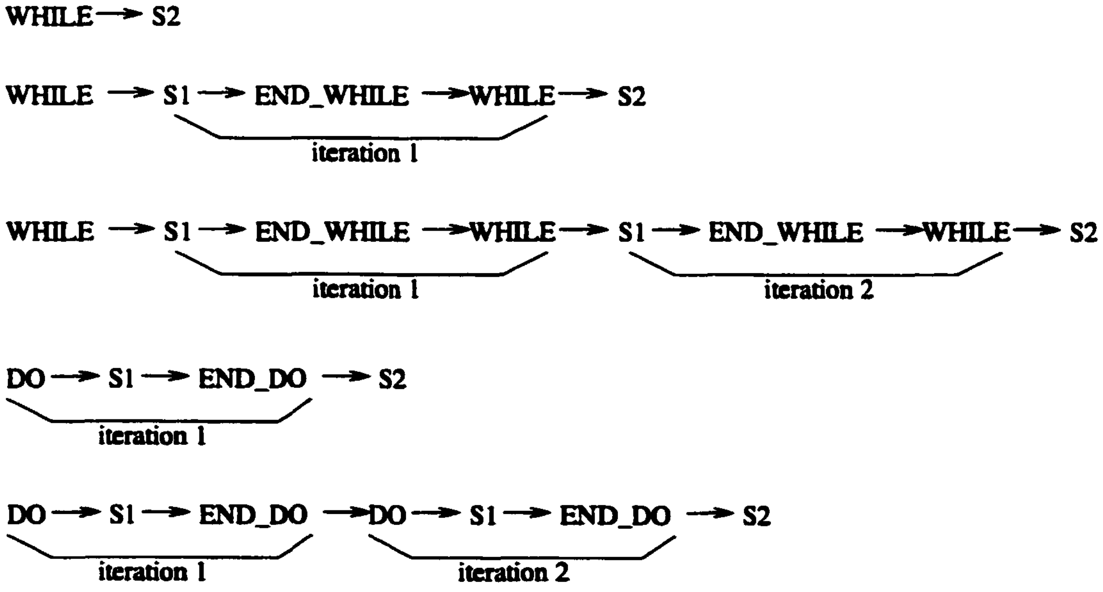
If the loop is executed one time, the happens before relationships among the nodes are:

WHILE → S1 → END_WHILE → WHILE → S2

A new iteration of the loop is begun when the first node of the loop body is executed. In this example, node S1 is the first node of the loop body. The last node of an iteration is the WHILE node. If i iterations of a precondition loop occur, the WHILE node is executed $i+1$ times, and the back edge is followed i times. Consider the case when the loop is executed two times. The happens before relationships are:

WHILE → S1 → END_WHILE → WHILE → S1 → END_WHILE → WHILE → S2

The following summarizes the happens before relationships and the beginning and ending of loop iterations for the `while` loop.


WHILE → S2
WHILE → S1 → END_WHILE → WHILE → S2
iteration 1
WHILE → S1 → END_WHILE → WHILE → S1 → END_WHILE → WHILE → S2
iteration 1
iteration 2
DO → S1 → END_DO → S2
iteration 1
DO → S1 → END_DO → DO → S1 → END_DO → S2
iteration 1
iteration 2

The happens before relationship and the beginning and ending of loop iterations are also shown for `do-while` loops, which will be discussed next.

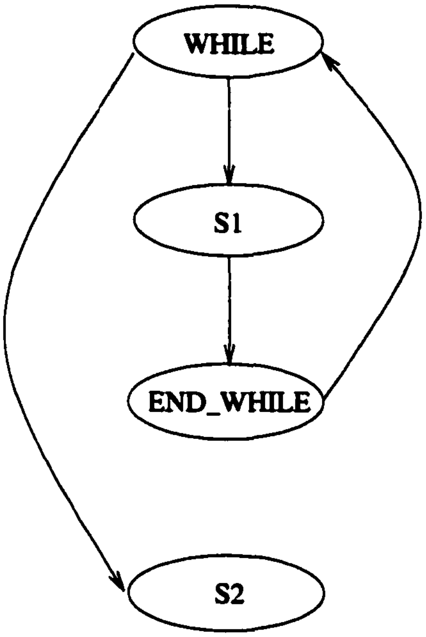


Figure 7.2: Control flow graph with a while loop

According to property 6.1, if a path exists from node a to node b, then $a \rightarrow b$ when both are executed. By examining the happens before relationship between the S1 node and the END_WHILE node, we see that property 6.1 requires updating. A path exists from node END_WHILE to node S1 in figure 7.2, but it is not the case that END_WHILE → S1 when one iteration of the loop occurs. Consider two iterations of the loop. The END_WHILE of the first iteration does not happen before the S1 of the first iteration, but the END_WHILE of the first iteration does happen before the S1 of the second iteration. In general, END_WHILE → S1 if

1. the loop is executed 2 or more times

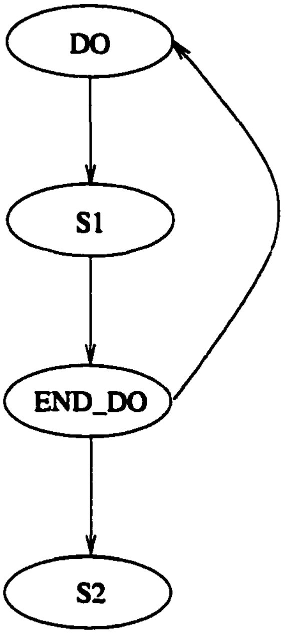


Figure 7.3: Control flow graph with a do - while loop

2. END_WHILE occurs in iteration i of the loop, and
3. S1 occurs in iteration $i + 1$ or greater.

Next we need to examine the postcondition loop. Figure 7.3 shows a flow graph for a do loop. If the loop is executed only one time, the happens before relationships among the nodes are:

DO → S1 → END_DO → S2.

The happens before relationships for two executions of the loop are:

DO → S1 → END_DO → DO → S1 → END_DO → S2.

The boundary nodes of an iteration for a postcondition loop are different than those of a precondition loop. The first execution of DO begins iteration 1 of the loop, and END_DO completes that iteration. If the loop is iterated i times, DO and END_DO are executed i times, and the back edge is followed $i - 1$ times.

Consider the happens before relationship between S1 and END_DO. A path exists from the END_DO node to the S1 node, but the relationship END_DO → S1 is true only if

1. the loop is executed more than once,
2. END_DO occurs in iteration i, and
3. S1 occurs in an iteration greater than i.

Property 6.1' subsumes property 6.1 to account for the occurrence of loops. The property is given in two parts for completeness, but only part 1 is modified. The variable l is used to denote a loop.

Property 6.1'

PART 1. *If a path exists from node a to node b in FG_i, then $a \rightarrow b$ when*

1. *a and b are both executed and a back edge is not part of the path from node a to node b, or*
2. *a and b are both executed, the back edge of loop l is part of the path from node a to b, node a occurs in iteration i of loop l and node b occurs in iteration j, where $j > i$, of loop l, or*
3. *a and b are both executed, the back edge of loop l is part of the path from node a to b, loop l is a precondition loop, nodes a and b occur in the same iteration, and node b is the head of the loop.*

PART 2. *If $a \rightarrow b$ when both a and b are executed, then a path exists from node a to node b in FG_i.*

Condition (1) of part 1 is equivalent to property 6.1. Conditions (2) and (3) quantify which happen before relationships are possible with the addition of back edges. Condition (2) of part 1 allows the relationship END_WHILE → S1 of figure 7.2 when multiple iterations of the loop occur, and S1 occurs in a later iteration than END_WHILE. Also notice that this condition allows the happens before relationship WHILE → WHILE where the first WHILE occurs in an earlier iteration than the second. As for postcondition loops, the condition S1 → DO is allowed for figure 7.3 when two or more iterations occur. Condition (3) of part 1 allows S1 → WHILE when both occur in the same iteration.

7.2 *H* Graph

With the possibility of loops in the source code of each process of the distributed system, loops are also possible in H. In the algorithm for constructing H, additions are required for detecting the repeated execution of communication commands and representing these repetitions as cycles in H. Cycles occur in H if

1. a send command is in the body of a loop and the send is *possible*,
2. a send command is *possible*, the matching receive is *ready*, and both occur in the body of a loop, or
3. a combination of (1) and (2).

Cycles are created in H with back edges. The graph retains the properties of a tree; there exist a root node and leaf nodes. The terminology *ancestor*, *descendent*, *parent* and *child* will remain in use for the relationships defined by forward edges. The relationships between nodes defined by back edges will be discussed following the modified `Crt_H()` algorithm.

Properties 6.5 through 6.8 and lemma 6.1 correspond to the H graph. The substantial changes to the H graph construction algorithm do not invalidate these properties and lemma. The node relationship *ancestor* is fundamental to properties 6.5 and 6.7, and these properties remain valid with the clarification of the *ancestor* relationship. Properties 6.6 and lemma 6.1 are not affected by loops. Property 6.8 is discussed following the modified `Crt_H()` algorithm.

```
P0                       P1
 do                       do
   async_send(1,x)          async_recv(0,y)
 while                    while
```

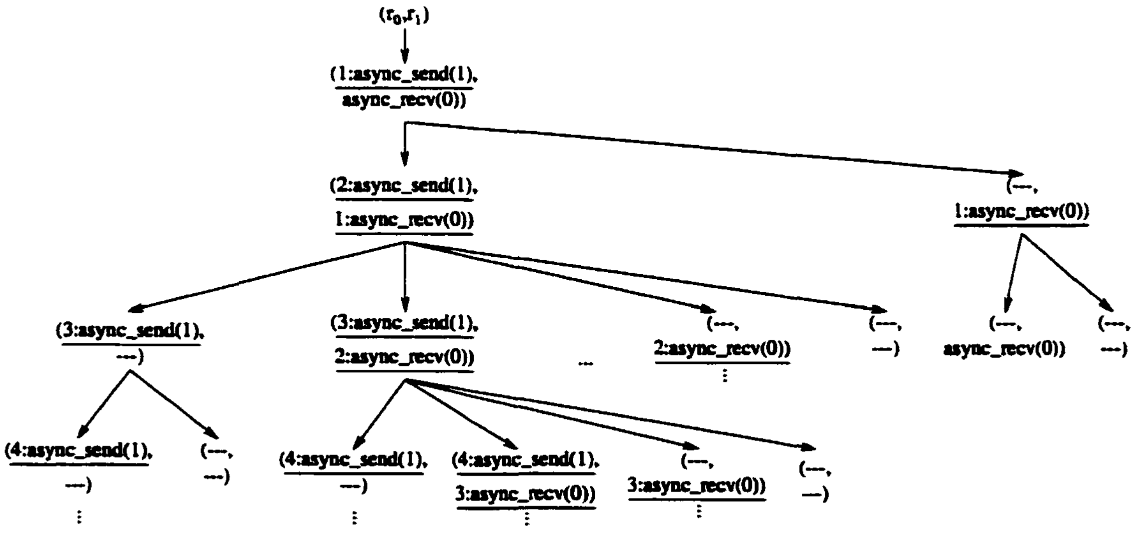

(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(—, 1:async_recv(0))
(3:async_send(1), —)
(3:async_send(1), 2:async_recv(0))
...
(—, 2:async_recv(0))
(—, —)
(—, async_recv(0))
(—, —)
(4:async_send(1), —)
(—, —)
(4:async_send(1), —)
(4:async_send(1), 3:async_recv(0))
(—, 3:async_recv(0))
(—, —)

Figure 7.4: Example 1

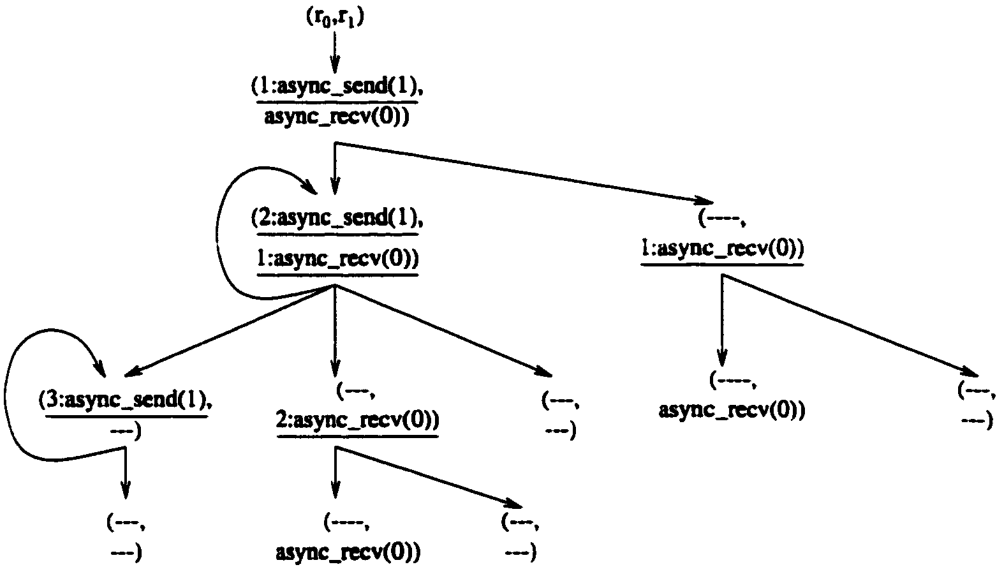


Figure 7.5: Example 1 with back edges

The detection of loops in H requires significant additions to the `Crt_H()` algorithm. Two examples, useful for describing the additions to `Crt_H()`, demonstrate the occurrence of loops in H. The first example is a two process distributed system. The source code of each process and the graph resulting from algorithm `Crt_H()` in chapter 6 is shown in figure 7.4. The graph H can not accurately represent the execution of this distributed system without back edges. Communication commands are repeatedly executed, but the loops are not shown as cycles in H since this version of the algorithm does not detect loops. A pattern can be observed in H. The nodes (2:async_send(1), 1:async_recv(0)) and (3:async_send(1), 2:async_recv(0)) of figure 7.4 represent the same state of the system. Although the counters corresponding to the sends and receives differ, the send in each node represents the same command in P_0, and the receive in each node represents the same command in P_1. Another system state is represented by nodes (3:async_send(1),—) and (4:async_send(1),—) of figure 7.4. The complete H graph with the inclusion of back edges is shown in figure 7.5.

```
P0                        P1
do                        do
  async_send(1,x)           async_recv(0,y)
  async_send(1,x)           async_recv(0,y)
while                     while
```

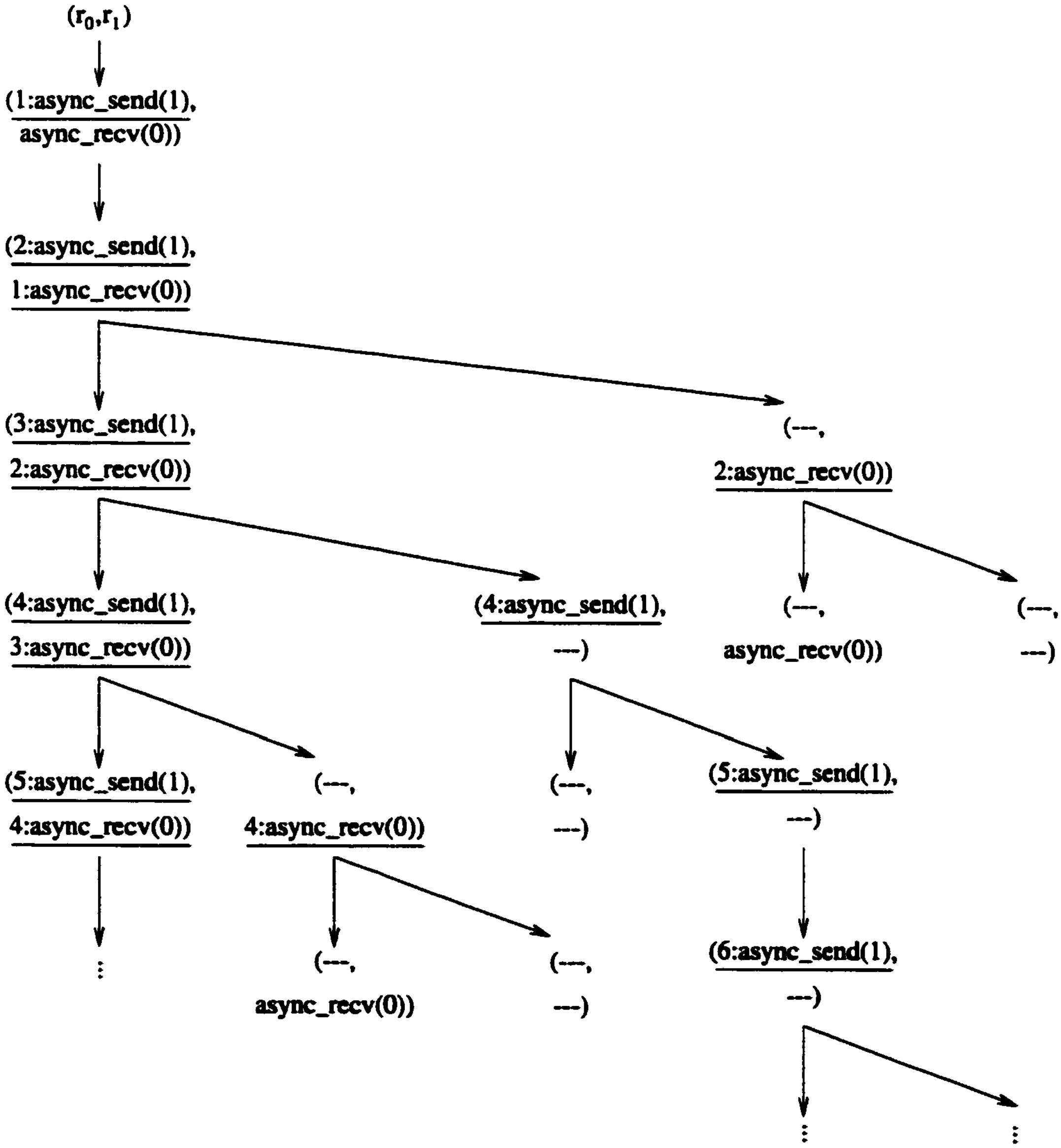

(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))
(—, 2:async_recv(0))
(4:async_send(1), 3:async_recv(0))
(4:async_send(1), —)
(—, async_recv(0))
(—, —)
(5:async_send(1), 4:async_recv(0))
(—, 4:async_recv(0))
(—, —)
(5:async_send(1), —)
⋮
(—, async_recv(0))
(—, —)
(6:async_send(1), —)
⋮
⋮

Figure 7.6: Example 2

```
P0                      P1
do                      do
  async_send(1,x)         async_recv(0,y)
  async_send(1,x)         async_recv(0,y)
while                   while
```

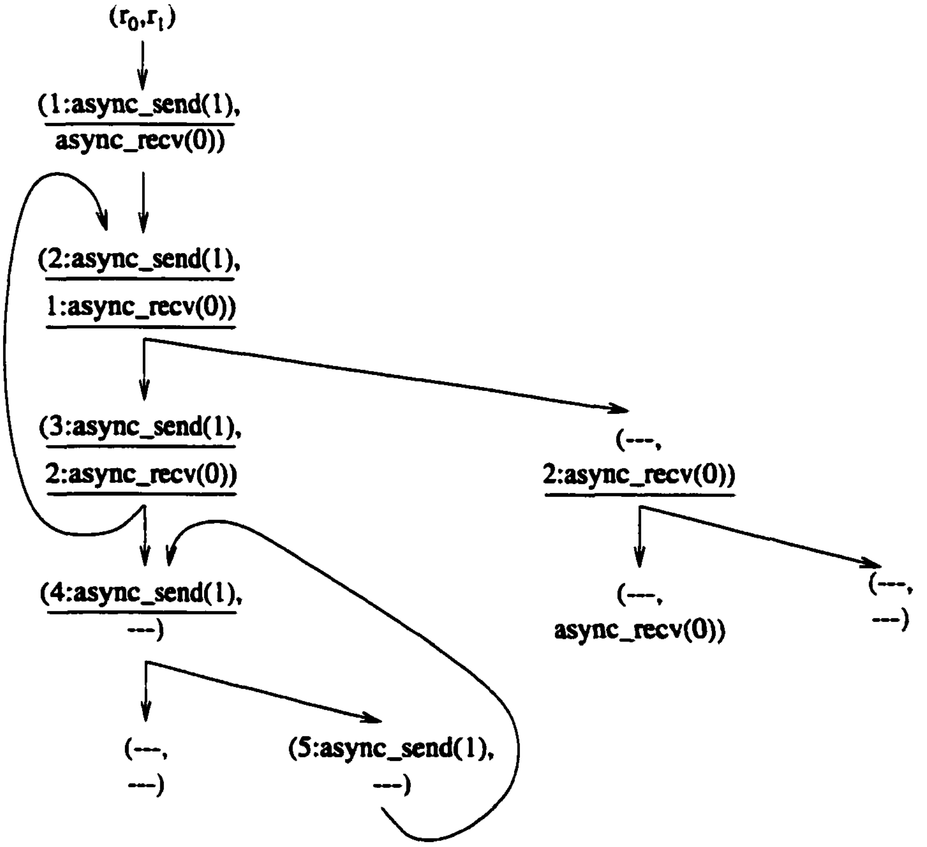

(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))
(—, 2:async_recv(0))
(4:async_send(1), —)
(—, async_recv(0))
(—, —)
(—, —)
(5:async_send(1), —)

Figure 7.7: Example 2 with back edges

Loop detection is more difficult in the example of figure 7.6. The nodes (2:async_send(1), 1:async_recv(0)) and (3:async_send(1), 2:async_recv(0)) syntactically appear to represent the same state, but they do not. The node (2:async_send(1), 1:async_recv(0)) represents the first send of P_0 and the first receive of P_1, whereas the node (3:async_send(1), 2:async_recv(0)) represents the second send in P_0 and the second receive in P_1. Nodes

(2:async_send(1), 1:async_recv(0)) and (4:async_send(1),3:async_recv(0)) represent the same state of the system, and nodes (3:async_send(1),2:async_recv(0)) and (5:async_send(1),4: async_recv(0)) represent another state of the system. The complete H graph representing the execution of the distributed system with back edges is given in figure 7.7

Additional information is required to detect and represent loops in H. For each node, n, of H the following information is needed.

- A temporary back edge, *temp_back*, used by `Crt_H()` is a field of n.
- An array of node pointers that are the children of n. *Kids[MAXEDGES]* is a field of n. Each entry represents a child that is the result of a forward or back edge. The forward edge children occur first in the array.
- An array of integers *Kid_type[MAXEDGES]*, where *Kid_type[i]* indicates the type of edge for *Kids[i]*, is a field of n. A zero entry indicates a forward edge, and a one entry indicates a back edge.
- An array of pointers to the parents nodes of n, *Parents[2]*, is a field of n. Entry *Parents[0]* is the parent of n that is defined by a forward edge. Each node has a parent from a forward edge. If a node is pointed to by a back edge, then the node also has a parent that is defined by a back edge. The entry *Parent[1]* is the parent of n that is defined by a back edge or NULL if the n is not pointed to by a back edge. An example of a parent resulting from a back edge is node (3:async_send(1), 2:async_recv(0)) which is a parent of (2:async_send(1), 1:async_recv(0)) of figure 7.7.

When node n is added to H, a check is made to determine if the state represented by this node has already been represented by another node in n's execution path. This is done by comparing n with its ancestors. First n is compared with its parent. If the parent does not represent the same state, then the grandparent is compared against n. This continues until either a node that represents the same state of n is found or the root node is reached.

Two comparisons are required to determine if node n and its ancestor node, n', represent the same state. The first comparison identifies *syntactically* identical nodes. *Syntactically* identical meaning that for each entry, v_i, in n, there exists v_i' in n' which is identical with the exception of the counter value. If nodes n and n' are syntactically identical, the second comparison is necessary to determine whether the nodes represent the same state. For each pair of entries, v_i and v_i', where v_i and v_i' are not equal to the inactive marker, the test insures that *FGnode[i]* of n is equal to *FGnode[i]* of n'. If *FGnode[i]* of n is equal to *FGnode[i]* of n', both point to the same node of FG_i. Passing the test implies that v_i and v_i' represent the same command of P_i. If the test is true for each pair, (v_i, v_i'), then the two nodes represent the same state.

