

Copyright

by

Venkataesh Velamuri Murty

1997

Controlling the Order of Events in Distributed Systems

by

Venkataesh Velamuri Murty, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 1997

Controlling the Order of Events in Distributed Systems

Approved by
Dissertation Committee:

In memory of

nana

Acknowledgments

I thank Professor Vijay K. Garg for supervising this research and providing guidance along the way. I also thank Professors Jacob A. Abraham, Craig M. Chase, Mohamed G. Gouda, and Aleta M. Ricciardi for serving in my committee.

Lastly, I thank my friends, who made my stay in Austin a pleasant one.

VENKATAESH VELAMURI MURTY

The University of Texas at Austin

August 1997

Controlling the Order of Events in Distributed Systems

Publication No. _____

Venkataesh Velamuri Murty, Ph.D.

The University of Texas at Austin, 1997

Supervisor: Vijay K. Garg

In an asynchronous distributed system, processes communicate only via messages with unbounded transmission time. Relative process speeds are arbitrary and processes do not have access to a common clock. In such systems, it is often easier to develop distributed programs when the underlying system offers certain message ordering guarantees. The main motivation of this dissertation is to provide a framework in which a user can specify the desired characteristics of the underlying system, and a protocol layer maps the underlying asynchronous system to the desired specification.

The main contribution of this dissertation is an understanding of the limitations of inhibition based protocols (where a protocol operates by delaying events) in implementing message orderings. A message ordering specification is characterized as a set of acceptable runs. We study the problem of determining which message ordering specifications can be implemented in a distributed system. Further, if a specification can be implemented, we give a technique to determine whether it can be implemented by tagging information with user messages or if it requires control messages. To specify the message ordering, we use a novel method called *forbidden*

predicates. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering, (as well as many new message orderings) can be concisely specified using forbidden predicates. We then present an algorithm that determines from the forbidden predicate the type of protocol needed to implement that specification.

We present two algorithms for message orderings. First we give an efficient algorithm for synchronous ordering, and second we present a general algorithm to generate protocols implementing a class of specifications that can be implemented by inhibition based protocols without control messages.

Lastly, we present an implementation of a generator that gives efficient protocols for specifications that can be implemented by inhibition based protocols without control messages. This provides a framework in which a user specifies the desired message ordering and the framework guarantees the specification, insulating the user from the complexities of message orderings.

Contents

Acknowledgments	v
Abstract	vi
List of Tables	xi
List of Figures	xii
Notation	xiv
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Distributed System	4
1.2.1 Message Ordering	5
1.2.2 Protocols	5
1.3 Main Contributions of the Dissertation	6
1.4 Outline	8
Chapter 2 Characterization of Message Ordering Specifications and Protocols	10
2.1 System Model	11
2.2 Protocols	14
2.3 Specifications	20

2.4	Limitations of Protocols	23
2.4.1	General Protocols	24
2.4.2	Tagged Protocols	27
2.4.3	Tagless Protocols	32
2.5	Limit Sets	35
2.6	Related Work	41
2.7	Summary	42
Chapter 3 Protocol for Message-Orderings		44
3.1	Algorithm	45
3.2	Proof of Correctness	49
3.2.1	Proof of Safety	52
3.2.2	Proof of Liveness	55
3.3	Related Work	56
3.4	Summary	57
Chapter 4 Forbidden Predicates		58
4.1	Forbidden Predicates	59
4.2	Specification Graph	62
4.3	Impossibility and Lower-Bounds	65
4.4	Related Work	71
4.5	Summary	72
Chapter 5 Algorithm to Implement Message Ordering		74
5.1	Extensions to Forbidden Predicates	74
5.2	Algorithm for a Two Clause Predicate	77
5.2.1	Detection	78
5.2.2	Safety	79
5.3	Discussion of the General Algorithm	80
5.4	Proof of the Correctness of the Algorithm	83

5.5	Discussion	87
5.5.1	Garbage Collection	87
5.5.2	Induction Argument	89
5.6	Related Work	91
5.7	Summary	92
Chapter 6 Implementation		93
6.1	Interface to the Protocol	93
6.2	Code Generator	94
6.3	Input	97
6.4	Output	99
Chapter 7 Conclusion and Future Work		104
7.1	Summary and Discussion	104
7.2	Future Work	105
Bibliography		112
Vita		117

List of Tables

3.1	Protocol to send a message from a bigger to a smaller process. . . .	47
3.2	Protocol to send a message from a smaller to a bigger process. . . .	48

List of Figures

2.1	Illustration of causal past with respect to a process.	13
2.2	Inhibitory protocol to implement FIFO.	15
2.3	Knowledge of concurrent events.	18
2.4	Differences in causality relation between system's and user's views. .	21
2.5	Prefixes of \mathcal{H}	24
2.6	A cut belonging to \mathcal{H}_{gn}	25
2.7	Numbering scheme for an element $\mathcal{H} \in \mathcal{X}_{gn}$	26
2.8	Constructing the next prefix given \mathcal{H}^i	28
2.9	Construction of \mathcal{G} given \mathcal{H}^i for process j	29
2.10	Construction of \mathcal{G} given \mathcal{H}^i for process j	33
2.11	Construction of \mathcal{H} from $(\mathbb{H}, \triangleright)$	39
3.1	Asymmetric property of an algorithm implementing SYNC.	46
3.2	A bigger process sending a message to a smaller process.	47
3.3	A smaller process sending a message to a bigger process.	48
3.4	Protocol messages to implement SYNC.	50
4.1	Construction of a run using a forbidden predicate.	67
5.1	Detection of different stages of the predicate.	78
5.2	Pseudo-code for an algorithm implementing WC, EC, and UC. . . .	82
5.3	Pseudo-code for an algorithm implementing WC, MEC, and MUC. .	90

6.1	Definition of the class <code>Msg</code>	94
6.2	Send and Deliver functions.	95
6.3	Architecture of the implementation.	96
6.4	An example input file.	98
6.5	Syntax for writing <code>Filter</code> and <code>Predicate</code>	98
7.1	A multicast message $\{s, r_a, r_b, r_c\}$	106
7.2	A collated message $\{s, r_i, r_f\}$	106
7.3	Global event $\{x.a, x.b, x.c, x.d\}$ to implement \mathbb{X}_{sync}	110

Notation

Z	Set of process identifiers
\mathcal{H}, \mathcal{G}	System run
$(\mathbb{H}, \triangleright)$	System run \mathcal{H} as viewed by user
H_i	Events in process i in a system executing \mathcal{H}
H	Events in a system executing \mathcal{H} , $H = \cup H_i$
\mathbb{H}	Events in user's view in a system executing \mathcal{H}
F, G, H	Event sets
\rightarrow	Causality relation in system model
\prec	Projection of \rightarrow to a process
\triangleright	Causality relation with respect to the user
$\triangleright \cup \equiv$	$\triangleright \cup \equiv$
M	Set of all messages
M_{ij}	Set of messages from process i to j
$Msg(H)$	Set of messages in the event set H i.e., $x \in Msg(H) \Leftrightarrow (x.s^* \in H) \vee (x.s \in H) \vee (x.r^* \in H) \vee (x.r \in H)$
$Msg(h)$	Message corresponding to the event h i.e., $x \equiv Msg(h) \Leftrightarrow (x.s^* \equiv h) \vee (x.s \equiv h) \vee (x.r^* \equiv h) \vee (x.r \equiv h)$
x, y, z, x_i	Free variable used to represent messages
$x.s^*$	Invocation of the message x
$x.s$	Send of the message x
$x.r^*$	Receive of the message x