If the ancestor node n' represents the same state as node n, then n' is possibly the entry point of a loop, and the parent of n is possibly the exit point of this loop. The next decision is whether to add a back edge from the parent of n to n' to form the loop. Two cases exist for the relative location of nodes n and n' in H.

1. The parent of node n is also node n'.

2. The parent of node n is not node n'. Node n' is an ancestor of the parent of node n.

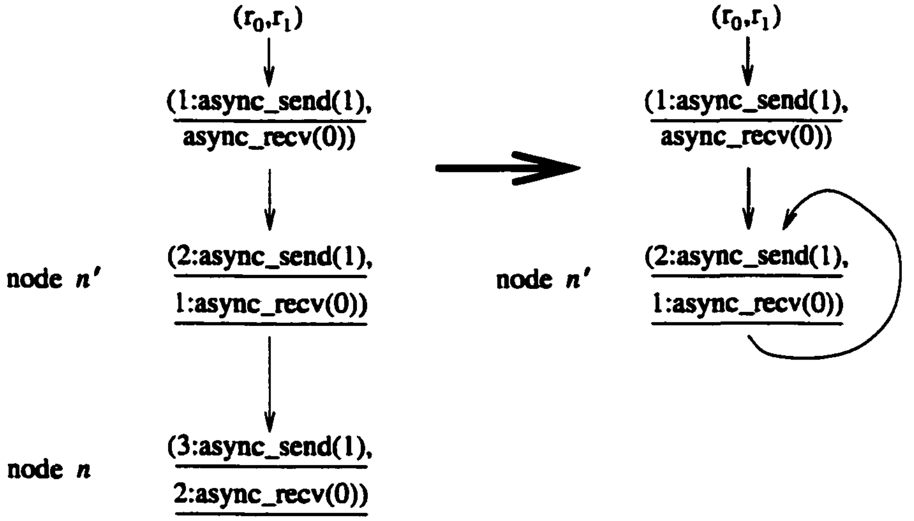

(r_0,r_1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))
node n'
node n
(r_0,r_1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
node n'

Figure 7.8: Case 1

If case 1 is true then a loop has been detected in H. A back edge is added from n' to itself, and node n is removed from H. Figure 7.8, the portion of figure 7.4 needed to demonstrate case 1, shows the transformation of H when the loop is detected. Case 2 requires more information to determine whether a loop has been found in H. Figure 7.9 is a distributed system that demonstrates case 2. Nodes n' and n represent the same state, but adding a back edge from the parent of n to n' would be incorrect. The state represented by node (5:async_send(1), 2:async_recv(2), —) does not recur after node n. Additionally, the state represented by nodes n and n' does not recur after node n.

Continuing to generate nodes of the execution path that includes n and n' is necessary to determine if a loop exists in H. If the nodes from n' to the parent of n are duplicated immediately after n, a loop exists in H. A back edge is added from the parent of n to n' creating a cycle. Node n and its descendants are removed from H.

The algorithm `Check_Loops()` checks for cases 1 and 2. Determining whether a back edge should be added for case 2 requires the use of field *temp_back*. Whenever case 2 is true. `Check_Loops()` sets the field *temp_back* of n's parent to point to n'. Figures 7.10 and 7.11 demonstrate the use of *temp_back*. The dashed edge represents the value of *temp_back*. Figures 7.10 and 7.11 show the generation of H in figure 7.6 as each node is added. Only the portion of H relevant to the addition of a back edge is shown.

Step 3 of figure 7.10 shows the first occurrence case 2. Nodes n and n' represent the same state and a temporary back edge (*temp_back*) is added from the parent of n to n'. A back edge can not be added until it is known that the state represented by node (3:async_send(1). 2:async_recv(0)) occurs again immediately after node n. The node added in step 4. (5:async_send(1), 4:async_recv(0)). represents the same state as node (3:async_send(1). 2:async_recv(0)). A temporary back edge is added from the parent of n (4:async_send(1). 3:async_recv(0)) to n' (3:async_send(1). 2:async_recv(0)). When this temporary back edge is added. n' also has a temporary back edge that points to node (2:async_send(1), 1:async_recv(0)). This indicates that the state of nodes (2:async_send(1), 1:async_recv(0)) and (3:async_send(1). 2:async_recv(0)) are repeated by (4:async_send(1), 3:async_recv(0)) and (5:async_send(1), 4:async_recv(0)) nodes. The temporary back edge of node n' becomes the back edge, and the nodes after n' are removed as shown in the resulting H.

```
P0                     P1                      P2
 ⋮                      ⋮                       ⋮
 async_send(1,x)        async_recv(0,y)         async_send(1,z)
 do                     do
   async_send(1,x)        async_recv(0,y)
 while(...)               async_recv(2,y)
                        while(...)
```

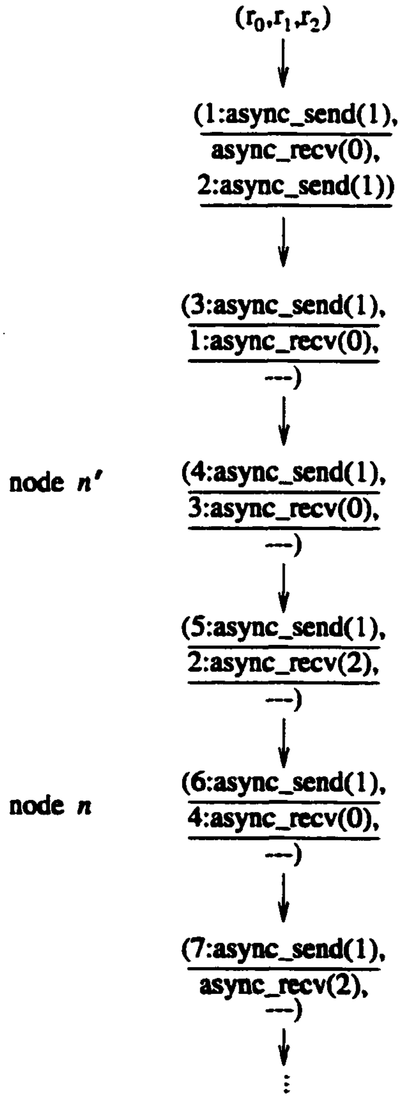

(r0,r1,r2)
(1:async_send(1), async_recv(0), 2:async_send(1))
(3:async_send(1), 1:async_recv(0), —)
node n'
(4:async_send(1), 3:async_recv(0), —)
(5:async_send(1), 2:async_recv(2), —)
node n
(6:async_send(1), 4:async_recv(0), —)
(7:async_send(1), async_recv(2), —)
⋮

Figure 7.9: Case 2

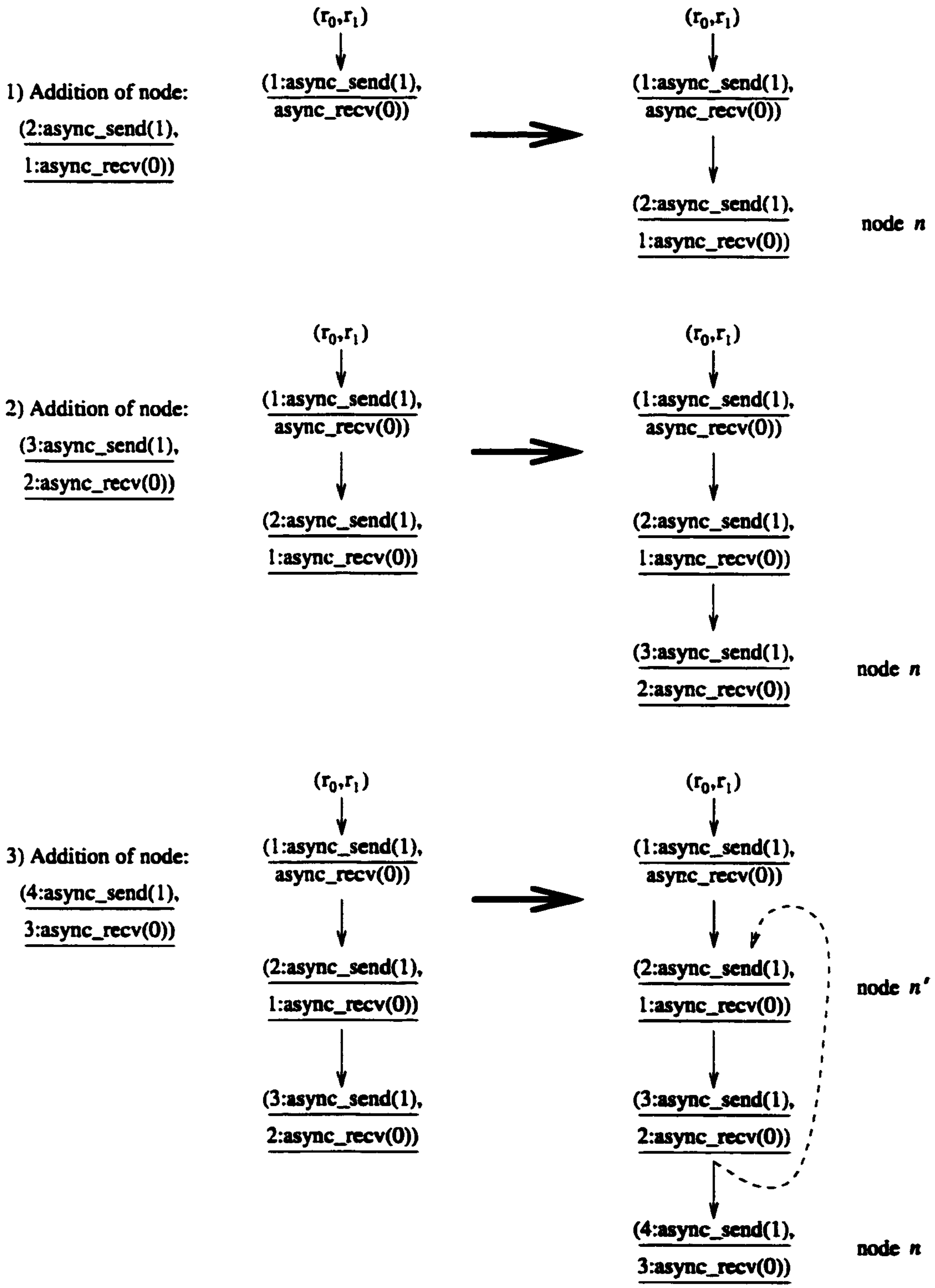

1) Addition of node:
(2:async_send(1), 1:async_recv(0))
(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
node n
2) Addition of node:
(3:async_send(1), 2:async_recv(0))
(3:async_send(1), 2:async_recv(0))
node n
3) Addition of node:
(4:async_send(1), 3:async_recv(0))
node n'
(4:async_send(1), 3:async_recv(0))
node n

Figure 7.10: Detecting a loop

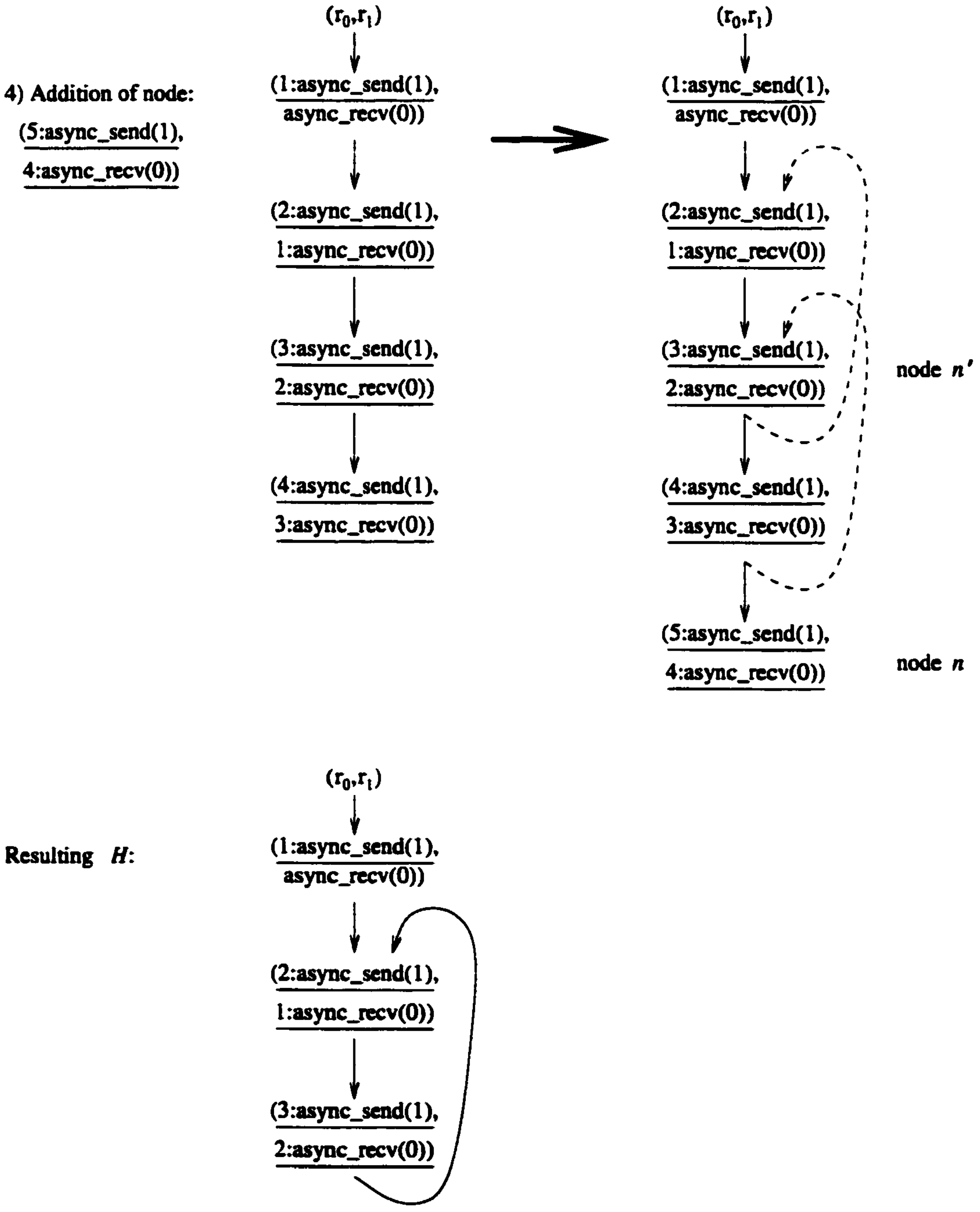

4) Addition of node:
(5:async_send(1), 4:async_recv(0))
(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))
(4:async_send(1), 3:async_recv(0))
(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))
node n'
(4:async_send(1), 3:async_recv(0))
(5:async_send(1), 4:async_recv(0))
node n
Resulting H:
(r0,r1)
(1:async_send(1), async_recv(0))
(2:async_send(1), 1:async_recv(0))
(3:async_send(1), 2:async_recv(0))

Figure 7.11: Detecting a loop

When node n is added to H, algorithm `Crt_H()` invokes algorithm `Check_Loops()` to check for the existence of a loop with the additional node. Algorithm `Check_Loops()` checks the ancestors of n for a node representing the same state as n. If one is found, the variable *Poss_Head* is set to the matching node, and variable *Poss_Tail* is set to the parent of n. If *Poss_Head* and *Poss_Tail* refer to the same node, an occurrence of case 1 is found, a back edge is added from *Poss_Head* to itself, and node n is removed from H. If case 2 is verified, nodes from the parent of *Poss_Tail* to *Poss_Head* are traversed checking for values in *temp_back*. If all nodes have values in *temp_back*, then the loop has been repeated. In *Poss_Head*, the value of *temp_back* is replicated as the back edge. If any node has no value in *temp_back*, the potential loop body has not been repeated.

When a loop is added to H, nodes require removal. If a back edge is added for case 1, then only node n needs to be removed. When a back edge is added for case 2, the nodes and their children that were created to duplicate the loop body must be removed. When traversing H from the parent of *Poss_Tail* to *Poss_Head*, the variable *prev_traverse* is set to the previously checked node. If a back edge is added, node *prev_traverse* and its children are removed by the `Remove_Nodes()` function. Entries may remain in *CCS_Q* for children of the removed node. When a node is removed from H, the queue *CCS_Q* is scanned for entries whose parent is the removed node. If any are found, they are removed from *CCS_Q* by the `RemoveQ()` function.

```
Crt_H()
    Initialize queues $Msg\_Q_0, \ldots, Msg\_Q_{N-1}$, $CCS\_Q$ to empty
    Initialize array $counter[0] \ldots counter[N-1]$ to 0
    Create root node $r$
    $r.CCS = CCS_0$
    Determine_SUCC($r, CCS_0, Msg\_Q_0, \ldots, Msg\_Q_{N-1}, CCS\_Q$)
    while $CCS\_Q$ is not empty
        $item$ = behead($CCS\_Q$)        /* format of item is <node.list,$Q_0,\ldots,Q_{N-1}$, counter> */
        $Parent = item.node$
        $LL = item.list$
        $Msg\_Q_0,\ldots,Msg\_Q_{N-1} = item.Q_0,\ldots,item.Q_{N-1}$
        $counter = item.counter$
        for each <CCS, FGnode> entry in $LL$
            Create a node $n$ in $H$
            $n.CCS = CCS$
            $n.FGnode = FGnode$
            AddEdge($Parent$, $n$)
            Determine_SUCC($n$,$CCS$, $Msg\_Q_0,\ldots, Msg\_Q_{N-1}$, $counter$, $CCS\_Q$)
        end for
    end while
end algorithm


Determine_SUCC($n$, $CCS$, $Msg\_Q_0,\ldots, Msg\_Q_{N-1}$, $counter$, $CCS\_Q$)
    $Msg\_Q'_0,\ldots,Msg\_Q'_{N-1} = Msg\_Q_0,\ldots, Msg\_Q_{N-1}$
    $counter' = counter$
    $Loop$ = false
    if ($n$ = root node)
        SUCC($CCS$) = Generate_SUCC($n$)      /* <CSS,FGnode> is entry in SUCC(CCS) */
        if (SUCC($CCS$) $\neq$ NULL)
            Add <$n$.SUCC($CCS$),$Msg\_Q'_0,\ldots, Msg\_Q'_{N-1}$,$counter'$> to the tail of $CCS\_Q$
        end if
    else
        for $i = 0$ to $N-1$
            if ($v_i$ of $CCS$ = async_recv($j$))
                if ($Msg\_Q'_i$ has entry <$counter$, $j$ >)
                    /* $v_i$ is a ready receive */
                    $item$ = behead first <$counter$, $j$ > entry in $Msg\_Q'_i$
                    append $item.counter$ to $v_i$ in $CCS$        /* item.counter:async_recv($j$) */
                end if
            end if
        end for
        for $i = 0$ to $N-1$
            if ($v_i$ of $CCS$ = async_send($j$))
                $counter[j]'$++
                Add <$counter[j]'$, $i$ > to $Msg\_Q'_j$
                Append $counter[j]'$ to $v_i$ in $CCS$
            end if
        end for
        $Loop$ = Check_Loop($n$)                               /* Changes for loop start here */
        if ($Loop$ = false)
```

```
            SUCC($CCS$) = Generate_SUCC($n$)
            if (SUCC($CCS$) $\neq$ NULL)
                Add <n,SUCC(CCS),$Msg\_Q'_0,\ldots, Msg\_Q'_{N-1}$,counter'>
                to the tail of CCS_Q
            end if
        end if                                          /* Changes for loop stop here */
    end if
end function


Check_Loop($n$)
    Found_Match = false
    Poss_Head = n.parent[0]        /* Check if an ancestor of n represent the same state of n */
    while (Poss_Head $\neq$ ROOT) AND (Found_Match = false)
        Found_Match = Check_Dup($n$, Poss_Head)
        if (Found_Match = false)
            Poss_Head = Poss_Head.parent[0]
        endif
    endwhile
    if (Found_Match = true)   /* Poss_Head represents the same state as n. Does loop exist? */
        Poss_Tail = n.parent[0]
        if (Poss_Head = Poss_Tail)                                /* Case 1 */
            Add_BackEdge(Poss_Head, Poss_Tail)
            RemoveNodes($n$)
            return(true)
        else                                                      /* Case 2 */
            traverse_node = Poss_Tail.parent[0]
            while (traverse_node $\neq$ Poss_Head) AND (traverse_node.temp_back $\neq$ NULL)
                prev_traverse = traverse_node
                traverse_node = traverse_node.parent[0]
            end while
            if (traverse_node $\neq$ Poss_Head) OR                 /* a potential back edge */
                ((traverse_node = Poss_Head) AND (Poss_Head.temp_back = NULL))
                Poss_Tail.temp_back = Poss_Head
                return(false)
            else                                        /* a loop exists, add the back edge */
                Add_BackEdge(Poss_Head.temp_back, Poss_Head)
                RemoveNodes(prev_traverse)
                return(true)
            endif
        endif
    endif
end function
```

```
Check_Dup(node1, node2)
    CCS1 = node1.CCS
    CCS2 = node2.CCS
    i = 0
    Equal = true
    while (i < N) AND (Equal = true)
        if ($v_i \in$ CCS1 = $v_i \in$ CCS2)   /* Do not compare counter that may be appended to $v_i$ */
            if ( node1.FGnode[i] $\neq$ node2.FGnode[i])
                Equal = false
            end if
        else
            Equal = false
        endif
        i++
    end while
    return(Equal)
end function
```

```
Remove_Nodes(n)
    index = 0
    while (n.Kids[index] $\neq$ NULL)
        Remove_Nodes(n.Kids[index])
        index++
    end while
    Delete n
    RemoveQ(n)
end function
```

```
RemoveQ(CCS)
    item = head of CCS_Q
    while (item $\neq$ NULL)
        if item.parent = CCS
            Remove item from CCS_Q
        end if
        item = next entry in CCS_Q
    end while
end function
```

```
AddEdge(parent, n)
    i = 0
    while (parent.Kids[i] ≠ NULL)
        i++
    end while
    parent.Kids[i] = n
    n.Parents[0] = parent
end function
```

```
Add_BackEdge(n, parent)
    i = 0
    while (parent.Kids[i] ≠ NULL)
        i++
    end while
    parent.Kids[i] = n
    n.Parents[1] = parent
end function
```

The addition of back edges to H represents the repeated execution of a portion of the distributed system's execution. When following a possible execution path of H and a back edge occurs in the path, this back edge represents an iteration of the loop associated with the back edge. If, when considering only the forward edges of H,

- nodes a and b are in an execution path in H,
- nodes a and b are both nodes of the same loop, and
- node a is an ancestor of node b,

then, when considering forward and back edges,

- b is an ancestor of a when b occurs in iteration i of the loop and a occurs in an iteration greater than i, and
- a is an ancestor of b when a occurs in iteration i and b occurs in iteration i or greater.

Referring back to figure 7.7, node (2:async_send(1), 1:async_recv(0)) is an ancestor of node (3:async_send(1), 2:async_recv(0)), and node (3:async_send(1), 2:async_recv(0)) is an ancestor of (2:async_send(1), 1:async_recv(0)). The first ancestor relationship is inherent, but the second is only possible with the addition of the back edge. The second relationship is true only when node (3:async_send(1), 2:async_recv(0)) occurs in iteration i and node (2:async_send(1), 1:async_recv(0)) occurs in an iteration greater than i.

If node a is an ancestor of node b, then b is a descendant of a. The children of a are the descendants of a whose path length from a is equal to one. This path can be a forward or back edge. If b is a child of a, then a is the parent of b.

Property 6.8 states that tree H represents all the partial orders of the distributed system. Without back edges in H, the number of partial orders is finite. If H has back edges, the partial orders are known but the number of partial orders is potentially infinite. A bound is not known for the number of times a loop can be iterated. Each path from the root to a leaf node that includes a back edge represents a group of partial orders that have a repeating pattern. Graph H continues to represent all the partial orders.

7.3 POG

In chapter 6, the input to the algorithm `Crt_POG()` is the tree H and the output is the *POG*. With the possibility of back edges in H, the *POG* can also have back edges. Only the function `EQUIV()` of algorithm `Crt_POG()` is affected by the addition of back edges in H. Properties 6.9 through 6.13 and Lemmas 6.2 and 6.3 correspond to the *POG*. We demonstrate that these properties and lemmas are maintained with the addition of back edges.

Function `EQUIV()` serves the same function as described in chapter 6. However, the inclusion of back edges requires additional tests to determine equivalency of H nodes. Suppose CCS and CCS' are found to have equivalent communication commands, and the nodes that represent CCS and CCS' are n and n'. Function `EQUIV()` must check whether

1. n or n' is pointed to by a back edge, or
2. n or n' has a back edge.

Function `EQUIV()` calls function **`Check_Back()`** to determine if either (1) or (2) are true. If neither (1) nor (2) occurs, n and n' are equivalent. Both cases require further tests to determine equivalence.

In case (1), if only one of the nodes is referenced by a back edge, then n and n' are not equivalent. When nodes n and n' are each pointed to by a back edge, both node n are entry points of loops in H. The next test determines whether the loop associated with node n is *equivalent* to the loop associated with node n'. The recursive function **`TreeCmp()`** of algorithm **`Crt_POG()`** determines the equivalence of the two loops.

Node *back* is the node that has a back edge to node n, and node *back'* is the node that has a back edge to node n'. Nodes n and *back* define a subtree. Node n is the root node, and the nodes that are descendants of n but not the descendants of *back* comprise the nodes of the subtree. The variable *subtree* is the subtree defined by nodes n and *back*. Nodes n' and *back'* also define a subtree, *subtree'*. The two subtrees are traversed in lock step, starting at the root node, in depth first order. The current node of *subtree*, c, is compared against the current node of *subtree'*, c'. If

1. the CCS of node c is equivalent to the CCS of node c', and
2. the number of children of c is equal to the number of children of c'

then the traversal of the subtree continues. If either condition is false, the loops are not equivalent and the traversal stops. If both subtrees are completely traversed without falsifying either condition, then the loops are equivalent. If the loops are equivalent, then nodes n and n' are represented by a single node in the POG. The nodes of the equivalent loops, that are not the entry and exit points of the loop, will be united by the original `EQUIV()` algorithm.

If case 2 is found to be true, then the following two tests are required to determine the equivalence of n and n':

1. both n and n' have a back edge, and
2. the H node pointed to by the back edge of n is equivalent to the H node pointed to by the back edge of n'.

Function `Check_Node()` is called by `EQUIV()` to determine if these two tests are true. If both tests are passed, both n and n' will be represented by a single node in the POG.

The equivalent H nodes described in test 2 will already be represented by one node of the POG as a result of the previous case. A single back edge will be added from the

POG node that represents n and n' to the *POG* node representing the equivalent H nodes pointed to by the back edges of n and n'.

One additional field is added to the H nodes to transform H into the *POG*. The field *POGnode* is added to point to the *POG* node representing this H node. More than one H node may have the same value of *POGnode* since one *POG* node represents equivalent H nodes. The *POG* nodes also require additional fields that are replicated from the H nodes:

- *Kids[MAXEDGES]*
- *Kid_type[MAXEDGES]*
- *Parents[MAXEDGES]*

These *POG* node fields are functionally equivalent to their H node counterparts. Field *Kids[MAXEDGES]* is an array of pointers to the children of the *POG* node. Each entry represents either a forward or back edge. Field *Kid_type[MAXEDGES]* is an array of integers indicating the type of edge for each entry. Forward edge have a zero entry, while back edges have a one entry. Pointers to the parents of the *POG* node are maintained in the array *Parents[MAXEDGES]*.

Algorithm `Crt_POG()` and function `EQUIV()` are shown with required back edge additions. Supportive functions are also shown.

```
Crt_POG()
    Initialize queue VisitNodes to empty
    Create root node $S$ (labeled root)
    Add <$S$, KidsOf(root node of $H$)> as the first entry in the queue VisitNodes
    while ( VisitNodes not empty )
        item = behead( VisitNodes)                /* format of item is <node_ptr, node_set> */
        POG_ptr = item.node_ptr
        node_set = item.node_set                  /* node_set= $\{node_1, \ldots, node_m\}$, $m \geq 1$ */
        while ((EQUIV_set = EQUIV(node_set) $\neq \emptyset$)
            Commos = the sends and ready receives the CCSs of EQUIV_set
            Create POG node N and label with Commos
            AddEdge(POG_ptr, N)
            Node_SuccSet = $\emptyset$
            for each node of EQUIV_set
                Node_SuccSet = Node_SuccSet $\cup$ KidsOf(node)
            end for
            Add the entry <N, Node_SuccSet> to the tail of VisitNodes
            node_set = node_set - EQUIV_set
        end while
        for each nodeofnode_set
            if ((Commos = sends and ready receives of the CCS of node) $\neq$ NULL)
                Create POG node N and label with Commos
                AddEdge(POG_ptr, N )
                Add the entry <N, KidsOf(node)> to the tail of VisitNodes
            else
                Create POG node N and label as END node
                AddEdge(POG_ptr, N )
            endif
        endfor
    end while
end algorithm
```

```
EQUIV(node_set)
    node_set' = node_set
    EQUIV_found = false
    while (node_set' $\neq \emptyset$) AND (EQUIV_found = false)
        Node_1 = Select(node_set')
        EQUIV_set = {Node_1}
        Commos = the sends and ready receives of Node_1.CCS
        FGnode = Node_1.FGnode
        Back_Found1 = Check_Back(Node_1)
        local_set = node_set'
        while (local_set $\neq \emptyset$)
            Node_2 = Select(local_set)
            Commos_2 = the sends and ready receives of Node_2.CCS
            FGnode2 = Node_2.FGnode
            if (Commos = Commos_2) AND (FGnode = FGnode2)
```

```
                    $Back\_Found2$ = Check_Back($Node\_2$)
                    if ($Back\_Found1$ = $Back\_Found2$)
                        case $Back\_Found1$
                        0:
                            $EQUIV\_found$ = true
                            Add $Node\_2$ to $EQUIV\_set$
                        1:
                            if (TreeCmp($Node\_1$, $Node\_2$, $Node\_1.parent[1]$, $Node\_2.parent[1]$))
                                $EQUIV\_found$ = true
                                Add $Node\_2$ to $EQUIV\_set$
                            end if
                        2:
                            if (Check_Node($Node\_1$, $Node\_2$))
                                $EQUIV\_found$ = true
                                Add $Node\_2$ to $EQUIV\_set$
                            end if
                        end case
                    end if
                end if
            end while
        end while
        if ($EQUIV\_found$=true)
            return($EQUIV\_set$)
        else
            return($\emptyset$)
    end function
```

```
    Check_Back($Node$)
        /* Check 1: Is Node pointed to by a back edge? */
        if ($Node.parent[1] \neq$ NULL)
            return(1)
        /* Check 2: Does Node have a back edge? */
        $i = 0$
        while ($Node.Kids[i] \neq$ NULL)
            if ($Node.Kid\_Type[i] = 1$)
                return(2)
            end if
            $i$++
        end while
        /* Neither check 1 nor check 2 is true */
        return(0)
    end function
```

```
Check_Node(Node_1, Node_2)
    $i = 0$
    while (Node_1.Kid_Type[i] = 0)                    /* find back edge in Node_1 */
        i++
    end while
    BackNode1 = Node_1.Kid[i]
    $i = 0$
    while (Node_2.Kid_Type[i] = 0)                    /* find back edge in Node_2 */
        i++
    end while
    BackNode2 = Node_2.Kids[i]
    if (BackNode1.POGNode = BackNode2.POGNode)
        return(true)
    else
        return(false)
    end if
end function


TreeCmp(Root1, Root2, Term1, Term2)
    Kids1 = Root1.Kids
    Kids2 = Root2.Kids
    if ((Kids1 has no entries AND Kids2 has no entries) AND (Root1.CCS = Root2.CCS))
        return(true)
    end if
    if ((number of entries in Kids1 $\neq$ number of entries in Kids2)
        OR (Root1.CCS $\neq$ Root2.CCS))
            return(false)
    end if
    $i = 0$
    while (Kids1[i] $\neq$ NULL)
        if ((Kids1[i] $\neq$ Term1) AND (Kids2[i] $\neq$ Term2))
            if (TreeCmp(Kids1[i], Kids2[i], Term1, Term2) = false)
                return(false)
            end if
        else
            if (((Kids1[i] = Term1) AND (Kids2[i] $\neq$ Term2))) OR
                ((Kids1[i] $\neq$ Term1) AND (Kids2[i] = Term2)))
                return(false)
            end if
        end if
        i++
    end while
    return(true)
end function
```

```
KidsOf(node)
    kidset = NULL
    i = 0
    while (node.Kids[i] ≠ NULL)
        if node.Kid_type[i] = 0
            Add node.Kids[i] to kidset
        end if
        i++
    end while
    return(kidset)
end function
```

Properties 6.9, 6.10, 6.11, and 6.12 are derived directly from the properties of *H* and are affected by back edges as described in section 7.2. Lemmas 6.2 and 6.3 and property 6.13 remain true with the addition of loops in *H* and the *POG*. Modification of algorithm `Crt_POG()` is limited to additional checks for equivalency of nodes of *H*. The construction of the *POG* continues to preserve the causal and concurrent relationships in *H*. Property 6.13 states that each path from the root node to a leaf node of the *POG* represents a unique partial order. If there exists a path from the root node to leaf node n that contains a loop, then a different path exists from the root to n when the nodes of the loop are repeated.

7.4 LCP and LCP' events

The last modifications pertain to the algorithms **Bound_Assert()** and **Find_LCPs()** that determine the LCP and LCP' events. Lemmas 6.4 and 6.5 and properties 6.14 and 6.15 correspond to the identification of the LCP and LCP' events. These lemmas and properties are not affected by the possible occurrence of back edges in the POG. Two modifications are required for **Bound_Assert()**. The first modification stops searching a path for the last LCP' event when the ASSERT node occurs in the body of a loop. Without back edges in flow graph FG_i, the search stopped when either a RECEIVE node or the root node was encountered. With back edges the search should also stop if the ASSERT node itself is encountered. When **Bound_Assert()** searches for a RECEIVE node in the flow graph of figure 7.12 two paths are searched. One is the path including only the FOR node and the RECEIVE node. The search stops at the RECEIVE node. The other path starts at the FOR node. proceeds to the END_FOR node by following the back edge. The next node in the path is the ASSERT node. The search terminates since a receive does not exist on the path from the ASSERT node back to itself. If a RECEIVE node exists between the END_FOR node and the ASSERT node, as shown in figure 7.13, the RECEIVE is a last LCP' event and is added to the linked list *Local_LCPs*. In this case, the search succeeds when the RECEIVE node is encountered.

The second modification is needed when a loop occurs in the path being searched, but the ASSERT node is not part of the loop body. Without modification, the nodes of a loop will be followed infinitely if the loop occurs prior to the ASSERT node and a RECEIVE node is not found. The flow graph in 7.14 demonstrates the problem. The path ASSIGN, FOR, END_FOR, SEND, FOR, END_FOR, SEND, ... is repeatedly traveled unless a modification is made. When searching a path for a RECEIVE node, each node is flagged as *visited* when it is encountered. Before following a parent of a node in a search path, the visited flag of that node is tested. If it has not been set, then this path is searched. If it has been previously visited. this path is not searched.

For completeness, we repeat algorithm **Bound_Assert()** with modifications.

```
Bound_Assert()                                   /* input: $FG_i$ and assert_node */
    current_list = the parent nodes of assert_node
    NextBranch = NULL
    Local_LCPs = NULL
    crnt_node = first entry in current_list
    Remove crnt_node from current_list
    receive_found = false
    do
        while ((receive_found=false) AND (crnt_node $\neq$ root node of $FG_i$) AND
            (crnt_node $\neq$ assert_node) AND (crnt_node has not been visited))
            if (current_list $\neq$ NULL)
                Push current_list on the stack NextBranch
            endif
            Mark crnt_node as visited
            if crnt_node = receive
                Add crnt_node to Local_LCPs
                receive_found = true
            else
                current_list = parent nodes of crnt_node
                crnt_node = first entry in current_list
                Remove crnt_node from current_list
            endif
        endwhile
        if (NextBranch $\neq$ NULL)
            receive_found = false
            current_list = Pop(NextBranch)
            crnt_node = first entry in current_list
            Remove crnt_node from current_list
        endif
    while (NextBranch $\neq$ NULL)
end algorithm
```

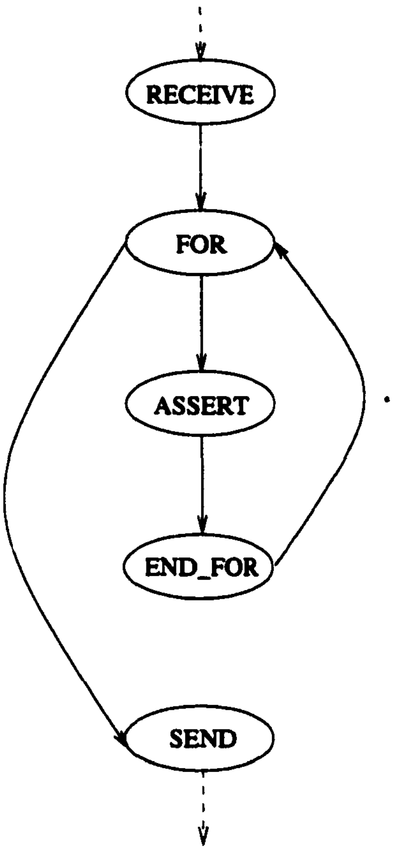


Figure 7.12: Assert in the loop body

Algorithm `Bound_Assert()` constructs a linked list, *Local_LCPs*, that are the last *LCP* events of the assert. This list is used by algorithm `Find_LCPs()` to determine the *LCP* and *LCP'* events of the *POG*. The search for *LCP* and *LCP'* requires `Find_LCPs()` to visit the ancestors of each *POG* node represented by an *LCP* event in *Local_LCPs*. Changes are necessary to `Find_LCPs()` to contend with back edges encountered during the search. Back edges in the *POG* define additional causal relationships as demonstrated by the portion of the *POG* shown in 7.15. Without considering the back edge, the causal relationships are

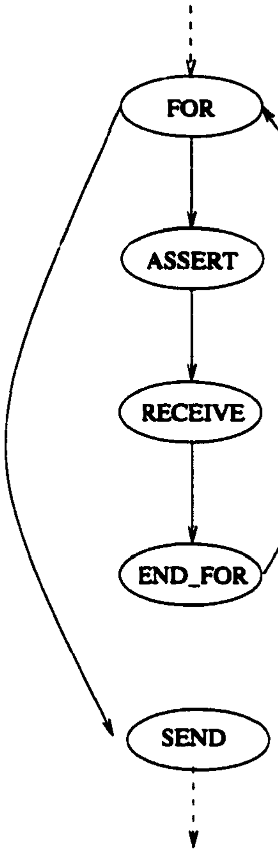


Figure 7.13: Assert and receive in the loop body

$1 : 0S1 \rightarrow 1 : 1R0 \rightarrow 1 : 2S0 \rightarrow 1 : 0R2$. The causal relationships $1 : 0R2 \rightarrow 1 : 2S0 \rightarrow 1 : 1R0 \rightarrow 1 : 0S1$ exist with the back edge. When determining the *LCP* and *LCP'* events, all casual relationships, including those derived from back edges, must be considered.

A node with a back edge pointing to it has two parents. One parent is the result of a forward edge, and the other parent is the result of a back edge. In the original version of `Find_LCPs()`, only parent nodes which result from forward edges are searched. To consider all the causal relationships in the *POG*, paths that include parent nodes that are the result of back edges are also searched.

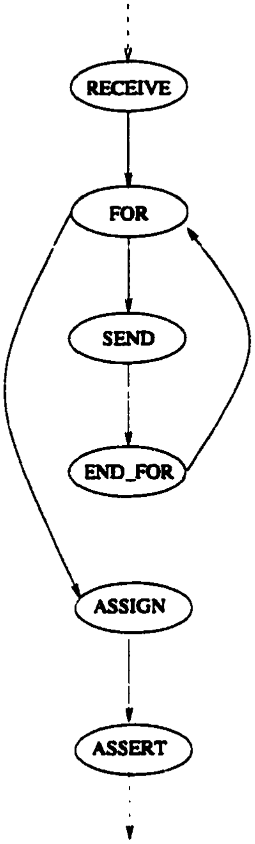


Figure 7.14: Assert not in the loop body

In the *POG* shown in figure 7.15. the assert occurs in P_1. and the receive of node *last* is the last *LCP* event of P_1. The search for *LCP* and *LCP'* events starts at node *last*. The send of node n, 1:0S1. is found to be an *LCP'* event. Node n has two parents, one resulting from a forward edge and one resulting from a back edge. At this point the search branches into two paths. The path that includes the parent of node n resulting from a forward edge is searched by the original `Find_LCPs()`. The path that includes node n' and node n'' should also be searched by `Find_LCPs()` since these nodes are ancestors of node n. The receive of node n'' is an *LCP* event and the send of node n' is an *LCP'* event.

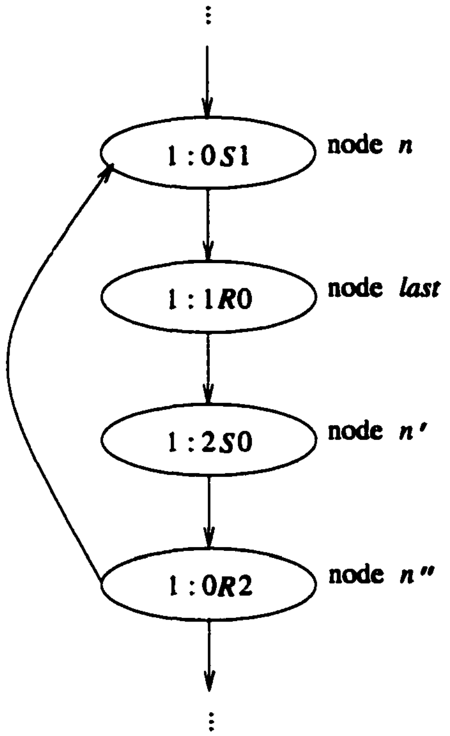


Figure 7.15: *POG* with a back edge

Notice in the *POG* of figure 7.15 that when the path follows the back edge parent of node n, the path $last.n.n''.n'$ can repeat indefinitely. When a back edge is encountered, the back edge must be followed to consider all causal relationships. By following the back edge once, all additional causal relationships defined by this back edge are considered. Additional variables are required in algorithm `Find_LCPs()` to follow paths that include parent nodes resulting from back edges and to not visit a parent that is the result of a back edge more than once in the same search path. A node is placed in the set *VisitOnce*if the node is a parent node resulting from a back edge, and the node is visited by the current search. Since the search can branch into two different paths, the state of the search prior to the branch is saved. The branch resulting from a forward edge is visited first. When the search of this branch has been completed, the other branch is searched by restoring the saved state and continuing the search at the branch point. An entry in the queue *StateQ* is the state of a search. The format of an entry in *StateQ* is <*POGNode*, *RwoSQ*, *SendQ*, *RecvQ*, *FoundProcs*, *Sends*, *Rec_wo_Sends*, *VisitOnce*>. The variable *POGNode* is the parent node resulting from the back edge. The remaining items are the values of variables before the branch. Algorithm `Find_LCPs()` is repeated with the appropriate modifications.

```
Find_LCPs()                                   /* Input: Local_LCPs Output: SendQ, RecvQ */

    StateQ = NULL
    for each entry in Local_LCPs where the event entry is $c : iRj$
        for each POG node that contains $c : iRj$
            Lastnode = POG node that contains $c : iRj$
            POGnode = Startnode
            FoundProcs = Sends = VisitOnce = $\emptyset$
            Rec_wo_Sends = $\{i\}$
            RwoSQ = NULL
            Insert <c, i, j, POGnode> in RwoSQ
            POGnode = ParentOf(POGnode)
            while (POGnode $\neq$ root node) AND (FoundProcs $\neq$ ({ 0, ... ,N-1 } - i))
                while (POGnode $\neq$ root node) AND (FoundProcs $\neq$ ({ 0, ... ,N-1 } - i))
                    for each receive, $c : jRk$, in POGnode
                        if (($j \in$ Sends) OR ( $j = i$)) AND ($k \notin$ FoundProcs)
                            Insert <c, j, k, POGnode> in RwoSQ
                            Rec_wo_Sends = Rec_wo_Sends + j
                        endif
                    endfor
                    for each send, $c : jSk$, in POGnode
                        if ($k \in$ Rec_wo_Sends) AND
                            ((Recv_POGnode = SearchQ(c, k, j)) $\neq$ NULL)
                            if (RwoSQ does not have an entry with k as the receiver)
                                Rec_wo_Sends = Rec_wo_Sends - k
                            endif
                            Sends = Sends + j
                            FoundProcs = FoundProcs + j
                            Insert <c, j, POGnode> in SendQ
                    Insert <c, k, Recv_POGnode> in RecvQ
                end if
            end for
            POGnode = ParentOf(POGnode)
        end while
        if (StateQ ≠ NULL)
            item = behead(StateQ)
            POGnode = item.POGnode
            RwoSQ = item.RwoSQ
            SendQ = item.SendQ
            RecvQ = item.RecvQ
            FoundProcs = item.FoundProcs
            Sends = item.Sends
            Rec_wo_Sends = item.Rec_wo_Sends
            VisitOnce = item.VisitOnce
        end if
    end while
  end for
end for
end algorithm
```

```
ParentOf(POGnode)
    if (POGnode.Parent[1] ≠ NULL)                    /* if POGnode has two parents */
        AND (POGnode.Parent[1] not in VisitOnce)
            Add entry
            <POGnode.Parent[1], RwoSQ, SendQ, RecvQ, FoundProcs,
                Sends, Rec_wo_Sends, VisitOnce>
            to StateQ
    end if
    return(POGnode.Parent[0])
end function
```

The changes to algorithm `Find_LCPs()` to facilitate searching paths including back edges for LCP and LCP' events do not affect lemmas 6.4 and 6.5 and properties 6.14 and 6.15. The entries in *SendQ* are the LCP events, and the entries in *RecvQ* are the LCP' events. No alterations to the method of adding entries into these queues results from the changes to `Find_LCPs()`. We conclude that our technique for identifying LCP and LCP' events remains valid.

The next chapter analyzes the distributed programs of chapter 2. The resulting POG is shown for each program, and the LCP and LCP' are determined from the POG.

Chapter 8

Static Analysis of Distributed Programs

We presented five distributed programs in Chapter 2. In this chapter, we apply the algorithms of chapters 6 and 7 to determine the LCP and LCP' events for each distributed program.

8.1 Set Partition

SETPART, the set partition program, is reproduced from section 2.1 with the addition of an assert statement A_1 in process P_1.

```
P0::
 1  mx = max($S$)
 2  async_send(1, mx)
 3  $S = S$ - {mx}
 4  async_recv(1, x)
 5  $S = S \cup$ {x}
 6  mx = max($S$)
 7  while (mx > x)
 8      async_send(1, mx)
 9      $S = S$ - {mx}
10      async_recv(1, x)
11      $S = S \cup$ {x}
12      mx = max($S$)
13  endwhile
```

```
P1::
14  while(true)
15      async_recv(0, y)
16      $T = T \cup$ {y}
17      mn = min($T$)
A1  assert($y = \max(S) \geq mn > x \wedge$
        $|S| = |S_0| \wedge S \cap T = y$ )
18      async_send(0, mn)
19      $T = T$ - {mn}
20  endwhile
```

An assert statement in either process is adequate for expressing expected system execution behavior. Placing the causal assert statement A_1 between lines 17 and 18 is useful for detecting incorrect execution and for locating errors in both P_0 and P_1. Assert statement A_1 is evaluated on each exchange.

A false evaluation of A_1 indicates erroneous execution of the program. SETPART's error is identified by the assert's falsifying clause. If y is not equal to max(S); P_0 did not send the correct value. If $\max(S) \not\geq mn$; processing should have stopped on the last exchange, and a likely error is P_0's exchange loop condition. If $mn \not> x$; either a value other than the minimum of T was chosen, or P_0 has erroneously altered the variable x since the last exchange. If the new size of S has changed, P_0 has not correctly added or removed a value from S since the last exchange. If the intersection of S and T is not equal to y; either S or T has not been correctly updated since the last exchange, and the results of the other clauses help in identifying the incorrect set.