$x.r$	Delivery of the message x
a, b, c	Message instances
f, g, h	Events
p, q	Stands for either s or r
i, j, k, l	Indices
n	Number of process
m	Number of free variables in a predicate or number of vertices in a predicate graph
E	Edge set
V	Vertex set
B	Forbidden Predicate
\mathcal{P}	Protocol
\mathcal{X}	Set of all partial orders \mathcal{H}
\mathcal{Y}, \mathcal{Z}	Subset of \mathcal{X}
\mathbb{X}	Set of all partial orders ($\mathbb{H}, \triangleright$)
\mathbb{Y}, \mathbb{Z}	Subset of \mathbb{X}

Chapter 1

Introduction

Controlling the order of events in a distributed system is necessary to achieve some desired behavior from the underlying distributed system. The goals of the research presented here are to provide a framework such that the user can specify the desired characteristics of a distributed system succinctly and automatically generate an efficient protocol to map the underlying system to the desired system.

1.1 Motivation

The motivation for the research is:

- First, to understand the limitations of inhibition based protocols in implementing message ordering specifications
- Second, to provide a framework for the user to specify the characteristics of the desired distributed system and automatically generate a protocol that maps the asynchronous distributed system to the specification.

In this section we explore different areas in distributed systems where such a framework is useful.

Software Development

Let us consider a system with two processes p_1 and p_2 . If p_1 sends two messages x and y to p_2 , then p_2 can receive the two messages in two possible ways, either x before y or y before x . Similarly, if p_1 sends k messages then there are $k!$ possible computations. If we impose the restriction that, if x is sent before y then x is received before y (in other words, FIFO ordering) the number of possible outcomes reduces to one. The two cases discussed, one in which the system is completely asynchronous and the other in which FIFO ordering is imposed, are extremes. In general, we may be interested in the case which allows some manageable extent of non-determinism. An obvious question is: ‘what is meant by manageable?’. This is dependent on the problem in question. In addition, during the software implementation process the programmer may be interested in increasing the non-determinism gradually. Thus, we are interested in a system that provides us with the whole range of non-determinism (between the two extremes), and in where the user can set the non-determinism to the required level. It is very beneficial if the system can provide a framework to avoid computations which either the programmer does not care about or wants to postpone until later.

Software Debugging

Selective control of non-determinism is helpful in debugging distributed programs. The process of debugging consists of monitoring a program in order to learn something about its behavior. Often the observed behavior of the program differs from what is expected. When this occurs, a debugger can be used to investigate the discrepancy. In the process of debugging a distributed program, non-determinism can result in a large number of possible runs. The programmer may not be interested in all the runs. The programmer may conjecture that unexpected behavior occurs only in a particular class of computations, for example those that exhibit the property - “when functions f and g execute concurrently”. Thus, the programmer is interested

in a framework in which he can specify the set of computations of interest and avoid the unnecessary cases.

Software Testing

In the process of software regression test, we are interested in test cases that are repeatable, producing the same output for a given input. In a distributed system, there is an additional variable - the non-determinism that plays a part in the output. Thus, there should be an easy way to control non-determinism to create repeatable test scenarios. In addition, we are interested in testing under different test environments – for instance, environments where all messages are causally ordered, or all messages are synchronous. Thus, a distributed system should provide a framework to facilitate simulating different environments.

Message Orderings

A fair amount of research has been done in deriving efficient algorithms to implement different message orderings, for example, causal ordering, flush channel primitives, synchronous ordering, marker message algorithms and broadcast/multicast algorithms. This work unifies many of the orderings studied in the literature.

Partial Order Services

Current applications that need to communicate objects (i.e., packets, frames) usually choose between a fully ordered service such as that currently provided by TCP [38] and one that does not guarantee any ordering such as that provided by UDP [38]. For some applications a partial order service [16] is more appropriate.

A motivating application for a partial order service is the emerging area of multimedia communications. These applications have a high degree of tolerance for less-than-fully-ordered data transport as well as data loss. A second application that could benefit from a partial order service involves remote and distributed databases.