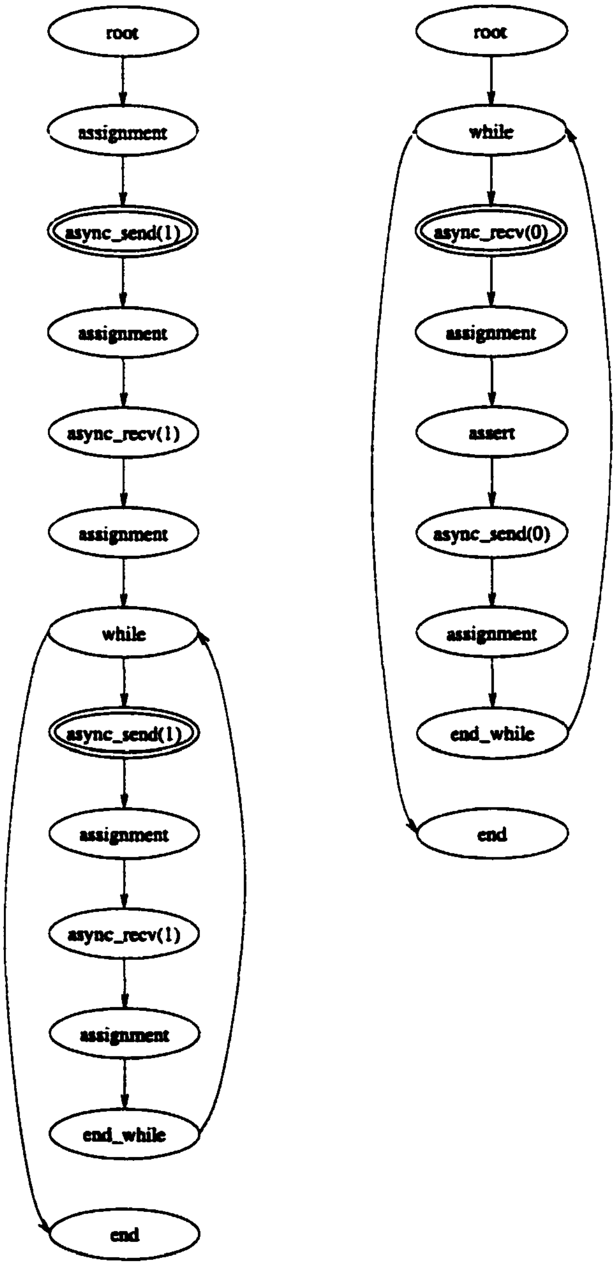
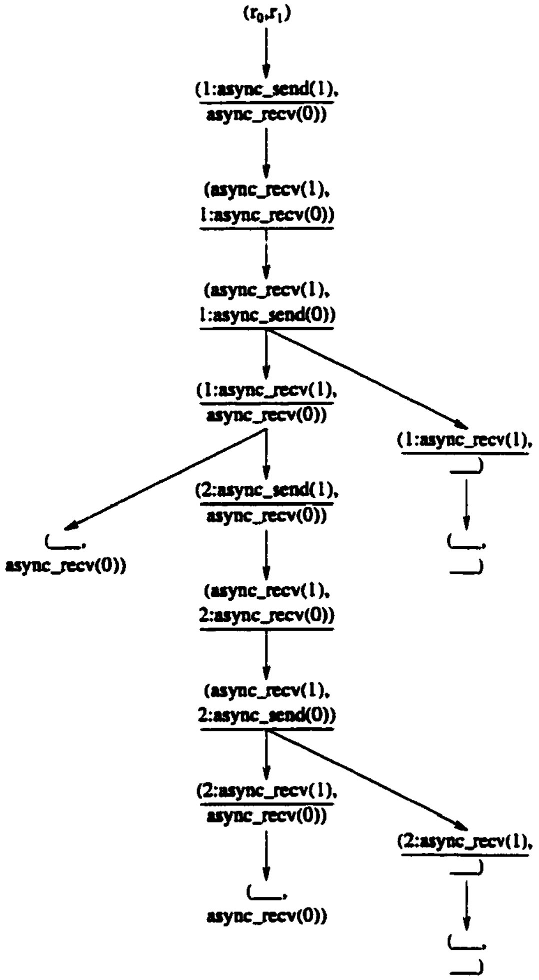

root
assignment
async_send(1)
assignment
async_recv(1)
assignment
while
async_send(1)
assignment
async_recv(1)
assignment
end_while
end
root
while
async_recv(0)
assignment
assert
async_send(0)
assignment
end_while
end

Figure 8.1: Flow Graphs for Set Partition



(r_0,r_1)
(1:async_send(1), async_recv(0))
(async_recv(1), 1:async_recv(0))
(async_recv(1), 1:async_send(0))
(1:async_recv(1), async_recv(0))
(1:async_recv(1), ⊔)
(⊔, async_recv(0))
(2:async_send(1), async_recv(0))
(⊔, ⊔)
(async_recv(1), 2:async_recv(0))
(async_recv(1), 2:async_send(0))
(2:async_recv(1), async_recv(0))
(2:async_recv(1), ⊔)
(⊔, async_recv(0))
(⊔, ⊔)

Figure 8.2: *H* for Set Partition

Suppose the programmer mistypes line 8 by sending x instead of mx to P_1. This mistake is detected by clause $y = \max(S)$ of A_1. The negative evaluation of this clause identifies an erroneous value sent by P_0. Alternatively, suppose P_0's condition to initiate another exchange is incorrectly a $\geq$ instead of a $>$, then line 7 is

7 while (mx $\geq$ x).

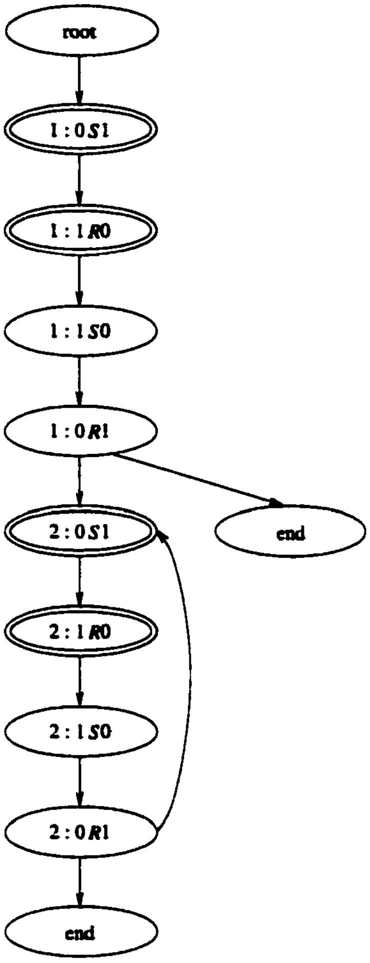


Figure 8.3: *POG* for Set Partition

This error prevents P_0 from detecting the sets are partitioned, and causes SETPART to enter an infinite loop. The clause $mn > x$ of A_1 detects this error the first time an invalid exchange is attempted by P_0 and eliminates the infinite loop problem.

Static analysis is performed by the algorithms of chapter 7 since loops are present in the program. First, algorithm `Create_FG`$_i$`()` constructs the control flow graphs. The resulting flow graphs are shown in figure 8.1.

Graph H is constructed by algorithm `Crt_H()` from the flow graphs. The resulting graph H is shown in figure 8.2. The back edge represents the continuous exchange of data between the two processes until the set is partitioned. The POG is constructed from H and is shown in figure 8.3.

Algorithm `Bound_Assert()` determines the last LCP' event of the assert statement in P_1. Node `async_recv(0)` of FG_1 is returned by `Bound_Assert()`. This node is shown in figure 8.1 with double circles. The event `async_recv(0)` of P_1 is represented by two POG nodes. One node has the entry 1:1R0, and the other node has the entry 2:1R0.

Starting with node 1:1R0 of the POG, we identify the LCP and LCP' events. The LCP' event is 1:1R0, and the LCP event is 1:0S1. For node 2:1R0, the LCP' event is 2:1R0, and the LCP event is 2:0S1. The nodes with double circles in figure 8.3 represent the LCP and LCP' events. Since the assert is in P_1, it is not necessary for P_1 to propagate state information to P_0. Our static analysis allows us to not piggyback messages from P_0 to P_1.

8.2 Mutual Exclusion

Assume a three process distributed system implements mutual exclusion by embedding the circulating token protocol in its distributed application. Additional assumptions are that process P_0 starts the token circulating, process P_1 evaluates the the assert statement A_1, and each process P_i initializes variable in_cs_i to false. Assertion A_1 detects mutual exclusion violation. The distributed application may incorporate message passing, but we only analyze the mutual exclusion code. The messages of the application will not affect our analysis. Below is the portion of the code we analyze.

MUTEX

```
$P_0$::
do
    async_send(1, token)
    async_recv(2, token, $waitsecs$)
    if message received
        if $want\_cs_0$
            $in\_cs_0$=true; $critsec_0$; $want\_cs_0$=false
        endif
        async_send(1, token)
    else /* async_recv timed out */
        do_other$_0$
    endif
enddo

$P_1$::
do
    async_recv(0, token, $waitsecs$)
    if message received
$A_1$  assert($in\_cs_0 = t \wedge in\_cs_1 = t \Rightarrow in\_cs_0 \rightarrow in\_cs_1 \vee in\_cs_1 \rightarrow in\_cs_0$ and
            $in\_cs_1 = t \wedge in\_cs_2 = t \Rightarrow in\_cs_1 \rightarrow in\_cs_2 \vee in\_cs_2 \rightarrow in\_cs_1$) and
            $in\_cs_0 = t \wedge in\_cs_2 = t \Rightarrow in\_cs_0 \rightarrow in\_cs_2 \vee in\_cs_2 \rightarrow in\_cs_0$)
        if $want\_cs_1$
            $in\_cs_1$=true; $critsec_1$; $want\_cs_1$=false
        endif
        async_send(2, token)
    else /* async_recv timed out */
        do_other$_1$
    endif
enddo

$P_2$::
do
    async_recv(1, token, $waitsecs$)
    if message received
        if $want\_cs_2$
            $in\_cs_2$=true; $critsec_2$; $want\_cs_2$=false
        endif
        async_send(0, token)
    else /* async_recv timed out */
        do_other$_2$
    endif
enddo
```

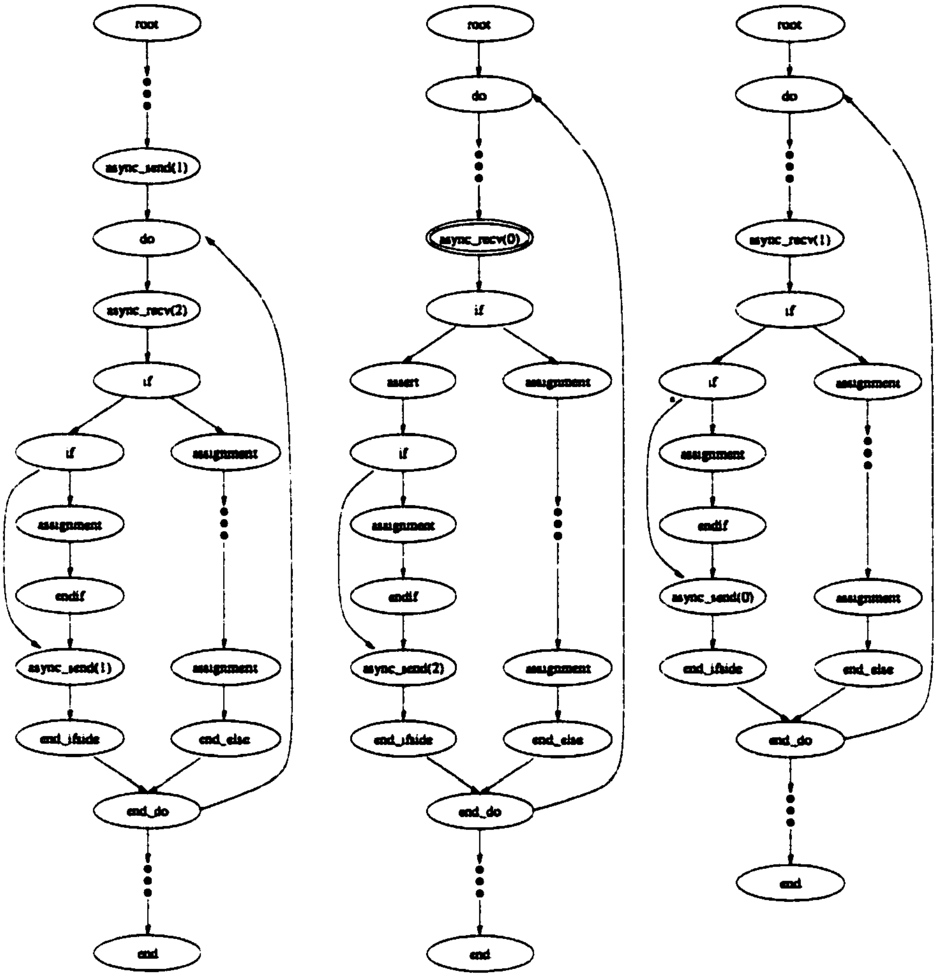

root
async_send(1)
do
async_recv(2)
if
if
assignment
assignment
endif
async_send(1)
assignment
end_ifside
end_else
end_do
end
root
do
async_recv(0)
if
assert
assignment
if
assignment
endif
async_send(2)
assignment
end_ifside
end_else
end_do
end
root
do
async_recv(1)
if
if
assignment
assignment
endif
async_send(0)
assignment
end_ifside
end_else
end_do
end

Figure 8.4: Flow Graphs for Mutual Exclusion

Assume line 26 of P_2 is erroneously omitted, and then suppose the following occurs. Process P_0 passes the token to P_1, and P_1 enters its critical section. Process P_2 wants to enter its critical section and has set *want_cs*$_2$ to true. While P_1 is in its critical section, the async_recv on line 25 times out. The condition of line 27 is true, and P_2 incorrectly enters its critical section while P_1 is in its critical section.

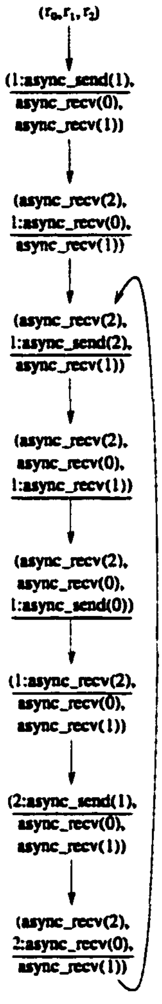

(r0,r1,r2)
(1:async_send(1),
async_recv(0),
async_recv(1))
(async_recv(2),
1:async_recv(0),
async_recv(1))
(async_recv(2),
1:async_send(2),
async_recv(1))
(async_recv(2),
async_recv(0),
1:async_recv(1))
(async_recv(2),
async_recv(0),
1:async_send(0))
(1:async_recv(2),
async_recv(0),
async_recv(1))
(2:async_send(1),
async_recv(0),
async_recv(1))
(async_recv(2),
2:async_recv(0),
async_recv(1))

Figure 8.5: Graph H for Mutual Exclusion

This invalid critical section entry by P_2 is detected by the assert statement A_1 when the token circulates around to P_1. The clause $(in\_cs_1 = t \wedge in\_cs_2 = t \Rightarrow in\_cs_1 \rightarrow in\_cs_2 \vee in\_cs_2 \rightarrow in\_cs_1)$ evaluates to false detecting that P_1 and P_2 entered their critical sections concurrently. The combination of in_cs_i being true and the timestamp of when in_cs_i was last modified conveys the last time P_i entered its critical section. With this information, the assert statement detects any of the processes violating mutual exclusion.

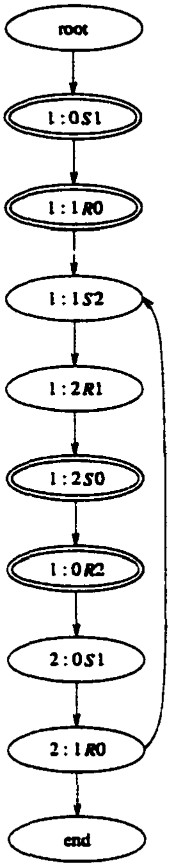


Figure 8.6: *POG* for Mutual Exclusion

The flow graphs for the circulating token protocol are shown in figure 8.4. The `do_other` statements in the source code are represented as a series of assignment nodes in the flow graphs. The H graph generated is shown in figure 8.5, and the *POG* is shown in figure 8.6.

Algorithm `Bound_Assert()` determines the last LCP' event of the assert statement in P_1, node `async_recv(0)` of FG_1. This node is shown in figure 8.4 with double circles. The event `async_recv(0)` of P_1 is represented by two *POG* nodes. One node has the entry 1:1R0, and the other node has the entry 2:1R0.

Starting with node 1:1R0 of the POG, we identify the LCP and LCP' events. The LCP' event is 1:1R0, and the LCP event is 1:0S1. For node 2:1R0, the LCP' events are 2:1R0, 1:0R2, and 1:2R1. The LCP events are 2:0S1, 1:2S0, and 1:1S2. The nodes with double circles in figure 8.6 represent the events that are the LCP and LCP' events. The messages that implement the circulating token are also the messages that piggyback state information for assert evaluation. The distributed program's application messages will not be tagged for piggybacking.

8.3 Bubble Sort

We continue with the distributed bubble sort program from chapter 2 that consists of six processes. The time space diagram for the bubble sort's execution is repeated in figure 8.7. The hashes on P_2's time line represent assertion evaluation. Two asserts in one of the six processes provides a thorough erroneous execution detection method. The assert statements can be in any one of the six processes and provide the same meaningful information. We have arbitrarily selected P_2. Process P_2's source code is shown below with the two assert statements A_{2a} and A_{2b}. The clause $P_i.list \leq P_i.recv\_list$ in the assert statements tests whether every element in $P_i.list$ is less than or equal to all elements of $P_i.recv\_list$, and the clause $P_i.list \geq P_i.recv\_list$ in the assert statements tests whether every element in $P_i.list$ is greater than or equal to all elements $P_i.recv\_list$.

```
P_2::
        integer pid, phase;
        arrays list, recv_list
        pid = 2
        read $q/6$ elements into list
        sort list
        for phase = 0 to 5
            if phase is even
                async_send(3, list)
                async_recv(3, recv_list)
                list = merge_sort(list, recv_list, first)
A_2a        assert( $P_2.list \leq P_2.recv\_list \wedge P_2.recv\_list = P_3.list \wedge$
                    $P_3.list \leq P_3.recv\_list \wedge P_3.recv\_list = P_4.list \wedge$
                    $P_4.list \leq P_4.recv\_list \wedge P_4.recv\_list = P_5.list \wedge$
                    $P_5.list \geq P_5.recv\_list$)
            endif
            if phase is odd && pid != 0 && pid != $N - 1$
                async_send(1, list)
                async_recv(1, recv_list)
                list = merge_sort(list, recv_list, last)
A_2b        assert( $P_2.list \geq P_2.recv\_list \wedge P_2.recv\_list = P_1.list \wedge$
                    $P_1.list \geq P_0.recv\_list \wedge P_1.recv\_list = P_0.list \wedge$
                    $P_0.list \leq P_0.recv\_list$)
            endif
        endfor
```

```
merge_sort(list, recv_list, half)::
        array merge_list
        merge_list = merging of recv_list and list
        sort merge_list
        if half = first
            return first half of elements in merge_list
        else
            return last half of elements in merge_list
        endif
```

The clause $P_i.recv\_list = P_{i+1}.list$, for $i = 2 \ldots 4$, of assert A_{2a} determines whether process P_i received the correct list from its right neighbor P_{i+1}. The clause $P_i.recv\_list = P_{i-1}.list$, for $i = 1 \ldots 2$, of assert A_{2b} determines whether process P_i received the correct

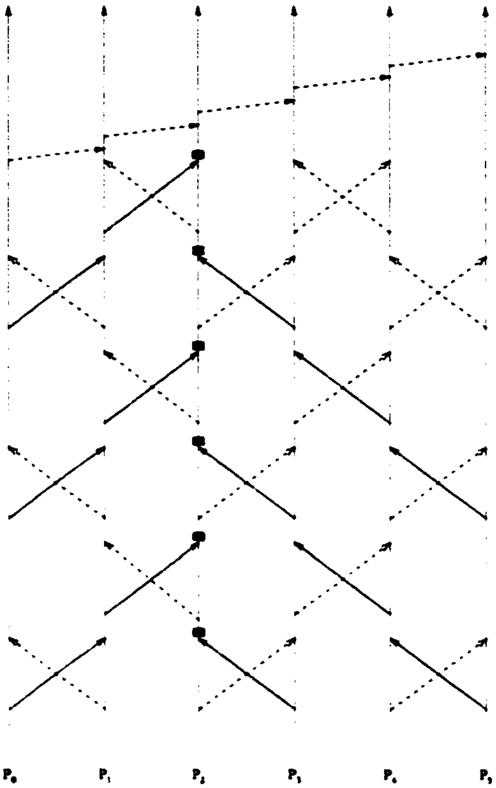


Figure 8.7: Time space diagram for Bubble Sort

list from its left neighbor P_{i-1}. The clauses $P_i.list \leq P_i.recv\_list$ and $P_i.list \geq P_i.recv\_list$ ensure that `merge_sort()` correctly sorted and halved the merged list.

Assume line 9 of P_4 is mistyped. The function merge_sort() is passed *last* instead of *first*. Function merge_sort() sorts and returns the last $q/6$ elements, and these elements are assigned to *list*. The correct execution should have assigned to *list* the first $q/6$ elements of the merged and sorted elements. In the next phase (odd), line 12 of P_4 sends this incorrect *list* to P_3. Assume P_3 is correct. In the following even phase when P_3 sends its supposedly correct list to P_2, the clause $P_4.list \leq P_4.recv\_list$ of assert A_{2a} evaluates to false detecting that P_4 executed incorrectly. This false evaluation singles out the error to P_4's execution of **merge_sort()** in an even phase. In general, errors in merging, sorting and halving

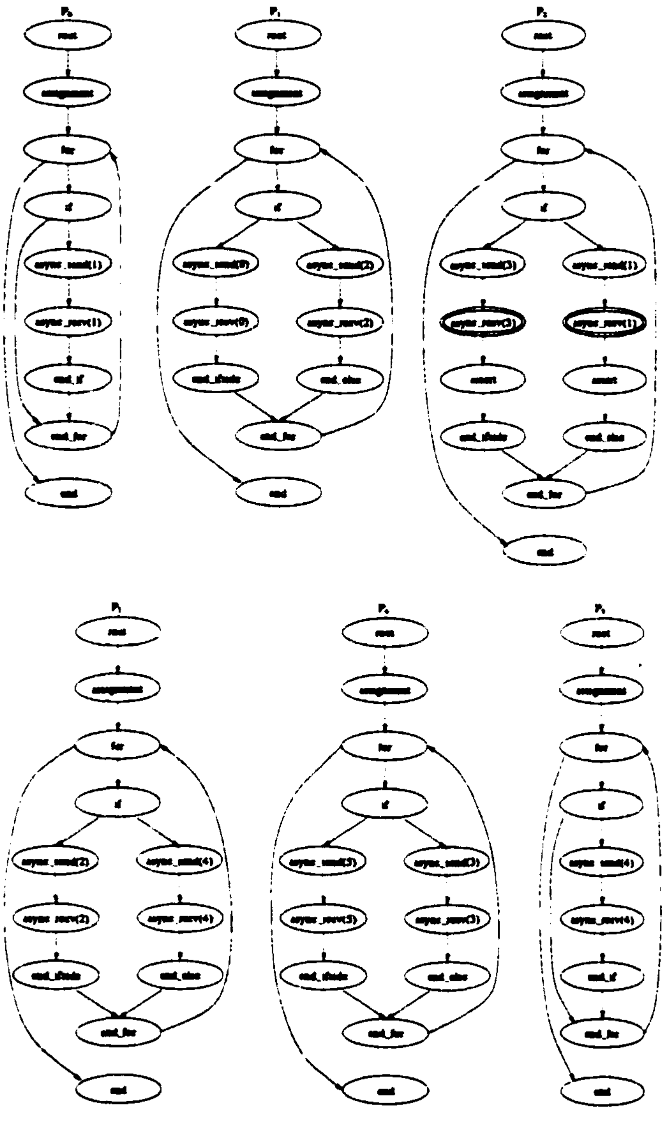


Figure 8.8: Flow Graphs for Bubble Sort

any of the process's list will be detected by the two assert statements. The comparisons $P_2.list \leq P_2.recv\_list$, $P_3.list \leq P_3.recv\_list$ and $P_4.list \leq P_4.recv\_list$ of A_{2a} ensures the correct execution of P_2 and its right neighbors. The comparisons $P_2.list \geq P_2.recv\_list$ and $P_1.list \geq P_0.recv\_list$ of A_{2b} ensures the correct execution of P_2 and its left neighbors.

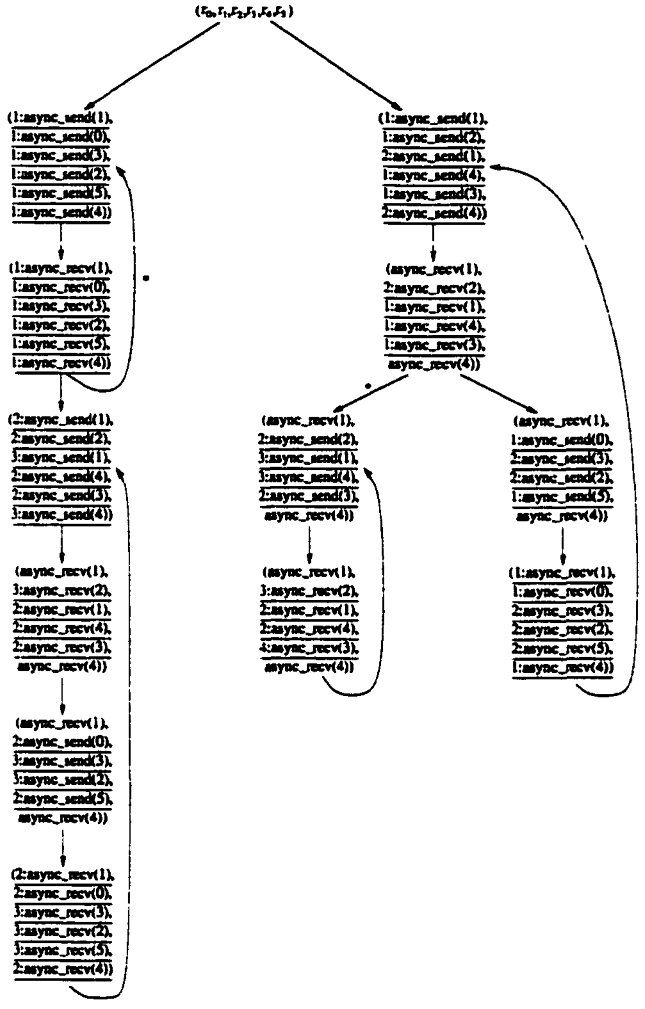

(r0,r1,r2,r3,r4,r5)
(1:async_send(1), 1:async_send(0), 1:async_send(3), 1:async_send(2), 1:async_send(5), 1:async_send(4))
(1:async_recv(1), 1:async_recv(0), 1:async_recv(3), 1:async_recv(2), 1:async_recv(5), 1:async_recv(4))
(2:async_send(1), 2:async_send(2), 3:async_send(1), 2:async_send(4), 2:async_send(3), 3:async_send(4))
(async_recv(1), 3:async_recv(2), 2:async_recv(1), 2:async_recv(4), 2:async_recv(3), async_recv(4))
(async_recv(1), 2:async_send(0), 3:async_send(3), 3:async_send(2), 2:async_send(5), async_recv(4))
(2:async_recv(1), 2:async_recv(0), 3:async_recv(3), 3:async_recv(2), 3:async_recv(5), 2:async_recv(4))
(1:async_send(1), 1:async_send(2), 2:async_send(1), 1:async_send(4), 1:async_send(3), 2:async_send(4))
(async_recv(1), 2:async_recv(2), 1:async_recv(1), 1:async_recv(4), 1:async_recv(3), async_recv(4))
(async_recv(1), 2:async_send(2), 3:async_send(1), 3:async_send(4), 2:async_send(3), async_recv(4))
(async_recv(1), 3:async_recv(2), 2:async_recv(1), 2:async_recv(4), 4:async_recv(3), async_recv(4))
(async_recv(1), 1:async_send(0), 2:async_send(3), 2:async_send(2), 1:async_send(5), async_recv(4))
(1:async_recv(1), 1:async_recv(0), 2:async_recv(3), 2:async_recv(2), 2:async_recv(5), 1:async_recv(4))

Figure 8.9: Graph H for Bubble Sort

In an even phase, process P_3 should send *list* to P_2. But consider the case when P_3 mistakenly sends *recv_list* instead of *list*. Clause $P_2.recv\_list = P_3.list$ of A_{2a} evaluates to false and identifies P_2 as sending the incorrect data. Assert A_{2a} ensures that P_2's right neighbors have sent the correct data, and assert A_{2b} ensures that P_2's left neighbors have sent the correct data.

The source code for the other five processes is not shown in this chapter, but the bubble sort algorithm in chapter 2 is sufficiently outlined for our static analysis. The flow graphs are shown in figure 8.8. The resulting *H* graph is shown in figure 8.9. Each process has an if/else branch in execution, and the combinations of different executions creates a large *H* graph. Only those branches that contribute to a path in the *POG* are shown in *H*. An edge with an asterisk denotes an incorrect decision made at the if/else branch of the processes. The resulting *POG* is shown in figure 8.10.

Since we have two assert statements in P_2. algorithm `Bound_Assert()` is called twice to determine the last LCP' events. For assert A_{2a}, the last LCP' event is `async_recv(3)` of FG_2. The event `async_recv(3)` is represented by three *POG* nodes. The *POG* node entries that represent this receive are 1:2R3. 3:2R3. and 2:2R3. For assert A_{2b}, the last LCP' event is `async_recv(1)` of FG_2. The event `async_recv(1)` is also represented by three *POG* nodes. The *POG* node entries that represents this receive are 2:2R1. 1:2R1, and 2:2R1. All six *POG* node representatives of these last LCP' events are underlined in figure 8.10.

For each of the last LCP' events. the LCP and LCP' events are determined by algorithm `Find_LCPs()`. The LCP and LCP' events are underlined in figure 8.11. The messages that piggyback state information are shown in the time space diagram of figure 8.7 as solid directional lines. The LCP and LCP' events for assert A_{2a} are identical to the LCP and LCP' events for assert A_{2b}. In this example, the additional assert statement did not increase the number of messages piggybacking state information.

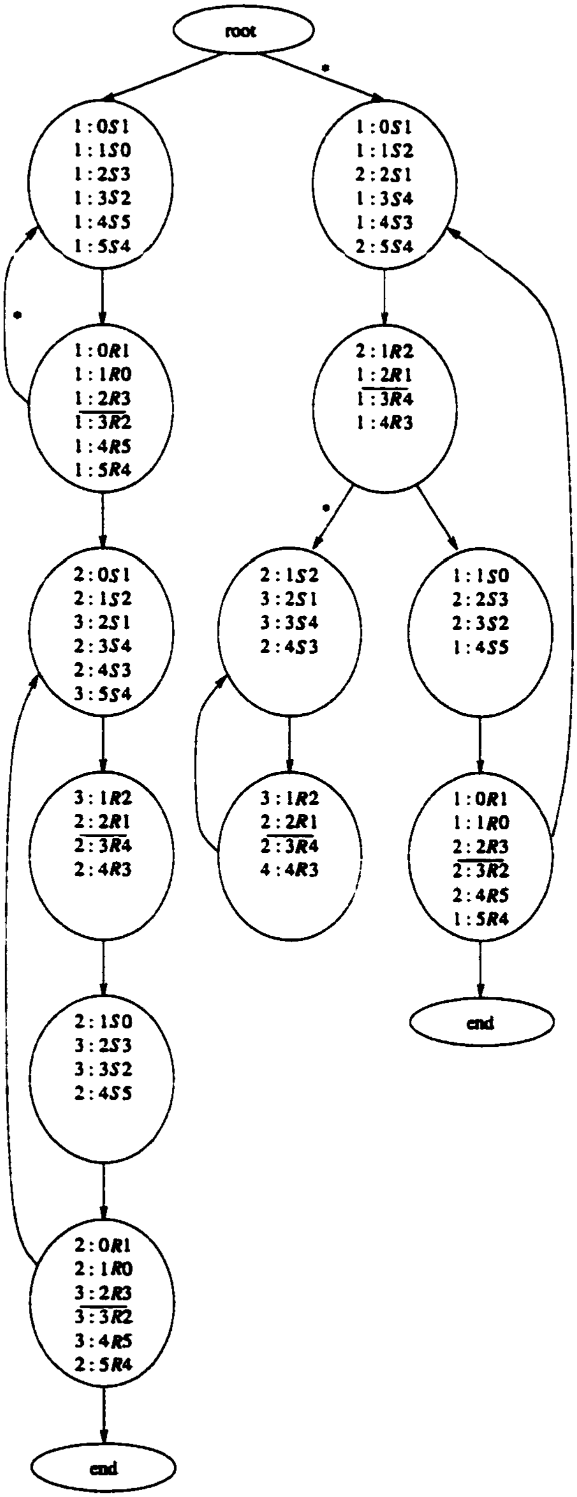

root
1:0S1
1:1S0
1:2S3
1:3S2
1:4S5
1:5S4
1:0S1
1:1S2
2:2S1
1:3S4
1:4S3
2:5S4
1:0R1
1:1R0
1:2R3
1:3R2
1:4R5
1:5R4
2:1R2
1:2R1
1:3R4
1:4R3
2:0S1
2:1S2
3:2S1
2:3S4
2:4S3
3:5S4
2:1S2
3:2S1
3:3S4
2:4S3
1:1S0
2:2S3
2:3S2
1:4S5
3:1R2
2:2R1
2:3R4
2:4R3
3:1R2
2:2R1
2:3R4
4:4R3
1:0R1
1:1R0
2:2R3
2:3R2
2:4R5
1:5R4
end
2:1S0
3:2S3
3:3S2
2:4S5
2:0R1
2:1R0
3:2R3
3:3R2
3:4R5
2:5R4
end

Figure 8.10: *POG* for Bubble Sort

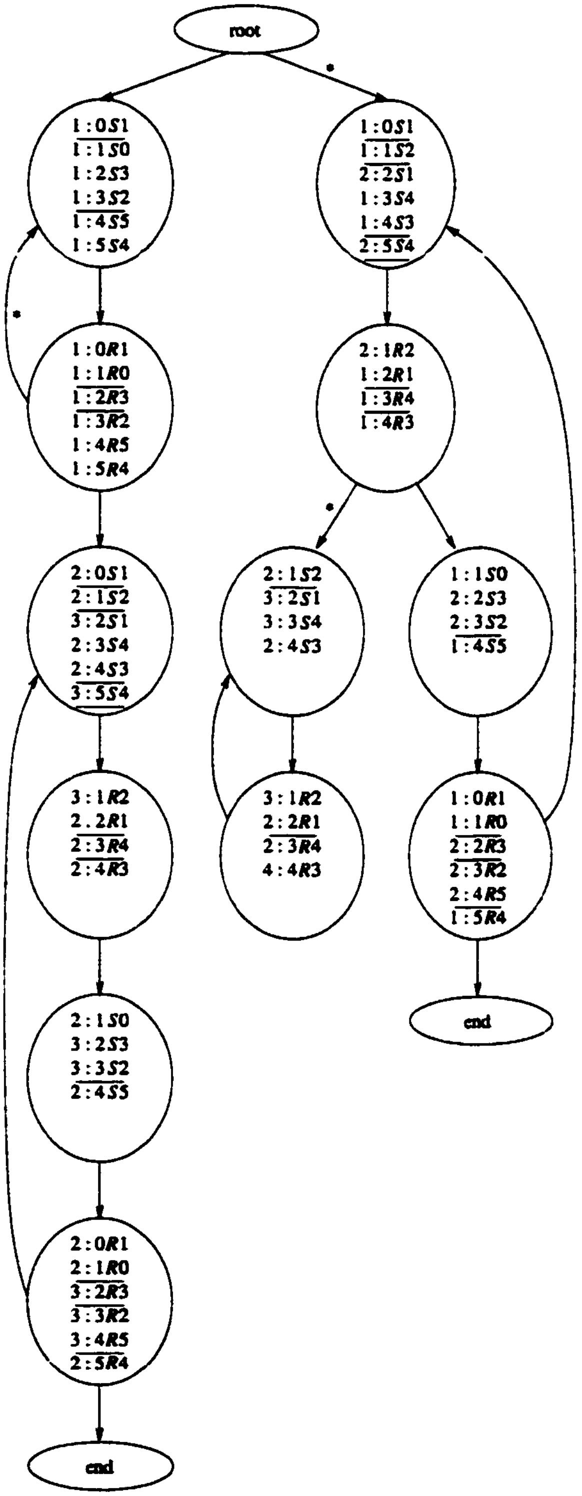

root
1:0S1
1:1S0
1:2S3
1:3S2
1:4S5
1:5S4
1:0S1
1:1S2
2:2S1
1:3S4
1:4S3
2:5S4
1:0R1
1:1R0
1:2R3
1:3R2
1:4R5
1:5R4
2:1R2
1:2R1
1:3R4
1:4R3
2:0S1
2:1S2
3:2S1
2:3S4
2:4S3
3:5S4
2:1S2
3:2S1
3:3S4
2:4S3
1:1S0
2:2S3
2:3S2
1:4S5
3:1R2
2.2R1
2:3R4
2:4R3
3:1R2
2:2R1
2:3R4
4:4R3
1:0R1
1:1R0
2:2R3
2:3R2
2:4R5
1:5R4
end
2:1S0
3:2S3
3:3S2
2:4S5
2:0R1
2:1R0
3:2R3
3:3R2
3:4R5
2:5R4
end

Figure 8.11: *LCP* and *LCP′* events for Bubble Sort

8.4 Tree Sort

Referring back to figure 2.5, we see that the distributed tree sort program of chapter 2 consists of 15 processes. We have selected P_1 to evaluate assert statement A_1. This assertion ensures that processes $P_1, P_3, P_4, P_7, P_8, P_9$ and P_{10} have correctly split, merged, and sorted the list P_0 sent to P_1. Since the left side and the right side of the tree are symmetric, a similar assert statement would be placed in P_2 to ensure processes $P_2, P_5, P_6, P_{11}, P_{12}, P_{13}$ and P_{14} correctly split, merge, and sort the list P_0 sent to P_2.

```
$P_1$:: (parent node)
        integer $child_1, child_2, parent$
        arrays $list, list_1, list_2$
        async_recv(0, $list$);
        split list into two halves: $list_1, list_2$
        async_send(3, $list_1$)
        async_send(4, $list_2$)
        async_recv(3, $list_1$)
        async_recv(4, $list_2$)
$A_1$:  assert($P_7.list$ is sorted $\wedge$ $P_8.list$ is sorted $\wedge$ $P_9.list$ is sorted $\wedge$
               $P_{10}.list$ is sorted $\wedge$ $P_3.list$ is sorted $\wedge$ $P_4.list$ is sorted $\wedge$
               $((P_7.list \cup P_8.list \cup P_9.list \cup P_{10}.list) = P_1.list) \wedge$
               $(P_7.list = P_3.list_1) \wedge (P_8.list = P_3.list_2) \wedge$
               $((P_3.list_1 \cup P_3.list_2) = P_3.list = P_1.list_1) \wedge$
               $(P_9.list = P_4.list_1) \wedge (P_{10}.list = P_4.list_2) \wedge$
               $(P_4.list_1 \cup P_4.list_2 = P_4.list = P_1.list_2) \wedge$
        $(P_1.list_1 \cup P_1.list_2 = P_1.list)$)
        merge $list_1$ and $list_2$ into $list$
        async_send(0, $list$)
```

In the correct implementation of tree sort, P_3 receives a list from P_1, and then P_3 is responsible for sorting this list and sending the sorted list to P_1. Assume P_3 erroneously sends the wrong list to P_1. Following is the incorrect implementation of P_3:

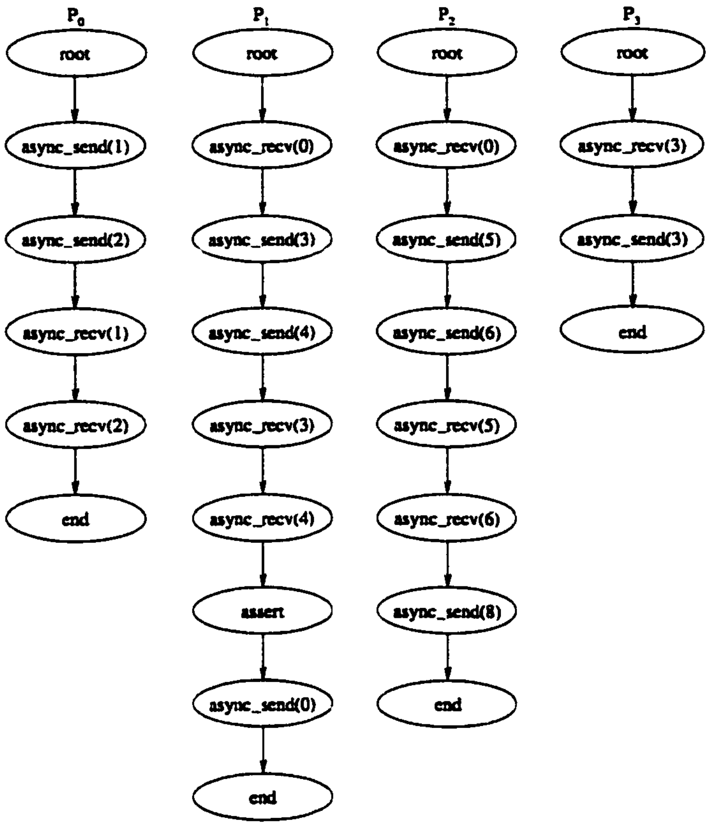

P0
root
async_send(1)
async_send(2)
async_recv(1)
async_recv(2)
end
P1
root
async_recv(0)
async_send(3)
async_send(4)
async_recv(3)
async_recv(4)
assert
async_send(0)
end
P2
root
async_recv(0)
async_send(5)
async_send(6)
async_recv(5)
async_recv(6)
async_send(8)
end
P3
root
async_recv(3)
async_send(3)
end

Figure 8.12: Flow Graphs for Tree Sort

```
$P_3$:: (parent node)
        integer $child_1, child_2, parent$
        arrays $list, list_1, list_2$
  1     async_recv(1, $list$);
  2     split list into two halves: $list_1, list_2$
  3     async_send(7, $list_1$)
  4     async_send(8, $list_2$)
  5     async_recv(7, $list_1$)
  6     async_recv(8, $list_2$)
  7     merge $list_1$ and $list_2$ into $list$
  8     async_send(1, $list_1$)
```

Line 8 is incorrect, P_3 should send *list* to P_1. Assert A_1 detects the error by two clauses evaluating to false. These clauses are $(P_3.list_1 \cup P_3.list_2 = P_3.list = P_1.list_1)$ and

($P_1.list_1 \cup P_1.list_2 = P_1.list$). The combination of these clauses identifies that $P_1.list_1$ is incorrect. Since none of the other clauses involving $P_3.list_1$ and $P_3.list_2$ evaluated to false, the false evaluation of ($P_3.list_1 \cup P_3.list_2 = P_3.list = P_1.list_1$) conveys that $P_1.list_1$ is not equal to $P_3.list$. With this information, the source of the error is easily found.

As another example of an incorrect implementation, suppose leaf process P_8 does not correctly sort its list. This error causes clause ($P_8.list$ is sorted) of A_1 to evaluate to false. None of the other clauses evaluate to false, and the source of the error is directly identified.

The flow graphs for P_0, P_1, P_2, and P_7 are shown in figure 8.12. The flow graphs for P_3, P_4, P_5 and P_6 are identical to P_2's flow graph with the exception of the destination and source of messages. Also, the flow graphs for $P_8, P_9, P_{10}, P_{11}, P_{12}, P_{13}, P_{14}$ are identical to P_7's flow graph with the exception of the destination and source of the message. The destinations and originations of the messages for the communication events are given in chapter 2.

The H graph for the tree sort program is shown in 8.13. There exists only one execution path since none of the processes have a possible branch in execution. The resulting *POG* is shown in figure 8.14. Algorithm `Bound_Assert()` returns the event `async_recv(4)` of P_2 as the last LCP' event. This event is identified in P_1's flow graph with double circles. One *POG* node represents this receive event, and that node's entry is $3{:}1R4$. This event, as well as the LCP and LCP' events determined by algorithm `Find_LCPs()`, is underlined in figure 8.14. Figure 8.15 is the time space diagram of tree sort's execution with the six messages that piggyback state information shown as solid lines.

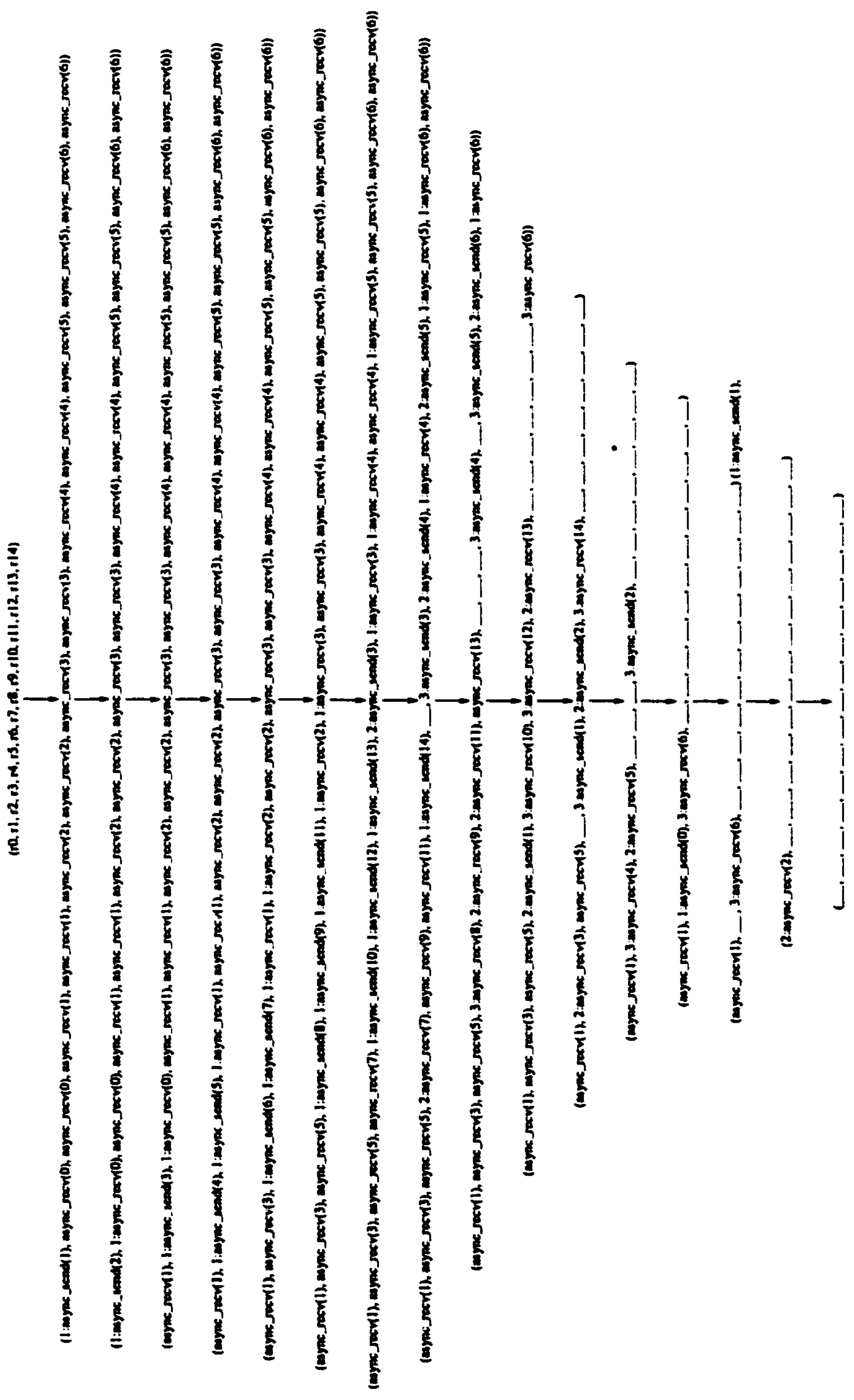

(r0, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14)