Consider the case where a database user transmitting queries to a remote server expects records to be returned in some order, although not necessarily in total order. In effect, a partial order extends the service level from two extremes – ordered and unordered – to a range of discrete values encompassing both extremes and all possible partial orderings in between.

Separation of Concerns

Many protocols such as global snapshot algorithms [11, 20, 27], check-pointing and rollback recovery [19, 25, 24], and deadlock detection [9] require special messages to find consistent-cuts in a computation. These protocols require some form of inhibition of the special messages in order to guarantee correctness.

1.2 Distributed System

The model we use of a distributed system consists of a finite set of processes p_1, \dots, p_n , communicating using reliable messages. A computation is characterized by a finite sequence of events on each process. In general, we assume a system with unbounded message delivery and without a global clock. Two events in a computation are related under the “*happened-before*” relation [26] if one of the following holds.

- The second event happened after the first event in the same process.
- One event is a send of a message, and the second is the reception of the same message.
- If there exists a third event, where the first event happened-before the third event, and the third event happened-before the second event.

1.2.1 Message Ordering

A distributed computation or a run describes an execution of a distributed program. At an abstract level, a run can be defined as a partially ordered set $(\mathbb{H}, \triangleright)$, where \mathbb{H} is the set of events in the system and \triangleright the “happened before” relation between events. It is often easier to develop distributed programs when the partially ordered set $(\mathbb{H}, \triangleright)$ is guaranteed to satisfy certain message ordering properties. For example, many distributed algorithms work correctly only in the presence of FIFO channels. This guarantee on the ordering of messages is either provided explicitly – by communication primitives such as causal ordering [5] and logically synchronous ordering [10, 14]; or is built into the algorithm itself – as with global snapshot [20] and recovery algorithms [19].

A message ordering specification is characterized as the set of acceptable runs, that is, a subset of \mathbb{X} , where \mathbb{X} is the set of all runs. For example, a system satisfying causal ordering can be viewed as the set of runs, say \mathbb{X}_{co} , such that for all runs in \mathbb{X}_{co} , and for all pairs of messages, $(s_1 \triangleright s_2) \Rightarrow \neg(r_2 \triangleright r_1)$, where s_j is the send of a message and r_j is its corresponding receive.

1.2.2 Protocols

A protocol maps the underlying system to the desired system. The mapping is achieved by inhibition of events. A protocol for a distributed system specifies for each process at every stage of the computation, the set of events that are enabled. Thus, a protocol has the power to disable an event until the occurrence of prerequisite events. For example, in the implementation of FIFO, a message is delayed until all messages with the same source and destination sent earlier have been received.

To facilitate the discussion of inhibition, an event in a user’s view is broken into two underlying system events: the request of the event and the execution of the event. For example, the receive event in a user’s view is broken into the receive and the delivery of the message, and similarly the send event into the invocation

and the send of the message.

1.3 Main Contributions of the Dissertation

The contributions of this work can be divided into three categories – first, the theory of message ordering, second, algorithms for message ordering, and lastly automatic generation of efficient algorithms for message orderings.

First, in the theory of message ordering the dissertation’s contributions are:

- A new characterization of inhibition based protocols and classification of them into three classes, namely protocols that require knowledge of the concurrent past, protocols that require knowledge of the causal past, and protocols that require knowledge of the local past.
- A definition of a message ordering specification as a set of valid runs. In this abstract setting we derive:
 - the necessary and sufficient conditions for the existence of an inhibition based protocol, and
 - the knowledge required by the protocol to implement the specification.

An asynchronous distributed system can be viewed as a set of all possible runs \mathbb{X} , and a message ordering specification a subset of \mathbb{X} .

- In general \mathbb{X} is an infinite set, thus we need a finite representation for “useful” specifications. We present a method called *forbidden predicates* and present:
 - an efficient algorithm to determine the existence of an inhibition based protocol, and
 - the knowledge required by the protocol to implement the specification.