Figure 8.13: Graph *H* for Tree Sort

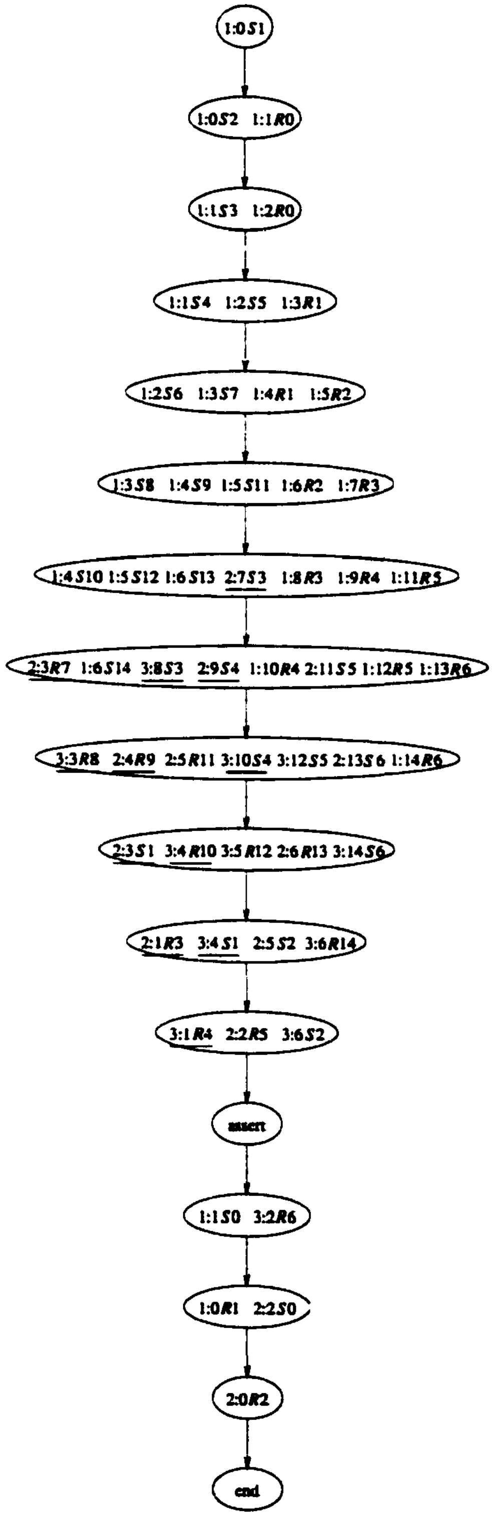

1:0S1
1:0S2 1:1R0
1:1S3 1:2R0
1:1S4 1:2S5 1:3R1
1:2S6 1:3S7 1:4R1 1:5R2
1:3S8 1:4S9 1:5S11 1:6R2 1:7R3
1:4S10 1:5S12 1:6S13 2:7S3 1:8R3 1:9R4 1:11R5
2:3R7 1:6S14 3:8S3 2:9S4 1:10R4 2:11S5 1:12R5 1:13R6
3:3R8 2:4R9 2:5R11 3:10S4 3:12S5 2:13S6 1:14R6
2:3S1 3:4R10 3:5R12 2:6R13 3:14S6
2:1R3 3:4S1 2:5S2 3:6R14
3:1R4 2:2R5 3:6S2
assert
1:1S0 3:2R6
1:0R1 2:2S0
2:0R2
end

Figure 8.14: *POG* for Tree Sort

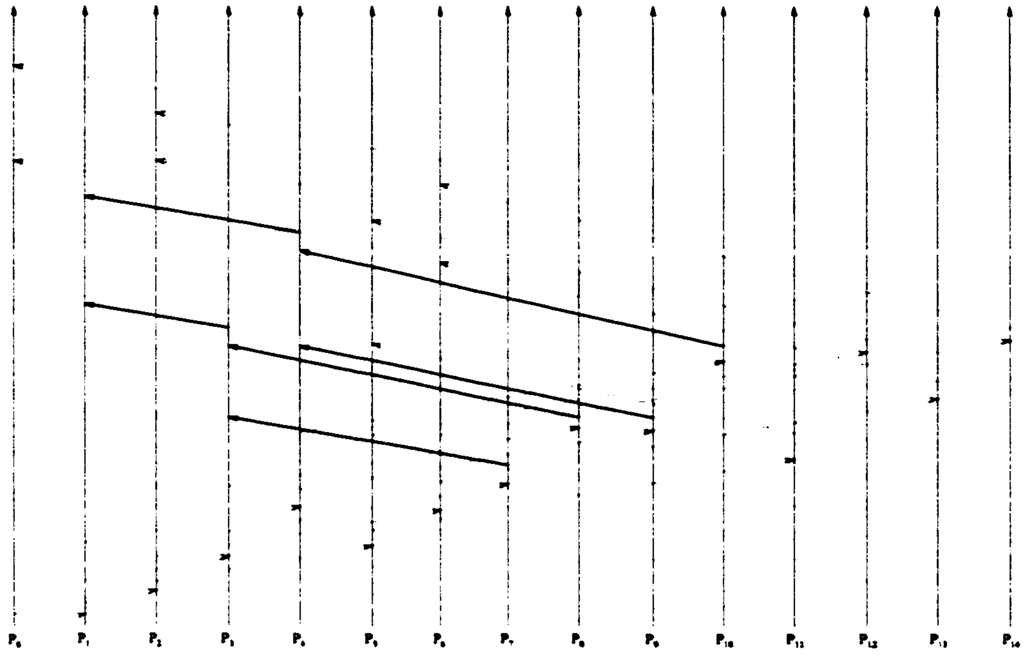


Figure 8.15: Time Space Diagram for Tree Sort

8.5 Positive Acknowledgement/Retransmission

The two process distributed program implementing positive acknowledgement and retransmission is repeated from chapter 2 with the addition of assert statement A_0. Process P_0 sends a message to P_1, and P_1 acknowledges receipt of that message. Process P_0 retransmits the message until an acknowledgement for the message is received.

```
P0::
    MsgBitSend : bit                                   /* alternating bit */
    sbuffer: message                     /* buffer for outgoing data message */
    event: (MsgArrival, CksumErr, TimeOut)          /* different interrupt events */

    MsgBitSend = 0                              /* initialize alternating bit */
    FromHost(sbuffer)                        /* get the data message from host */
    repeat
```

```
        async_send(1, sbuffer, MsgBitSend)
        StartTimer;                               /* time to wait for acknowledgement */
        wait(event)                     /* possibilities MsgArrival, CksumErr, TimeOut */
        if $event$ = MsgArrival
            async_recv(1, ack)                     /* receive the acknowledgement */
$A_0$: assert($P_1$.IncomingBit = $P_0$.MsgBitSend $\wedge$ $P_0$.MsgBitSend $\neq$ $P_1$.MsgBitReceive $\wedge$
    $P_0$.sbuffer = $P_1$.rbuffer $\wedge$ $P_1$.event = MsgArrival)
            FromHost(sbuffer)                 /* an acknowledgment has arrived intact */
            inc(MsgBitSend)                       /* increment by 1 then mod 2 */
        endif
    until doomsday
```

```
$P_1$::
    MsgBitReceive : bit                                  /* alternating bit */
    IncomingBit : bit                                 /* incoming message's bit */
    rbuffer: message                             /* buffer for incoming data message */
    event: (MsgArrival, CksumErr)                     /* different interrupt events */

    MsgBitReceive = 0                                /* initialize alternating bit */
    repeat
        wait(event)                              /* possibilities MsgArrival, CksumErr */
        if event = MsgArrival                         /* a valid message has arrived */
            async_recv(0, rbuffer, IncomingBit)             /* accept the message */
            if IncomingBit = MsgBitReceive
                ToHost(rbuffer)                          /* pass the data to the host */
                inc(MsgBitReceive)                      /* increment by 1 then mod 2 */
            endif
            async_send(0, acknowledgement)
        endif
    until doomsday
```

Assert A_0 first determines if *IncomingBit* was correctly received at P_1 and was not erroneously changed by P_1. The second clause of the assert, P_0.*MsgBitSend* $\neq$ P_1.*MsgBitReceive*, ensures that *MsgBitSend* and *MsgBitRecv* are correctly updated. The third clause, P_0.*sbuffer* = P_1.*rbuffer*, determines whether P_1 received the correct message, and the last clause, P_1.*event* = MsgArrival, ensures that P_1 sent the acknowledgement only after it received a message from P_0.

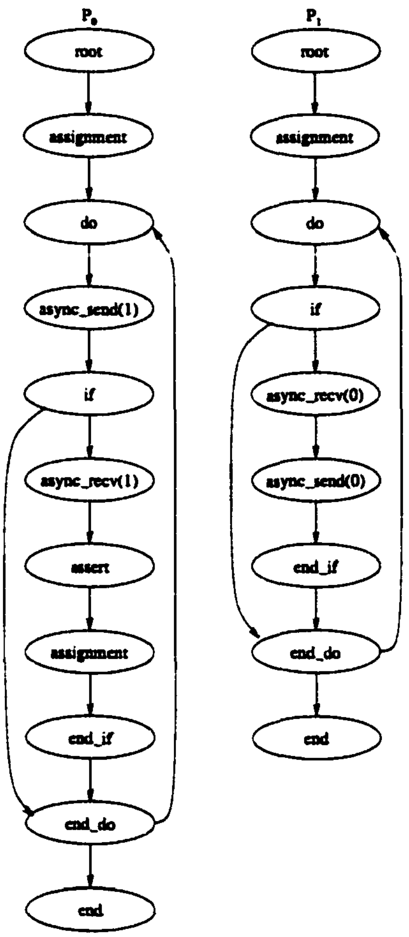

P0
root
assignment
do
async_send(1)
if
async_recv(1)
assert
assignment
end_if
end_do
end
P1
root
assignment
do
if
async_recv(0)
async_send(0)
end_if
end_do
end

Figure 8.16: Flow Graphs for Positive Ack/Retrans

In the correct implementation of this distributed program, process P_1 increments *MsgBitReceive* when a new message is received. Suppose P_1 increments *MsgBitReceive* when it receives any valid message. This error occurs if either line 18 is omitted or if line 20 is placed after line 21.

Assume line 18 is omitted. Suppose the following events occur. P_0 sends a message to P_1. P_1 receives the message and correctly passes the message to the host and increments *MsgBitReceive*. Process P_1 then sends an acknowledgement, but the acknowledgement is

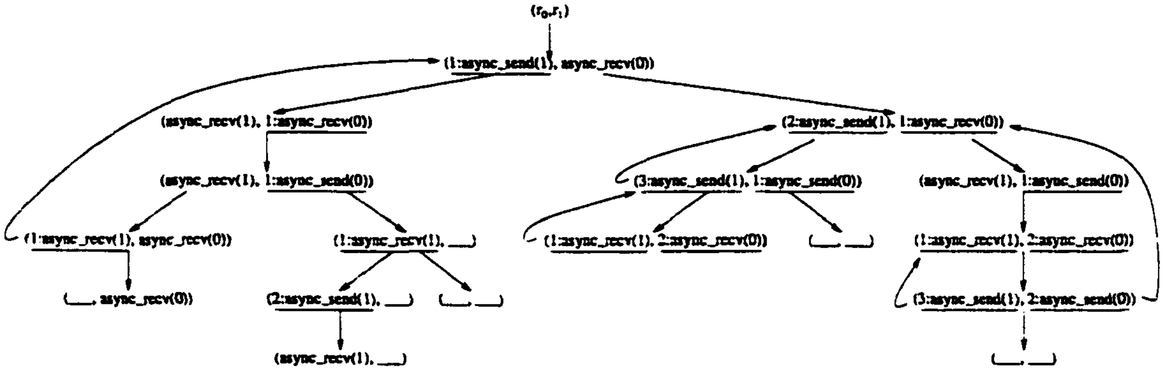

(r0,r1)
(1:async_send(1), async_recv(0))
(async_recv(1), 1:async_recv(0))
(2:async_send(1), 1:async_recv(0))
(async_recv(1), 1:async_send(0))
(3:async_send(1), 1:async_send(0))
(async_recv(1), 1:async_send(0))
(1:async_recv(1), async_recv(0))
(1:async_recv(1), ___)
(1:async_recv(1), 2:async_recv(0))
(___, ___)
(1:async_recv(1), 2:async_recv(0))
(___, async_recv(0))
(2:async_send(1), ___)
(___, ___)
(3:async_send(1), 2:async_send(0))
(async_recv(1), ___)
(___, ___)

Figure 8.17: Graph H for Positive Ack/Retrans

lost. Process P_0 times out and retransmits the same messages. Process P_1 receives a duplicate message. Since line 18 is missing, P_1 erroneously passes the message to the host and increments *MsgBitReceive*. Process P_1 then sends an acknowledgement to P_0, and the acknowledgement is received by P_0. The assert statement is evaluated. The second clause of A_0, $P_0.MsgBitSend \neq P_1.MsgBitReceive$, evaluates to false and identifies the error.

As another example of an incorrect implementation, assume P_1 sends an acknowledgement for any event. This error occurs if line 22 is placed after line 23. Suppose the following events occur. Process P_0 sends a message to P_1. The message is corrupted in transit. Process P_1 is interrupted and procedure wait returns a CKsumErr event. Line 16 evaluates to false, but then P_1 incorrectly sends an acknowledgement to P_0. Process P_0 receives the acknowledgement. The assert statement is then evaluated, and clause $P_1.event =$ MsgArrival evaluates to false. This clause identifies that P_1 sent an invalid acknowledgement.

The flow graphs are shown in figure 8.16, and graph H in shown in figure 8.17. Although the two processes' source code is short, the execution behavior of the distributed program

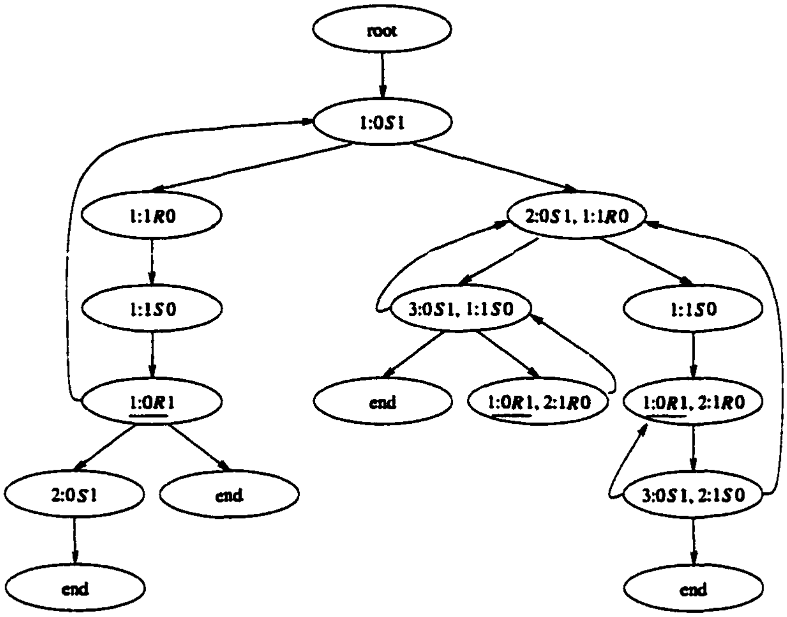

root
1:0S1
1:1R0
1:1S0
1:0R1
2:0S1
end
end
2:0S1, 1:1R0
3:0S1, 1:1S0
1:1S0
end
1:0R1, 2:1R0
1:0R1, 2:1R0
3:0S1, 2:1S0
end

Figure 8.18: *POG* for Positive Ack/Retrans

is complex. The main reason for this is the `async_recv(1, ack)` of line 8. Process P_1 will continue execution regardless of whether P_0 receives P_1's acknowledgement. The result, as shown in figure 8.17, is multiple branches of execution. The resulting *POG* is shown in figure 8.18.

The assert statement is evaluated when the `if` condition of P_0 evaluates to true. Algorithm `Bound_Assert()` identifies statement `async_recv(1)` of P_0 as the last *LCP'* event. This event is represented by three *POG* nodes which are underlined in figure 8.18. Only the messages sent from P_0 to P_1 need to piggyback state information. Algorithm `Find_LCPs()` identifies the *POG* node entries that represent the send event of P_1 as the *LCP* event. Both *LCP* and *LCP'* events are underlined in the *POG* of figure 8.19.

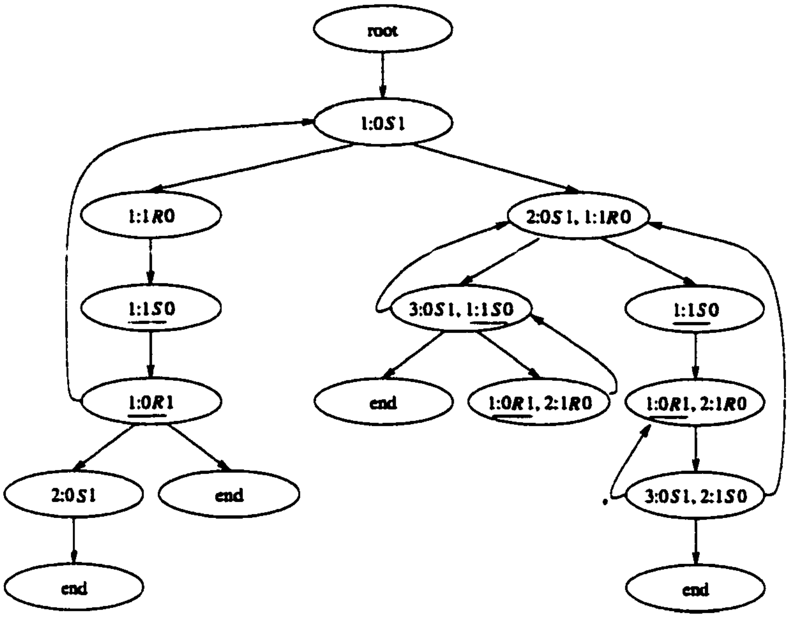


Figure 8.19: *LCP* and *LCP'* events for Positive Ack/Retrans

The five examples analyzed in this chapter are diverse in their communication behavior. Together they demonstrate the robustness of our static analysis technique. For each example. the analysis identifies the latest causally preceding communication events. The assert is evaluated with the causal global state obtained by piggybacking state information on the messages of the *LCP* and *LCP'* events.

8.6 Prototype

A prototype system has been written to demonstrate the feasibility of analyzing distributed programs for evaluating distributed asserts. Our prototype is a two-pass compiler. The grammar for our compiler is shown in appendix A. The C source files for the distributed processes are the input of the compiler. For assert statement evaluation, code is added to the processes' source files to create and maintain a causal global state, and the *LCP* messages are identified and altered to piggyback this causal global state. The remainder of the distributed program is not altered.

We will use the distributed program SETPART as our running example in the following explanation of our system. The source code for SETPART appears below.

P_0:

```
#include <stdio.h>
#include <async.h>
#include <sys/time.h>

int S[16];
int x;

main(argc,argv)
     int argc;
     char *argv[];
{
  int count;
  int numcount;

  int len;
  int i;
  int mx;

  if (argc < 2)
     {
       fprintf ( stderr, "USAGE: %s <st size>\n", argv[0]);
       exit(1);
     }

  init_async (121, 0, 2, 0, 0.0, 0, 0 );
  count = atoi(argv[1]);
  Init_List ( S, &numcount, count, 0);

  printf("Initial set in P0\n\t");
  for ( i=0; i<numcount; i++ )
       printf("%6d ", S[i]);
  printf("\n\n");

  mx = max( S );
  x = -99999;
  while (mx > x)
```

```
      {
        async_send ( 1, &mx, sizeof(mx) );
        Remove ( S, &numcount, mx );
        len = sizeof(int);
        async_recv ( 1, &x, &len, 60 );
        Add ( S, &numcount, x );
        mx = max( S );
      }

  printf("Final set in P0\n\t");
  for ( i=0; i< numcount; i++ )
       printf("%6d ", S[i]);
  printf("\n\n");

  close_async();
}
```

P_1

```
#include <stdio.h>
#include <async.h>
#include <sys/time.h>

int mn;
int T[16];
int y;

main(argc,argv)
     int argc;
     char *argv[];
{
  int count;
  int numcount;
  int len;
  int i;
  int devdata;

  devdata = 1;
  if (argc < 2)
      {
        fprintf ( stderr, "USAGE: %s <set size>\n", argv[0]);
        exit(1);
      }
  init_async ( 121, 1, 2, 0, 0.0, 0, 0 );
  count = atoi(argv[1]);
  numcount = Init_List ( T, &numcount, count, 1);

  printf("Initial set in P1\n\t");
  for ( i=0; i<numcount; i++ )
       printf("%6d ", T[i]);
  printf("\n\n");
  while (devdata > 0)
     {
       len = sizeof(int);
       devdata = async_recv ( 0, &y, &len, 60 );
       if (devdata > 0)
     {
       Add ( T, &numcount, y );
       mn = min( T );
       assert((max(CG._P0_S.S) == y) &&
         (max(CG._P0_S.S) >= mn) &&
                    (mn > CG._P0_x.x)       &&
                    (intersect(CG._P0_S.S, T) == y) );
       async_send ( 0, &mn, sizeof(mn) );
       Remove ( T, &numcount, mn );
     }
     }

  printf("Final set in P1\n\t");
  for ( i=0; i<numcount; i++ )
       printf("%6d ", T[i]);
  printf("\n\n");
  close_async();
}
```

The first pass of our compiler consists of four phases. The initial phase parses the source code in each of the process input files and creates a control flow graph for each process as described in chapter 7. A declaration table, **VarMap**, is created. Each variable in a process has an entry in the table consisting of variable type, identifier and amount of memory required.

When an assert statement is detected by the parser, an entry containing only the variable identifier is added to the list **assert_vars** for each non-local variable that occurs in the assert. Since processes can have identical variable identifiers, a notation has been developed to distinguish the process in which a variable resides. Non-local variables of the assert must be specified in the following format:

CG._*Pi_id.id*

The process number is indicated by *i*, and the *ids* indicate the variable identifier. For example, set partition's assert is

```
assert((max(CG._P0_S.S) == y) &&
     (max(CG._P0_S.S) >= mn) &&
     (mn > CG._P0_x.x) &&
     (intersect(CG._P0_S.S, T) == y) );
```

The list **assert_vars** will have two entries, `CG._P0_S.S` and `CG._P0_x.x`, after parsing this assert.

The second phase creates three files for each process: **assert*i*.h**, **pigRecv*i*.c**, and **pigSend*i*.c** where *i* is the process number. Each **assert*i*.h** file defines a data structure for the causal global state and will be included in P_i. A structure exists in the included file for each entry of **assert_vars**. The type and size of each item of **assert_vars** are found in the table **VarMap**. The singular difference between **assert*i*.h** and **assert*j*.h** is the initialization of vector time. The files **assert0.h** and **assert1.h** created for SETPART are shown below.

assert0.h:

```
#define MAXPS 2
struct
{
  struct
  {
    int S[16];
    int vtime;
  } _P0_S;
  struct
  {
    int x;
    int vtime;
  } _P0_x;
} CG, tmpCG;
int _vector_t[MAXPS] = {1, 0 };
```

```
assert1.h:
 #define MAXPS 2
 struct
 {
   struct
   {
     int S[16];
     int vtime;
   } _PO_S;
   struct
   {
     int x;
     int vtime;
   } _PO_x;
 } CG, tmpCG;
 int _vector_t[MAXPS] = {0,1 };
```

The symbol **MAXPS** indicates the number of processes in the distributed program. As shown in procedure `Update()` of chapter 4. the integer `vtime` is used for updating the causal global state. The variable `CG` is the causal global state. and the variable `tmpCG` is for temporarily holding a received causal global state. Vector time is maintained in the array `_vector_t[]`.

The file **pigSend*i*.c** is included by process P_i. This file contains the source code for function `Piggy_Send()` which piggybacks the causal global state onto an outgoing message. This function is also responsible for updating the causal global state prior to piggybacking state information. The `Piggy_Send()` functions differ for each process. `Piggy_Send()` for process P_i is only responsible for updating `CG` with the variables that reside locally in P_i. The files **pigSend0.c** and **pigSend1.c** for SETPART are shown below.