This dissertation answers some fundamental questions in the area of message orderings. For example, consider the algorithm for causal ordering by Raynal, Schiper and Toueg [32]. In their algorithm a process P_i tags a message with the

matrix m where $m[j, k]$ is the knowledge of process P_i about the messages sent from P_j to P_k . It is natural to ask whether the message ordering can be further restricted by sending higher-levels of knowledge (for example, by using three dimensional matrices: what P_i knows that P_j knows about messages sent from P_k to P_l). It is an easy consequence of the results of this dissertation that no additional tagging of information can restrict the message ordering further.

Similarly, we show that there does not exist a protocol to implement any message ordering more restrictive than synchronous ordering. That is, if the time diagram of an invalid run can be drawn such that all messages are vertical then such a specification is not implementable.

Although, there has been a fair amount of research in the area of message orderings, neither a succinct representation for all the message orderings nor a formal treatment to study the relationship between the orderings has been done. We present a new method, called *forbidden predicates*, to specify these message orderings. Informally, a forbidden predicate is a conjunction of causality relationships between events, and a run is valid if there does not exist a set of events that makes the forbidden predicate true. Forbidden predicates can be used to describe a large class of message specifications. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering, as well as others, can be concisely specified using forbidden predicates.

Second, in algorithms for message orderings, we explore two main issues – first, efficient algorithms for synchronous, causal, and asynchronous orderings, and second generalization of the protocols for a class of specifications that can be implemented by tagging. The two main contributions are:

- An efficient algorithm to implement synchronous ordering.
- A general algorithm to generate protocols implementing a class of specifications that can be implemented by inhibition based protocols with only knowledge of the causal past.

The protocol for synchronous ordering is a more efficient algorithm than the existing ones, such as Bagrodia’s rendezvous [4] and the algorithm by Soneoka and Ibaraki [36]. The algorithm presented results in fewer messages with the same or quicker response time.

The general algorithm generates protocols for the existing message ordering specifications like FIFO, causal ordering, F-channel primitives, and the orderings induced among the user or control messages in many of the protocols such as global snapshot algorithms [11, 20, 27], check-pointing and rollback recovery [19, 25, 24], and deadlock detection [9]. The work can be easily extended to include other message orderings like FIFO Broadcast and CBCAST.

Lastly, we present an implementation of a protocol generator that generates efficient (with respect to tagged information) protocols for specifications that can be implemented by inhibition based protocols with only knowledge of the causal past. This provides a communication framework in which a user specifies the message ordering for a run of the distributed program and the framework guarantees the specification, insulating the user from the complexities of message orderings.

1.4 Outline

This dissertation consists of seven chapters. The main focus of the dissertation is a distributed system characterized by point-to-point messages and protocols that operate by delaying events. The protocols are classified into three types depending on the amount of knowledge required: local, causal, or concurrent.

Chapter 2 presents a formal specification of a desired distributed system and defines inhibition based protocols. We define the three classes of protocols and present the necessary and sufficient conditions for the existence of a protocol in a class for a given specification.

Chapter 3 discusses implementation of three message orderings: asynchronous ordering, causal ordering, and synchronous ordering. In this chapter, we also present

an efficient protocol to implement synchronous ordering.

Chapter 4 introduces a succinct representation called forbidden predicates for a class of message orderings. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and synchronous ordering as well as others can be concisely specified using forbidden predicates. We present an algorithm for the existence of a protocol in a class for a given forbidden predicate.

Chapter 5 studies the issues for automatic generation of efficient protocols for a forbidden predicate.

Chapter 6 describes the prototype design for an automatic generator of protocols. We show that the protocols generated are as efficient as the protocols in the literature for some of the extensively studied message orderings.

Chapter 7 examines some extensions to the idea of a global event (a message can be thought of as a global event with a send and a receive) to multicast messages and other generalizations.

Chapter 2

Characterization of Message Ordering Specifications and Protocols

In this chapter, we are interested in characterizing message ordering specifications and protocols that operate by delaying events. Usually an event from the user's view is equivalent to two underlying system events: the event request and the event execution. For example, to implement causal ordering, the receive events of the user messages are implemented as two underlying system events: receive of the message and the delivery of the message. Therefore, we differentiate between the two views by defining the system's view of a run and the user's view of the same run. In this chapter, each *user* event h , like send of a message or receive of a message, is characterized by two *system* events, that is h^* and h , where h^* represents the request of the event and h the execution.

2.1 System Model

A distributed system consists of a set of n processes communicating using messages from a message set. The basic entity in our model is a message.

Definition 2.1 A message x consists of four system events. They are *invocation* event $x.s^*$, *send* event $x.s$, *receive* event $x.r^*$ and *delivery* event $x.r$. We say, $Msg(h)$ is a message x if h is either $x.s^*$, $x.s$, $x.r^*$, or $x.r$.

The *invocation* event represents the request by the origin process to send the message and the *send* event the actual send of the message. Similarly, the *receive* event represents the destination process's receiving the message and the *delivery* event when the destination process processes the message.

Definition 2.2 If $Z = \{1, 2, \dots, n\}$ is a set of natural numbers used to identify the processes in a system, then

$$M = \bigcup_{(i,j) \in Z \times Z} M_{ij},$$

is the set of messages, where M_{ij} is the set of messages from process i to j . The set $M_{ii} = \emptyset$ for any $i \in Z$.

The set M_{ij} represents the set of all messages that can be sent by process i to process j . Thus, in any run the messages sent by process i to process j is a subset of M_{ij} . The restriction $M_{ii} = \emptyset$ states that a process does not send messages to itself.

A cut¹ defines the state of the system, and the state is a function of the events executed by each process.

Definition 2.3 Let H_i be the event set of process i , such that

$$H_i \subseteq \{x.s^*, x.s : x \in M_{ij} \text{ for all } j\} \cup \{x.r^*, x.r : x \in M_{ki} \text{ for all } k\}.$$

A finite sequence of events H_i is called the *local state* of i .

Definition 2.4 Given an n -vector of local states, $\mathcal{H} = (H_1, H_2, \dots, H_n)$, where H_i is the local state of i for each i , event h happened-before event h' , denoted as $h \rightarrow h'$,

¹In this dissertation we use cut to mean consistent cut

if any one of the following conditions is true:

1. h and h' are both events in H_i for some i , and h is before h' in the sequence H_i , or
2. h is the send of a message, i.e., $x.s$ and h' is the corresponding receive, i.e., $x.r^*$ for some message $x \in M$, or
3. the event h *happened-before* event h'' and the event h'' *happened-before* the event h' , for some event $h'' \in H$, where $H = \cup_i H_i$.

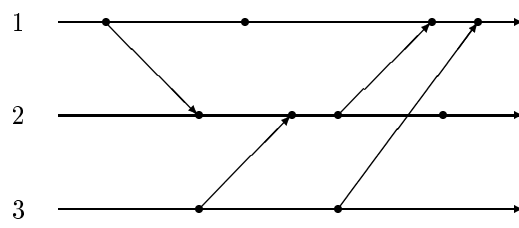
A *cut* satisfies the usual notions of a computation in a distributed system, that is, it is a partial order, it has no spurious messages, and it includes the execution of an event only if it has been requested by the user.