PigSend0.c

```
 #include <stdio.h>
 #include <async.h>
 Piggy_send(i, data, sizedata)
```

```
   int i;
   char *data;
   int sizedata;
 {
   char *dataptr;

   dataptr=(char *)malloc(sizeof(CG)+sizedata);
   memcpy(dataptr, data, sizedata);
   memcpy(CG._PO_S.S, S, (sizeof(int) * 16));
   CG._PO_S.vtime = _vector_t[0];
   CG._PO_x.x = x;
   CG._PO_x.vtime = _vector_t[0];
   memcpy((dataptr+sizedata), &CG, sizeof(CG));

   return(async_send(i, dataptr, sizeof(CG)+sizedata));
 }

PigSend1.c
 #include <stdio.h>
 #include <async.h>
 Piggy_send(i, data, sizedata)
   int i;
   char *data;
   int sizedata;
 {
   char *dataptr;

   dataptr=(char *)malloc(sizeof(CG)+sizedata);
   memcpy(dataptr, data, sizedata);
   memcpy((dataptr+sizedata), &CG, sizeof(CG));

   return(async_send(i, dataptr, sizeof(CG)+sizedata));
 }
```

The file `pigRecv`*i*`.c` is included by process P_i. This file contains the source code for function `Piggy_Recv()` which receives an incoming message that has been piggybacked with a causal global state. The newly received causal global state is copied into the variable `tmpCG`. The `Piggy_recv()` of P_i updates P_i's causal global state with the latest state information by comparing the `vtime` of corresponding entries in `CG` and `tmpCG`. The entry with the largest `vtime` has the latest state information. This is consistent with the causal state propagation protocol described in chapter 4. `Piggy_recv()` of P_i only updates the components of the causal global state that do not correspond to it's own variables. The files **`pigRecv0.c`** and **`pigRecv1.c`** for SETPART are shown below.

`pigRecv0.c`:

```
#include <stdio.h>
#include <async.h>
Piggy_recv(i, data, sizedata, time)
  int i;
  char *data;
  int *sizedata;
  int time;
{
  char *dataptr;
  int CGsize;

  CGsize = sizeof(CG) + *sizedata;
  dataptr=(char *)malloc(CGsize);

  if (async_recv(i, dataptr, &CGsize, time) < 0)

    return(-1);
  *sizedata = CGsize - sizeof(CG);
  memcpy(data, dataptr, *sizedata);
  memcpy(&tmpCG, (dataptr + *sizedata), sizeof(CG));
  return(*sizedata);
}
```

`pigRecv1.c`:

```
#include <stdio.h>
#include <async.h>
Piggy_recv(i, data, sizedata, time)
  int i;
  char *data;
  int *sizedata;
  int time;
{
  char *dataptr;
  int CGsize;

  CGsize = sizeof(CG) + *sizedata;
  dataptr=(char *)malloc(CGsize);

  if (async_recv(i, dataptr, &CGsize, time) < 0)

    return(-1);
  *sizedata = CGsize - sizeof(CG);
  memcpy(data, dataptr, *sizedata);
  memcpy(&tmpCG, (dataptr + *sizedata), sizeof(CG));
```

```
   if (CG._PO_S.vtime < tmpCG._PO_S.vtime)
   { memcpy(CG._PO_S.S, tmpCG._PO_S.S, (sizeof(int) * 16));
     CG._PO_S.vtime = tmpCG._PO_S.vtime;
   }
   if (CG._PO_x.vtime < tmpCG._PO_x.vtime)
   { memcpy(&CG._PO_x.x, &tmpCG._PO_x.x, sizeof(CG._PO_x.x));
     CG._PO_x.vtime = tmpCG._PO_x.vtime;
   }
   return(*sizedata);
 }
```

The third phase determines the LCP and LCP' events. The H graph and the POG are constructed according to the algorithms *Crt_H* and *Crt_POG* given in chapter 7. From the POG, the LCP and LCP' events are determined. These events are found according to the algorithms *Bound_Assert* and *Find_LCPs* also of chapter 7. This phase produces the same results for SETPART that where given in section 8.1.

The last phase of pass one forks a child process that is the second pass of the compiler and establishes a pipe from the first pass process to the second pass process. Through this pipe the identification of the LCP and LCP' events are sent to the second pass. The identification of each event consists of two numbers: process identifier and communication node identifier. As the nodes of the control flow graph are created in phase one, a counter **commoNodeID** is assigned to each communication node. The counter **commoNodeID** is initialized to one each time a new control flow graph is built and incremented each time an **async_recv** or **async_send** node is added.

The second pass of the compiler reads the LCP and LCP' event identifications and stores this information in the table **IDMap**. The distributed processes are parsed again by pass two, and a new source file is created for each process. The name of each file is **N.***file***.c**, where *file* is the name of the original source file. If the names of SETPART's original source files are `proc0.c` and `proc1.c`, then `N.proc0.c` and `N.proc1.c` are the two new source files created by pass two. These new source files are the result of altering the original files to incorporate piggybacking of data on the LCP and LCP' events.

The first line written in process P_i's new file is `#include "asserti.h"`. When a line of source code is read by the parser that is not an `async_send()` or `async_recv()` function call that corresponds to an LCP or LCP' event, the line is written to the new source file. The parsing of pass two does not create internal data structures, only a `commoNodeID` counter is maintained as in pass one. When a send or receive command is detected during parsing, the `commoNodeID` is incremented and the table `IDMap` is checked to determine if the command is an LCP or LCP' event. If the command is an LCP or LCP' event and is an `async_send()` function call, the function name is replaced with `Piggy_send`. The parameters of the function are not altered. A line is also added after the function call to update vector time. If the command is an LCP or LCP' event and is an `async_recv()` function call, the function name is replaced with `Piggy_recv`. Again the parameters of the function are not altered, and a line is added after the function call to update vector time.

After the source file for P_i has been parsed, two lines are added to the end of the new source file to include the `pigRecvi.c` file and the `pigSendi.c` file, thus completing the creation of the new file. Once all new source file are created, our two pass compiler is finished. The new files for SETPART are shown below.

N.proc0.c:

```
#include "assert0.h"
#include <stdio.h>
```

```
#include <async.h>
#include <sys/time.h>

int S[16];
int x;

main(argc,argv)
     int argc;
     char *argv[];
{
  int count;
  int numcount;

  int len;
  int i;
  int mx;

  if (argc < 2)
     {
       fprintf ( stderr, "USAGE: %s <st size>\n", argv[0]);
       exit(1);
     }

  init_async (121, 0, 2, 0, 0.0, 0, 0 );
  count = atoi(argv[1]);
  Init_List ( S, &numcount, count, 0);

  printf("Initial set in P0\n\t");
  for ( i=0; i<numcount; i++ )
       printf("%6d ", S[i]);
  printf("\n\n");

  mx = max( S );
  x = -99999;
  while (mx > x)
     {
       Piggy_send(1, &mx, sizeof(mx));
       _vector_t[0]++;

       Remove ( S, &numcount, mx );
       len = sizeof(int);
       async_recv(1, &x, &len, 60);
       _vector_t[0]++;

       Add ( S, &numcount, x );
       mx = max( S );
     }

  printf("Final set in P0\n\t");
  for ( i=0; i< numcount; i++ )
```

```
        printf("%6d ", S[i]);
    printf("\n\n");

    close_async();
  }
  #include "pigRecv0.c"
  #include "pigSend0.c"
```

N.proc1.c:

```
  #include "assert1.h"
  #include <stdio.h>
  #include <async.h>
  #include <sys/time.h>


  int mn;
  int T[16];
  int y;

  main(argc,argv)
       int argc;
       char *argv[];
  {
    int count;
    int numcount;
    int len;
    int i;
    int devdata;

    devdata = 1;
    if (argc < 2)
       {
         fprintf ( stderr, "USAGE: %s <set size>\n", argv[0]);
         exit(1);
       }
    init_async ( 121, 1, 2, 0, 0.0, 0, 0 );
    count = atoi(argv[1]);
    numcount = Init_List ( T, &numcount, count, 1);

    printf("Initial set in P1\n\t");
    for ( i=0; i<numcount; i++ )
         printf("%6d ", T[i]);
    printf("\n\n");

    while (devdata > 0)
       {
         len = sizeof(int);
         devdata =   Piggy_recv(0, &y, &len, 60);
```

```
      _vector_t[1]++;

      if (devdata > 0)
    {
     Add ( T, &numcount, y );
     mn = min( T );
     assert((max(CG._PO_S.S) == y) &&
      (max(CG._PO_S.S) >= mn) &&
                  (mn > CG._PO_x.x)       &&
                  (intersect(CG._PO_S.S, T) == y) );
     async_send(0, &mn, sizeof(mn));
          _vector_t[1]++;

     Remove ( T, &numcount, mn );
    }
     }

  printf("Final set in P1\n\t");
  for ( i=0; i<numcount; i++ )
       printf("%6d ", T[i]);
  printf("\n\n");

  close_async();
}
#include "pigRecv1.c"
#include "pigSend1.c"
```

The new files are ready for compilation and execution. After compilation, the executing programs create and maintain a causal global state for the assert statements. The assert statement is evaluated using the causal global state transmitted via the identified *LCP* messages. Despite the potential disturbance to the timing of the distributed program's execution by increasing message sizes, the timing changes of our technique are minor compared to other existing techniques. We do not add messages to the distributed execution and execution is not suspended to gather state information. By preserving the causal relationships, the distributed program maintains the same functionality of the original.

Chapter 9

Conclusions

Our research addresses the difficult issue of monitoring the execution of a distributed system. We have developed a runtime method for monitoring both stable and unstable properties that does not disrupt the computation of the distributed system. We used the sequential assert statement as the basis for our development of the distributed assert statement. A distributed assert statement is evaluated with that statement's causal global state. The causal global state incorporates the state of the system as a whole as it may have causal impact upon the assert statement.

We have developed a runtime protocol that constructs the causal global state and evaluates the assert statement where no additional synchronization or message passing is imposed on the distributed application. The causal global state is immediately available providing real-time feedback.

The protocol increases the size of only the messages corresponding to the *LCP* and *LCP'* events. We refined our protocol by statically analyzing the distributed program in order to reduce the amount of piggybacked data. Our techniques are able to analyze complex distributed programs where each process has branches in execution and nested loops. The *POG* is able to represent all concurrent and causal relationships and all possible paths of the system's execution. By having this information condensed into the *POG*, we are able to determine the assert's LCP and LCP' events.

In conclusion, our work provides a practical solution for monitoring a distributed system's execution that is not only theoretically sound, but also implementable. Our solution provides a powerful monitoring tool that can be used throughout the system's life cycle, and the only responsibility left to the distributed program developer is to assert predicates as needed. The developer must understand causality to create informative predicates since they will be evaluated with a causal global state.

9.1 Communication Systems

Two message passing systems are commonly used for writing distributed programs. These systems are PVM (Parallel Virtual Machine) and MPI(Message Passing Interface). Both can run on a variety of architecture platforms and provide a library of communication commands. Our work has not been ported to these systems, but we will address what would be involved.

PVM is the forerunner of MPI. PVM provides asynchronous reliable FIFO point-to-point communication on a heterogeneous network of machines running Unix. A process sends a message to another process with the command **pvm_send()**. The **pvm_send()** has the same functionality as our **async_send()**. A process receives a message with one of the following commands: **pvm_recv()**, **pvm_trecv()**, or **pvm_nrecv()**. The command **pvm_recv()** is a blocking receive and is equivalent to our **async_recv()**. As with our **async_recv()**, there is an option to receive from any process instead of a specific process. This is achieved with a -1 in the process identification field. We have not addressed this issue in our analysis, although only minor modifications are necessary to handle the -1 option. Consider a four process system with the following line in process P_0.

```
async_recv(-1, y)
```

In terms of flow of execution, this is equivalent to the nested if/else statements shown below. Since all paths of executable are assumed possible in our analysis, boolean expressions are not necessary and the textual order of the receiving processes is irrelevant in the nested if/else statements.

```
if ()
    async_recv(1, y)
else if ()
        async_recv(2, y)
    else if ()
            async_recv(3, y)
```

We are able to analyze communication commands embedded in nested if/else statements. The only modification required to our analysis is to recognize the -1 option and treat this as nested if/else statements.

The command **pvm_trecv()** is a blocking receive with the ability to timeout after a specified length of time. The command **pvm_nrecv()** is non-blocking receive. If a message has not arrived when **pvm_nrecv()** is executed, it returns immediately. Our **async_recv()** has an option of specifying a length of time to wait for a message. Setting this field to zero is equivalent to a nonblocking receive. We did not explicitly address nonblocking and timeout receives, but they can be analyzed with minor modifications. Consider a four process system with the following line in process P_0.

```
async_recv(1, y, 0)
```

The zero is the timeout. In terms of execution flow, this is equivalent to the if statement shown below.

```
if ()
    async_recv(1, y)
```

The only change to our analysis is to recognize the use of the timeout field and to analyze in the same manner as an `if` statement and a receive command.

Multicasting is also possible in PVM. The command **`pvm_mcast()`** is executed by the sender of the multicast message. The sender of the multicast messages may send to all processes except itself. An array of process identifiers is provided to the command **`pvm_mcast()`** specifying which processes should be sent the message. We do not have an equivalent command in our asynchronous library. If the array contains the values 1 and 2, this is equivalent to two asynchronous send commands, one sending to P_1 and one sending to P_2. Our analysis is able to handle a sequential series of send commands. The modifications necessary to analyze a multicast command are to read the pids from the array and treat each entry as a separate send command.

MPI provides reliable FIFO communication which can be either asynchronous or synchronous indicated by the send command. Communication can also be either blocking or nonblocking. Both the send and receive commands indicate whether blocking is desired. MPI's and PVM's blocking have different semantics. MPI attempts to improve system performance by overlapping communication and computation. Nonblocking communication is one way to achieve this overlap. A nonblocking send is initiated with a command that copies the message to a buffer and immediately returns. While computation is preceding, the message is copied out of the send buffer. The send is completed with a command to verify that the message has been transferred. Similarly, a receive command initiates the receive operation and immediately returns. While computation continues, data is transfered into the receive buffer. A separate command completes the receive operation.

MPI's library of communication commands is large, and it is not necessary to discuss each command. We will describe how each type of communication can be achieved with a subset of the commands. Asynchronous communication can be achieved with the communication pair **`MPI_BSend()`** and **`MPI_Recv()`**. The B preceding Send indicates that message buffering is to be used. The send blocks by default, meaning the send will wait until the message is copied out of the sender's buffer before it returns control to the caller. The receive also blocks by default, meaning it returns only after the receive buffer contains the message. MPI's blocking asynchronous communication can be analyzed as we currently analyze our **`async_send()`** and **`async_recv()`**.

Nonblocking communication is indicated with an I in the communication commands: **`MPI_IBSend()`** and **`MPI_IRecv()`**. The command **`MPI_IBSend()`** places the message in the buffer. The command **`MPI_Test()`** verifies that the send has completed. We only need to analyze the **`MPI_IBSend()`**, and it can be analyzed in the same manner as **`async_send()`**.

The contents of the send buffer reflect the causal information of the sending process. The computation that occurs between the `MPI_IBSend()` and `MPI_Test()` do not affect the causal global state and can be considered as occurring after the send. The command `MPI_IRecv()` only initiates the receiving of the message. The command `MPI_WAIT()` is one of several commands that can complete the receive. The command `MPI_WAIT()` waits for the receive to complete. The commands that complete the receipt of the message should be analyzed in the same manner as `async_recv()` since this is when the message is received by the process.

The commands for synchronous communication are `MPI_SSend()` and `MPI_Recv()`. Our work will require modifications to analyze synchronous communication. Synchronous message passing means that the sending process blocks until the message is received by the destination process. We discussed synchronous communication when describing Taylor's work in chapter 6. Since the rendezvous of a send/receive pair in the synchronous domain can be considered a single event on the sending and receiving processes. the algorithms for constructing the *POG* and the *H* will require modification to correctly represent the happens before relationships. The algorithms for finding the *LCP* and *LCP'* events will also require minor modifications.

MPI's communication commands have the same options that are available with PVM's commands. We discussed the analysis of these options when describing PVM. For example, the MPI receive command also has a wild card to indicate it will accept a message from any process. MPI also provides commands for broadcasting. The analysis of these broadcast commands can be handle in the same manner as with PVM multicast commands.

In conclusion, the major work for analysis programs written in either of these two message passing systems is for synchronous communication. As described, the remaining work will require minor modifications for recognizing the particular system's asynchronous communication commands.

9.2 Complexity issues of static analysis

The worst case performance of our static analysis is exponential in the number of possible concurrency states. For the worst case, assume every node of a flow graph can occur in the same concurrency state with every node from the other processes' flow graphs. If we let T be the number of nodes of all the processes' flow graphs, then an upper bound on the number of nodes of one flow graph is $O(T)$. The worst case bound on the number of concurrency states is $O(T^N)$, where N is the number of processes in the distributed application.

Although static analysis can have exponential performance, the time spent analyzing does not affect the execution of the distributed system. The analysis is done prior to execution, and provides insight into the application's behavior.

Performance improving refinements to the analysis algorithms have been considered. Localized portions of the *POG* can be constructed based on the location of the assert statement. Only the events that occur before the execution of the assert statement need to be represented in the *POG*. Representation of communication events that occur after the last LCP' events is not necessary to determine the remaining LCP and LCP' events. Our algorithms can be modified to determine the last LCP' events before constructing H and

POG graphs. When a last *LCP'* event is represented in *H*, construction of that branch of execution can stop. This can result in a smaller *H* and *POG*, depending on the location of the assert statement.

Space conservation is possible by not generating the complete *H* graph prior to generating the *POG*. As a portion of the *H* graph is generated, the corresponding portion of the *POG* can be generated. This portion of *H* is no longer needed and can be discarded. The space required to store the entire *H* graph would not be necessary.

9.3 Future Work

Our work can be extended in several directions. Three major areas are described.

9.3.1 Data Analysis

To minimize the amount of piggybacked data, we statically analyze a distributed program and identify the *LCP* and *LCP'* events. This can greatly reduce the number of messages piggybacking data. Additional reductions can be obtained by performing data analysis with regard to the assert statement. In the simplest case, processes only send state information regarding variables used in the asserted predicate. The amount of data piggybacked, and the sizes of the causal state buffers are reduced to include only relevant variables. The maximum size of a process's causal state buffer is one tuple for each variable in the assert. Since a process only piggybacks the contents of its causal state buffer, this maximum also applies to the increased size of messages.

Consider a distributed program where a process's *LCP* event is executed more than one time (e.g., it occurs in the body of a loop), as demonstrated in the distributed program SETPART where process P_0's *LCP* event occurs in a while loop. If P_0's state information changes every time the *LCP* event is executed, then this state information should be piggybacked to correctly propagate the state of the process. If, however, the state information does not change, piggybacking duplicate state information is not necessary.

Sophisticated static analysis, such as data flow analysis [1], can provide the information required to determine whether the state of the process has changed since the last piggybacking of state information. This type of static analysis, in combination with determining the *LCP* and *LCP'*, can provide additional reductions in the amount of piggybacked data.

9.3.2 Modifications to the Distributed Program

If we change the location of an assert statement or add assert statements to the distributed application, the affects to our static analysis are minor. The *POG* does not require modification since a different assert location does not affect the concurrency and causal relationships of the distributed program. When an assert is added or relocated in process P_i, P_i's flow graph can be updated with the appropriate location of the assert node. As with all assert statements, algorithm `Bound_Assert()` is called to determine the *last* *LCP'* events, and algorithm `Find_LCPs()` is called to determine the *LCP* and *LCP'* events of the assert statement.

If the assert's predicate is changed, this will only affect data analysis. Although we have not developed these algorithms, we suspect that additional variables will not invalidate the prior data analysis. If variables are removed. the corresponding portion of the data analysis should also be removed.

If the distributed program is altered. the effects to the already existing flow graphs and *POG* are dependent on the type of changes. Changes to assignment statements will not affect the *POG* but may alter the data analysis. Additions or deletion of control constructs which do not alter communication events will not affect the *POG*. If control constructs are added or deleted that affect communication events. or if communication events are added or deleted. the *POG* is affected. The effects may be incremental. meaning that only a portion of the communication analysis requires reevaluation.

Since distributed assert statements are initially intended as a tool for debugging. altering the distributed program is expected. Incremental static analysis may provide a feasible and efficient solution for updating the flow graphs and the *POG*.

9.3.3 Global Assert Statement

We have demonstrated the usefulness of evaluating assert statements with causal global states. but distributed systems may remain which require their execution to be monitored with global states. In chapter 3, algorithms that capture global states of the distributed system's execution, problems capturing global states, and the lack of meaningful predicate evaluation with these states were described. Two of our conclusions about global state reasoning were (1) the consideration of all global states of the system is required for a meaningful evaluation of the predicate, and (2) obtaining global states should not invalidate other global states. Our work can be extended by developing a meaningful run-time evaluation of a global assert statement, i.e., evaluation against all consistent cuts that include the assert statement.

The *POG* is useful for evaluating a global assert statement. It provides the information needed to determine the consistent cuts of the distributed system's computation that include the assert statement. By examining a partial order of a distributed program, we can determine a lower and upper bound communication event in each process that define the region of execution that is concurrent to an assert statement. If P_j's lower and upper bound events are $lower_j$ and $upper_j$, then all events in P_j that happen between $lower_j$ and $upper_j$ are concurrent to the assert statement. A process's *LCP* message is the lower bound message of the process's concurrent region. The upper bounds can be determined from the *POG* by a similar method to *LCP* determination with node traversal occurring downward instead of upward. Once the lower and upper bounds are found in each process, all valid consistent cuts of the assert can be constructed from the concurrent regions' events.

A run-time method of gathering the information of the consistent cuts is required for global assert statement evaluation. One possibility is to send each local state and corresponding vector time that results from the execution of an event concurrent to the assert to a monitor process. The monitor process can glue together, using vector time stamps, the received local states to form global states for assert statement evaluation. The monitor process will have all the state information necessary for a meaningful evaluation of the assert statement. An evaluation method based on gathering state information concurrent to the assert is meaningful since evaluation is done with all global states that result from a consistent cut including the assert statement.

Admittedly, this is only a starting point for developing a global evaluation method, but the majority of the static analysis exists in the *POG*.

9.4 Concluding Remarks

A meaningful and reliable technique for examining the execution of distributed programs has been our goal. By developing both causal distributed assert statements and a static analysis technique for determining the LCP and LCP' events for piggybacking state information, we have achieved our goal with minimal interference to the execution of a distributed program. Existing run time debugging techniques are not reliable for detecting buggy programs since they capture only one of many global states. The one captured global state may or may not provide meaningful information. To capture a global state, these techniques add messages to the distributed execution which alter the causal relationships among events.

Our results provide a practical tool for the distributed system engineer. As demonstrated with our analyzed programs, the examination of an execution is easily achieved by inserting assert statements that express the expected behavior of the program. Our prototype evaluates the assert without requiring the programmer to alter the distributed program or to log state information. The programmer will need to rethink his debugging strategy. Instead of thinking globally, a causal view of the execution is necessary. Once this is achieved, causal assert statements convey meaningful insight into the program's behavior.