Definition 2.5 Let $Z = \{1, \dots, n\}$ be the processes in the system and M the message set. An n -valued vector of local state \mathcal{H} is a *cut* if the events in \mathcal{H} and the happened-before relation on the set of events satisfy the following conditions:

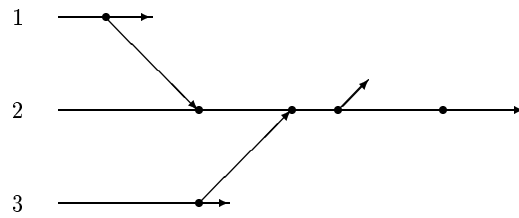
1. There are no spurious messages, i.e. if the receive event $x.r^*$ is in \mathcal{H} then the corresponding send $x.s$ is in \mathcal{H} for any message $x \in M$.
2. For any local state H_i , if a send event $x.s$ is in H_i then the corresponding invocation event $x.s^*$ is in H_i for any message $x \in M$. Similarly, if a delivery event $x.r$ is in H_i then the corresponding receive event $x.r^*$ is in H_i for any message $x \in M$.
3. The happened-before relation on the set of events in \mathcal{H} is a partial order, that is, there are no two events h and h' such that $h \rightarrow h'$ and $h' \rightarrow h$.

Definition 2.6 A *prefix* of a cut \mathcal{H} is any cut \mathcal{G} , such that G_i is a subsequence of H_i for all $i \in Z$.

A prefix of interest is the causal past of a cut \mathcal{H} with respect to a given process i (denoted by $CausalPast_i(\mathcal{H})$). Figure 2.1 shows the causal past of the cut \mathcal{H} with respect to process 2. Intuitively, the causal past with respect to a process i consists of



(a) $\text{Cut } \mathcal{H}$



(b) $\text{CausalPast}_2(\mathcal{H})$

Figure 2.1: Illustration of causal past with respect to a process.

all the events that are followed by some event in process i . Let $\mathcal{G} = \text{CausalPast}_i(\mathcal{H})$, then

1. $G_i = H_i$, and
2. $\forall j \neq i : g \in G_j \equiv (\exists h \in H_i : g \rightarrow h)$.

Definition 2.7 A *distributed system* is a 3-tuple (Z, M, \mathcal{X}) , where

1. $Z = \{1, \dots, n\}$ is a set of process identifiers,
2. M is a message set, and
3. \mathcal{X} is the set of all cuts over Z and M , i.e. $\mathcal{X} = \{ (\mathcal{H}, \rightarrow) : (\mathcal{H}, \rightarrow) \text{ is a cut} \}$.

Given a cut $\mathcal{H} \in \mathcal{X}$ we use the following notation to represent the messages that have not been requested by process i , messages that have been requested but not yet sent, messages that have been sent by the process i and are in transit, and finally the messages that have been received but not yet delivered.

$$I_i(\mathcal{H}) = \{ x.s^* : (x.s^* \notin H_i) \wedge (x \in M_{ik}) \}.$$

$$S_i(\mathcal{H}) = \{ x.s : (x.s^* \in H_i) \wedge (x.s \notin H_i) \}.$$

$$R_i(\mathcal{H}) = \{ x.r^* : (x.r^* \notin H_i) \wedge (\exists k : (x \in M_{ki}) \wedge (x.s \in H_k)) \}.$$

$$D_i(\mathcal{H}) = \{ x.r : (x.r^* \in H_i) \wedge (x.r \notin H_i) \}.$$

2.2 Protocols

In this section we define inhibitory protocols. Informally, an inhibitory protocol specifies that an event may be delayed until the occurrence of prerequisite events. For example, in the implementation of FIFO, a message is delayed until all messages sent earlier have been delivered. In Figure 2.2, the protocol enables the event r_2 only after the event r_1 has been executed.