Appendix A

Grammar

The italicized variables are nonterminals. and the all capitalized nonterminals are tokens in the lexer. Terminals appear in monospaced font.

translation_unit → *external_decl*
| *translation_unit external_decl*
external_decl → *function_defn*
| *declaration*
function_defn → *decl_specifiers declarator decl_list compound_stmt*
| *decl_specifiers declarator compound_stmt*
| *declarator decl_list compound_stmt*
| *declarator compound_stmt*
| *POUND* < *postfix_expr* >
| *POUND* < *postfix_expr* / *postfix_expr* >
| *POUND* " *postfix_expr* "
| *POUND* " *postfix_expr* / *postfix_expr* "

```
decl_specifiers         →  storage_class_specifier
                        |  storage_class_specifier decl_specifiers
                        |  type_specifier
                        |  type_specifier decl_specifiers
                        |  type_qualifier
                        |  type_qualifier decl_specifiers
init_declarator_list    →  init_declarator
                        |  init_declarator_list , init_declarator
init_declarator         →  declarator
                        |  declarator = initializer
storage_class_specifier →  TYPEDEF
                        |  EXTERN
                        |  STATIC
                        |  AUTO
                        |  REGISTER
type_specifier          →  VOID
                        |  CHAR
                        |  SHORT
                        |  INT
                        |  LONG
                        |  FLOAT
                        |  DOUBLE
```

| SIGNED
| UNSIGNED
| struct_or_union_specifier
| enum_specifier
| TYPE_NAME

struct_or_union_specifier
→ struct_or_union IDENTIFIER struct_decl_list
| struct_or_union struct_decl_list
| struct_or_union IDENTIFIER

struct_or_union → STRUCT
| UNION

struct_decl_list → struct_decl
| struct_decl_list struct_decl

struct_decl → specifier_qualifier_list struct_declarator_list ;

specifier_qualifier_list → type_specifier specifier_qualifier_list
| type_specifier
| type_qualifier specifier_qualifier_list
| type_qualifier

struct_declarator_list → struct_declarator
| struct_declarator_list , struct_declarator

struct_declarator → *declarator*

| *: constant_expr*

| *declarator : constant_expr*

enum_specifier → *ENUM enumerator_list*

| *ENUM IDENTIFIER enumerator_list*

| *ENUM IDENTIFIER*

enumerator_list → *enumerator*

| *enumerator_list , enumerator*

enumerator → *IDENTIFIER*

| *IDENTIFIER = constant_expr*

type_qualifier → *CONST*

| *VOLATILE*

declarator → *pointer direct_declarator*

| *direct_declarator*

direct_declarator → *IDENTIFIER*

| *(declarator)*

| *direct_declarator [CONSTANT]*

| *direct_declarator [ ]*

| *direct_declarator (parameter_type_list)*

| *direct_declarator (identifier_list)*

| *direct_declarator ( )*

pointer → *

| * type_qualifier_list

| * pointer

| * type_qualifier_list pointer

type_qualifier_list → type_qualifier

| type_qualifier_list type_qualifier

parameter_type_list → parameter_list

| parameter_list , ELIPSIS

parameter_list → parameter_decl

| parameter_list , parameter_decl

parameter_decl → decl_specifiers declarator

| decl_specifiers abstract_declarator

| decl_specifiers

identifier_list → IDENTIFIER

| identifier_list , IDENTIFIER

type_name → specifier_qualifier_list

| specifier_qualifier_list abstract_declarator

abstract_declarator → pointer

| direct_abstract_declarator

| pointer direct_abstract_declarator

direct_abstract_declarator

→ (***abstract_declarator***)

| [ ]

| [***constant_expr***]

| ***direct_abstract_declarator*** [ ]

| ***direct_abstract_declarator*** [***constant_expr***]

| ( )

| (***parameter_type_list***)

| ***direct_abstract_declarator*** ( )

| ***direct_abstract_declarator*** (***parameter_type_list***)

initializer → ***assignment_expr***

| ***initializer_list***

| ***initializer_list*** ,

initializer_list → ***initializer***

| ***initializer_list*** , ***initializer***

stmt → ***labeled_stmt***

| ***compound_stmt***

| ***expr_stmt***

| ***selection_stmt***

| ***iteration_stmt***

| ***jump_stmt***

labeled_stmt → ***IDENTIFIER : stmt***

| ***CASE constant_expr : stmt***

| ***DEFAULT : stmt***

compound_stmt →

| ***stmt_list***

| ***decl_list***

| ***decl_list stmt_list***

decl_list → ***declaration***

| ***decl_list declaration***

stmt_list → ***stmt***

| ***stmt_list stmt***

expr_stmt → ;

| ***expr*** ;

selection_stmt → ***IF (expr) stmt***

| ***IF (expr) stmt ELSE stmt***

| ***SWITCH (expr) stmt***

iteration_stmt → ***WHILE whileprod (expr) stmt***

| ***DO doprod stmt UNTIL (expr) ;***

| ***FOR tempprod (expr_stmt expr_stmt) stmt***

| ***FOR tempprod (expr_stmt expr_stmt expr) stmt***

whileprod → {}

doprod → {}

```
tempprod               →   {}
jump_stmt              →   CONTINUE ;
                       |   BREAK ;
                       |   RETURN ;
                       |   RETURN expr ;
relational_expr        →   shift_expr
                       |   relational_expr < shift_expr
                       |   relational_expr > shift_expr
                       |   relational_expr LE_OP shift_expr
                       |   relational_expr GE_OP shift_expr
shift_expr             →   additive_expr
                       |   shift_expr LEFT_OP additive_expr
                       |   shift_expr RIGHT_OP additive_expr
additive_expr          →   multiplicative_expr
                       |   additive_expr + multiplicative_expr
                       |   additive_expr - multiplicative_expr
multiplicative_expr    →   cast_expr
                       |   multiplicative_expr * cast_expr
                       |   multiplicative_expr / cast_expr
                       |   multiplicative_expr % cast_expr
cast_expr              →   unary_expr
                       |   ( type_name ) cast_expr
```

```
unary_expr            →   postfix_expr
                      |   INC_OP unary_expr
                      |   DEC_OP unary_expr
                      |   unary_operator cast_expr
                      |   SIZEOF unary_expr
                      |   SIZEOF ( type_name )
argument_expr_list    →   assignment_expr
                      |   argument_expr_list , assignment_expr
postfix_expr          →   primary_expr
                      |   postfix_expr [ expr ]
                      |   postfix_expr ( )
                      |   postfix_expr ( argument_expr_list )
                      |   postfix_expr . IDENTIFIER
                      |   postfix_expr PTR_OP IDENTIFIER
                      |   postfix_expr INC_OP
                      |   postfix_expr DEC_OP
                      |   SEND ( cast_expr , cast_expr , cast_expr )
                      |   RECV ( cast_expr , cast_expr , cast_expr , cast_expr )
                      |   ASSERT ( expr )
primary_expr          →   IDENTIFIER
                      |   CONSTANT
                      |   STRING_LITERAL
```

| (expr)

unary_operator → &

| *

| +

| -

| ~

| !

equality_expr → relational_expr

| equality_expr EQ_OP relational_expr

| equality_expr NE_OP relational_expr

and_expr → equality_expr

| and_expr & equality_expr

exclusive_or_expr → and_expr

| exclusive_or_expr ^ and_expr

inclusive_or_expr → exclusive_or_expr

| inclusive_or_expr | exclusive_or_expr

logical_and_expr → inclusive_or_expr

| logical_and_expr AND_OP inclusive_or_expr

logical_or_expr → logical_and_expr

| logical_or_expr OR_OP logical_and_expr

conditional_expr → logical_or_expr

| logical_or_expr ? expr : conditional_expr

assignment_expr → *conditional_expr*

| *unary_expr assignment_operator assignment_expr*

assignment_operator → =

| *MUL_ASSIGN*

| *DIV_ASSIGN*

| *MOD_ASSIGN*

| *ADD_ASSIGN*

| *SUB_ASSIGN*

| *LEFT_ASSIGN*

| *RIGHT_ASSIGN*

| *AND_ASSIGN*

| *XOR_ASSIGN*

| *OR_ASSIGN*

expr → *assignment_expr*

| *expr* , *assignment_expr*

constant_expr → *conditional_expr*

declaration → *decl_specifiers* ;

| *decl_specifiers init_declarator_list* ;

D → [0-9]

L → [a-xA-Z_]

H → [a-fA-F0-9]

E → [Ee][+-]?D^+

FS → `(f|F|l|L)`

IS → `(u|U|l|L)*`

AUTO → `auto`

BREAK → `break`

CASE → `case`

CHAR → `char`

CONST → `const`

CONTINUE → `continue`

DEFAULT → `default`

DO → `do`

DOUBLE → `double`

ELSE → `else`

ENUM → `enum`

EXTERN → `extern`

FLOAT → `float`

FOR → `for`

IF → `if`

INT → `int`

INT → `FILE`

LONG → `long`

REGISTER → `register`

RETURN → `return`

SHORT → `short`

SIGNED → `signed`

SIZEOF → `sizeof`

STATIC → `static`

STRUCT → `struct`

SWITCH → `switch`

TYPEDEF → `typedef`

UNION → `union`

UNSIGNED → `unsigned`

UNTIL → `until`

VOID → `void`

VOLATILE → `volatile`

WHILE → `while`

SEND → `async_send`

RECV → `async_recv`

ASSERT → `assert`

POUND → `#include`

IDENTIFIER → $L(L|D)^{*}$

CONSTANT → 0[xX]$H^{+}IS?$

| 0$D^{+}IS?$

| $D^{+}IS?$

| $D^{+}EFS?$

| $D^{*}.D^{+}(E)?FS?$

| $D^{+}.D^{*}(E)?FS?$

STRING_LITERAL → `"(\.|[^\"])*"`

RIGHT_ASSIGN → `>>=`

LEFT_ASSIGN → `<<=`

ADD_ASSIGN → `+=`

SUB_ASSIGN → `-=`

MUL_ASSIGN → `*=`

DIV_ASSIGN → `/=`

MOD_ASSIGN → `%=`

AND_ASSIGN → `&=`

XOR_ASSIGN → `^=`

OR_ASSIGN → `|=`

RIGHT_OP → `>>`

LEFT_OP → `<<`

INC_OP → `++`

DEC_OP → `--`

PTR_OP → `->`

AND_OP → `&&`

OR_OP → `||`

LE_OP → `<=`

GE_OP → `>=`

EQ_OP → `==`

NE_OP → `!=`

; → `;`

{ → `{`

} → `}`

Appendix B

Asynchronous Library Functions

NAME

init_async - initialize the asynchronous message transmittion facility

SYNOPSIS

```
#include <async.h>

int init_async(group, procid, numprocs. vtflag, simlost, nonfifo, traceflag)
            short group;
            short procid:
            short numprocs:
            short vtflag;
            double simlost;
            short nonfifo;
            short traceflag;
```

PARAMETERS

group	a positive short integer identifying the process group to which this process is a member.
procid	a short integer between 0 and numprocs-1 identifying the process number of this member of the process group.
numprocs	a short integer indicating the number of processes in this process group.
vtflag	a flag indicating whether or not vector clocks should be used during this execution. The difference in execution speeds and message sizes for most process groups is insignificant.
simlost	a double floating point number representing the probability of messages sent from this process being lost during transmittion. A value of 0.0 indicates that messages transmittion is reliable and a value of 1.0 will cause all messages sent from this process to be lost.
nonfifo	a flag indicating whether or not messages can be delivered out of order. Message order is simulated using the Miller-Park random number generator.
traceflag	a flag indicating whether or not traces of the execution should be constructed. If traceflag is true, then a file named progname.trace will be created. Refer to the async.h header file for the exact layout of the trace records.

DESCRIPTION

init_async initializes the asynchronous communication facilities provided by the libasync library. The first parameter identifies the group to which this process belongs. The group id is a short integer that identifies the set of processes within the distributed system. Processes are only allowed to communicate with other processes within their group. In addition, processes are only allowed to begin execution after **all** processes in the group have been started.

Each process in the system calls init_async to register with the process server and obtain the list of addresses for the other members of the group. Only after all members have registered are the processes allowed to proceed. If all processes have not registered within a specified timeout period, failure responses are sent to those processes that have registered and the group is removed from the registry. Later attempts to register within the same group are considered requests from a new group.

RETURN VALUES

0	Initialization failed. An indication of why should be printed to stderr.
1	Initialization was successful.

NOTES

The library containing this and other asynchronous communication related functions, along with the C header files are located in d̃ennis/public. To use them with gcc, the following command should be used.

```
gcc source -Id̃ennis/public/include -Ld̃ennis/public/lib -lasync -lm -ll
```

EXAMPLE PROGRAMS

Here are two programs that use asynchronous communication to send a simple "Hello World" string from process 0 to process 1. The receiving process then prints the number of bytes received and the received message. Notice that the message length is increased by 1 to insure the received message contains the '\0' string terminating character.

Process 0

```
#include <stdio.h>
#include <async.h>
main ()
```

```
{
   char message[32];

   /* group: 101, process: P0, 2 processes in group */
   init_async(101, 0, 2, 0, 0.0, 0);

   sprintf(message,"Hello World");
   /* send message to P1 */
   async_send(1, message, strlen(message)+1);

   /* finished */
   close_async();
}
```

Process 1

```
#include <stdio.h>
#include <async.h>
main ()
{
   char message[32];
   int msglen;

   /* group: 101, process: P1, 2 processes in group */
   init_async(101, 1, 2, 0, 0.0, 0);

   msglen = 32;
   /* receive message from P0 */
   async_recv(0, message, &msglen, 0);
   printf("received %d bytes [%s]0, msglen, message);

   /* finished */
   close_async();
}
```

SEE ALSO

async_send(2), async_recv(2), close_async(2), recv_qinfo(2), inc_vtime(2), get_vtime(2)

NAME

async_send - send an asynchronous message to another process

SYNOPSIS

```
#include <async.h>

int async_send( procid, msg, len )
        short procid;
        void *msg;
        int len;
```

PARAMETERS

procid	a short integer between 0 and numprocs-1 identifying the target process in the process group. If -1 is given as the target process identifier, the message is broadcast to all other processes in the process group.
msg	a pointer to the begining address of a message to be sent.
len	the length in bytes of the message. (Currently restricted to (MAXMSGSIZE) 10240 bytes.)

DESCRIPTION

If vector time is in use, the local component is incremented to indicate the occurrence of an event. The message pointed to by msg length len is then sent to process procid. If procid is -1, then the message is broadcast to all other processes in the process group. (See init_async(2) for a description of process groups.)

RETURN VALUES

0	The message was lost during the send process.
1	The message was successfully sent to the other process and awaits delivery.

SEE ALSO

init_async(2), async_recv(2), close_async(2), recv_qinfo(2), inc_vtime(2), get_vtime(2)

NAME

async_recv - receive an asynchronous message from another process

SYNOPSIS

```
#include <async.h>

int async_recv( procid, msg, len, waitsecs )
            short procid;
            void *msg;
            int * len;
            int waitsecs;
```

PARAMETERS

procid	a short integer between 0 and numprocs-1 identifying the transmitting process in the process group. If -1 is given as the source process identifier, the message is accepted from and process in the process group.
msg	a pointer to the begining address of a message to be sent.
len	a pointer to an integer to contain the length of the message in bytes. It is initialized to the length of the message buffer. (Currently restricted to (MAXMSGSIZE) 10240 bytes.)
waitsecs	an integer number of seconds to wait for the arrival of a message. If no message has arrived within waitsecs seconds, the function returns a -1. A value of 0 indicates that the timer should not be used and the function will wait forever.

DESCRIPTION

If vector time is in use, the local component is incremented to indicate the occurrence of an event. A message from process procid is copied to the address stored in msg. The length of the message is stored in len. If procid is -1, then the message is accepted from any process in the process group. (See init_async(2) for a description of process groups.) This option will return the next message in the order of arrival. If no message is available, the function will hang, waiting for an arrival. If no message arrives within waitsecs seconds, then the function returns with a value of -1.

RETURN VALUES

-1 No message was available for delivery within the time specified by the waitsecs parameter.

message length

The message was successfully received from the indicated process. Side effects are to store the message in the memory area pointed to by msg and to store the size of the received message in the integer pointed to by len.

SEE ALSO

init_async(2), async_send(2), close_async(2), recv_qinfo(2), inc_vtime(2), get_vtime(2)

NAME

close_async - terminate the asynchronous message transmittion facility

SYNOPSIS

#include <async.h>

int close_async()

DESCRIPTION

close_async terminates the asynchronous communication facilities initialized by a call to init_async. This function should always be called by the program using the async library. Failure to do so could leave zombie children wandering about.

RETURN VALUES

1	Termination was successful. Does not return until termination has been completed.

SEE ALSO

init_async(2). async_send(2). async_recv(2). recv_qinfo(2). inc_vtime(2). get_vtime(2)

NAME

recv_qinfo - check the status of the asynchronous message wait queues

SYNOPSIS

```
#include <async.h>

int recv_qinfo( procid )
        short procid;
```

PARAMETERS

procid	a short integer identifying the sending process from which messages should be checked. A value of -1 indicates that messages from **all** processes should be reported.

DESCRIPTION

recv_qinfo checks to see if any messages are waiting to be delivered to this process from process procid.

RETURN VALUES

0	No messages are waiting to be delivered from the indicated process.
1	Messages are waiting to be delivered from the indicated process.

SEE ALSO

init_async(2), async_send(2), async_recv(2), close_async(2), inc_vtime(2), get_vtime(2)

NAME

inc_vtime - increment the local component of the vector clock

SYNOPSIS

#include <async.h>

int inc_vtime()

DESCRIPTION

If vector clocks are being used in the asynchronous communication facilities, this function increments the local component to indicate the occurence of a significant local event.

RETURN VALUES

0	Vector clocks are not being used in this execution. See init_async(2).
local vector clock component	The value of the local component of the vector clock is returned after it has been incremented to indicate success.

SEE ALSO

init_async(2), async_send(2), async_recv(2), close_async(2), recv_qinfo(2), get_vtime(2)

NAME

get_vtime - return the current vector clock values

SYNOPSIS

```
#include <async.h>

int get_vtime( vt )
        unsigned int *vt;
```

PARAMETERS

vt	a pointer to an array of unsigned integers where the values in the vector clock should be placed.

DESCRIPTION

If vector clocks are being used in the asynchronous communication facilities, this function stores the current value of the vector clock in the array of unsigned integers pointed to by vt.

RETURN VALUES

-1	An error has occurred preventing the completion of the operation.
0	Vector clocks are not being used in this execution. See init_async(2).
1	The current values of the vector clock have been successfully placed in the vt array.

SEE ALSO

init_async(2), async_send(2), async_recv(2), close_async(2), recv_qinfo(2), inc_vtime(2)

NAME

trace - Add a local event record to a process' trace file

SYNOPSIS

#include <async.h>

int trace()

DESCRIPTION

trace is used with the asynchronous communication library event tracing facility. It creates an event record of type TRACE_LOCAL with the current vector time and adds that record to the trace information. See init_async(2) for information on initializing the tracing facilities.

RETURN VALUES

none	No values are returned from this function.

SEE ALSO

init_async(2), async_send(2), async_recv(2), recv_qinfo(2), inc_vtime(2), get_vtime(2), close_async(2)

Bibliography

[1] A.V. AHO, R. SETHI, AND J.D. ULLMAN. *Compilers:Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[2] SAMAN AMARASINGHE AND MONICA LAM. Communication optimization and code generation for distributed memory machines. *ACM SIGPLAN Notices*, 28(6):126–138, June 1993.

[3] K. APT, N. FRANCEZ, AND W. DE ROEVER. A proof system for communicating processes. *ACM Trans. Programming Languages and Systems*, 2:359–385, 1980.

[4] K.M. CHANDY AND L. LAMPORT. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[5] R. COOPER AND K. MARZULO. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, 1991.

[6] SUSAN GRAHAM DAVID BACON AND OLIVER SHARP. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[7] E. W. DIJKSTRA. A correctness proof for networks of communicating processes—A small exercise. Technical Report EWD-607, Burroughs, 1977.

[8] LORI CLARKE DOUGLAS LONG. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. Technical Report COINS 91-31, University of Massachusetts at Amherst, July 1991.

[9] SAMAN AMARASINGHE DROR MAYDAN AND MONICA LAM. Array data-flow analysis and its use in array privatization. *12th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, January 1993.

[10] R. GUPTA E. DUESTERWALD AND M. SOFFA. A pratical data flow framework for array reference analysis and its use in optimizations. *SIGPLAN Notices*, 28(6):68–77, June 1993.

[11] PAUL FEAUTRIER. Semantical analysis and mathematical programming. *Parallel and Distributed Algorithms.*, pages 309–320, October 1989. Proceedings of the International Workshop.

[12] DAVID GRIES. *The Science of Programming.* Springer-Verlag, 1981.

[13] M.S. HECHT. *Flow Analysis of Computing Programs*. Number ISBN 0-444-00210-3 in Programming Language Series. Elsevier North-Holland, New York, 1977.

[14] C.A.R. HOARE. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[15] C.A.R. HOARE. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[16] BRIAN KERNINGHAN AND DENNIS RITCHIE. *The C Programming Language*. Prentice-Hall, 1978.

[17] SYING-SYANG LIU KURT JOHMANN AND STEPHEN YAU. Context-dependent flow-sensitive interprocedural dataflow analysis. *Software Maintenance: Research and Practice*, 7:177-202, 1995.

[18] T.H. LAI AND T.H. YANG. On distributed snapshots. *Inforamtion Processing Letters*, pages 153–158, 1987.

[19] L. LAMPORT. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[20] G.M. LEVIN AND D. GRIES. A proof technique for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[21] W. S. LLOYD AND P. KEARNS. Using tracing to direct our reasoning about distributed programs. In *Proceedings of the 11th International Symposium on Distributed Computing systems*, pages 552–559, 1991.

[22] D.L. LONG AND L.A. CLARKE. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44–52, April 1989.

[23] ALAN CARLE MARY HALL, JOHN MELLOR-CRUMMEY AND RENE' RODRIGUEZ. Fiat: A framework for interprocedural analysis and transformation. *Proceedings of the 6th Workshop on Languages and Compilers for Parallel Computing*, pages 522–545, August 1993.

[24] BRAIN MURPHY MARY HALL AND SAMAN AMARASINGHE. Interprocedural parallelization analysis: A case study. *Proceedings of the 7th SIAM Conference on Parallel Processing*, pages 650–5, February 1995.

[25] VADIM MASLOV. Lazy array data-flow dependence analysis. *21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 311–25, January 1994.

[26] STEPHEN MASTICOLA. *Static Detection of Deadlock in Polynomial Time*. PhD thesis, Rutgers, Univeristy of New Jersey, 1993.

[27] STEPHEN P. MASTICOLA AND BARBARA RYDER. Non-concurrency analysis. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[28] FREIDEMANN MATTERN. Efficient algorithms for distributed snapshots and global time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993.

[29] FRIEDEMANN MATTERN. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, M. Cosnard et. al., editor, pages 215–226. Elsevier Science Publishers B. V., 1989.

[30] D. MATUSZEK. The case for the assert statement. *SIGPLAN Notices*, pages 36–37, 1976.

[31] LARRY L. PETERSON, NICK C. BUCHHOLZ, AND RICHARD D. SCHLICHTING. Preserving and using information in interprocess communication. *ACM Transactions on Computer Science*, 7(3):217–246, August 1989.

[32] MONICA LAM SAMAN AMARASINGHE, JENNIFER ANDERSON AND AMY LIM. An overview of a compiler for scalable parallel machines. *Proceedings 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 253–271, August 1993.

[33] REINHARD SCHWARZ AND FRIEDEMANN MATTERN. Detecting causal relationships in distributed computations: In search of the holy grail. Technical Report SFB 124-15/92, Department of Computer Science, University of Kaiserslautern, December 1992.

[34] KEN KENNEDY SEEMA HIRANANDANI AND CHAU-WEN TSENG. Compiling fortran d. *Communications of the ACM*, 35(8):66–80, August 1992.

[35] PHILLIP SHAFFER. Parallel implementation of real-time control programs. *Proceedings of the 27th IEEE Conference on Decision and Control*, 2508(2):1449–1454, December 1988.

[36] PHILLIP SHAFFER AND TIMOTHY JOHNSON. Data flow analysis of concurrency in a turbojet engine control program. *Proceedings of the 1988 American Control Conference*, 2503(3):1837-45, June 1988.

[37] M. SPEZIALETTI AND J. P. KEARNS. Simultaneous regions: A framework for the consistent monitoring of distributed systems. In *Proceedings of the 9th International Symposium on Distributed Computing systems*, pages 611–618, 1989.

[38] MADALENE SPEZIALETTI AND PHIL KEARNS. Efficient distributed snapshot. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[39] Sun. *SunOS Reference Manual Vol II*, 1989. C Library Function.

[40] Sun. *Sun Pascal Reference Manual*, 1991.

[41] ANDREW TANNENBAUM. *Networks*.

[42] RICHARD N. TAYLOR. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[43] ANSHUL GUPTA VIPIN KUMAR, ANANTH GRAMA AND GEORGE KARYPIS. *Introduction to Parallel Computing*. Benjamin/Cummings. 1994.

[44] Western Digital Corporation. *Ada Reference Manual*, 1983.

[45] M. YOUNG AND R. N. TAYLOR. Combining static analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

VITA

Sharon Jeanette Simmons

Born in New Orleans, Louisiana, April 21, 1962. Graduated from Hattiesburg High School in Hattiesburg, Mississippi, May 1980. At the University of Southern Mississippi, earned a B.S. in Computer Science in 1984 and earned a M.S. in Computer Science in 1991. Currently a Ph.D. candidate at the College of William and Mary.

The author is an assistant professor of Computer Science at the State University of West Georgia.