Definition 2.8 Given a distributed system (Z, M, \mathcal{X}) . An *inhibition* based protocol \mathcal{P} specifies a set of enabled events at each process for a given cut $\mathcal{H} \in \mathcal{X}$. Thus,

$$\mathcal{P} = \{ (P_1(\mathcal{H}), P_2(\mathcal{H}), \dots, P_n(\mathcal{H})) : \mathcal{H} \in \mathcal{X} \},$$

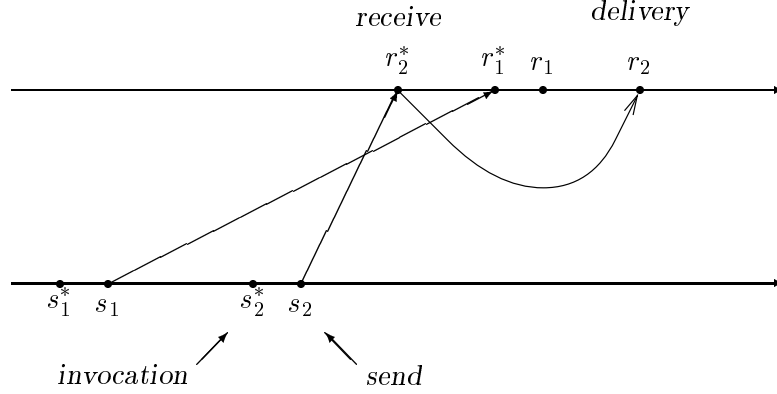


Figure 2.2: Inhibitory protocol to implement FIFO.

is a vector of enabled event sets for each cut, where $P_i(\mathcal{H})$ is the set of enabled events in process i after the execution of the cut \mathcal{H} and

$$P_1(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup S_i(\mathcal{H}) \cup R_i(\mathcal{H}) \cup D_i(\mathcal{H}).$$

Now we present some conditions to be satisfied by the vector of enabled event sets. The protocols do not have control over star-events. Clearly, a protocol cannot disable a user from requesting the execution of a message that has not been sent. Thus, we have

$$P_i(\mathcal{H}) \cap I_i(\mathcal{H}) = I_i(\mathcal{H}).$$

Similarly, a protocol cannot disable the receive of a message that has been already sent and is in transit. Thus

$$P_i(\mathcal{H}) \cap R_i(\mathcal{H}) = R_i(\mathcal{H}).$$

A protocol can disable or enable the send event and delivery event of a message if the invocation or the receive has been executed, respectively. Therefore,

$$I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \subseteq P_i(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \cup C_i(\mathcal{H}),$$

where $C_i(\mathcal{H}) = S_i(\mathcal{H}) \cup D_i(\mathcal{H})$.

(**Controllability**)

Notation: The notation C represents controllable events. We use C_i to represent controllable events in a process i . When the subscript is absent, say $C(\mathcal{H})$, then

we are referring to the union of all C_i s, that is, $C(\mathcal{H}) = \bigcup_i C_i(\mathcal{H})$. We follow this convention for the sets H, P, I, S, D, R , and C , that is, $H = \bigcup_i H$, $P(\mathcal{H}) = \bigcup_i P_i(\mathcal{H})$, and so on.

We define the set of *cuts*, \mathcal{X}_P , possible under the protocol \mathcal{P} inductively, based on the events enabled. The base case is the null cut \mathcal{H}_\emptyset in which $H = \emptyset$ belongs to the set \mathcal{X}_P , since this cut is possible even if the protocol does not enable any event. Let the cut \mathcal{H} be possible under the protocol. If some of the processes simultaneously execute an event enabled in their process, then the resulting cut also is possible under the protocol.

Definition 2.9 Given a protocol \mathcal{P} the *protocol set* \mathcal{X}_P is defined inductively using the following rules:

1. $\mathcal{H}_\emptyset \in \mathcal{X}_P$.
2. Let $\mathcal{H} \in \mathcal{X}_P$, then $\mathcal{G} \in \mathcal{X}_P$, where
 - (a) H_i is a prefix of G_i and they differ by at most one event, and
 - (b) $G_i \subseteq H_i \cup P_i(\mathcal{H})$.

In the next lemma, we show that \mathcal{G} is a *cut*, that is, it satisfies the three conditions of a cut.

Lemma 2.1 Let \mathcal{H} be a cut, and $(P_1(\mathcal{H}), P_2(\mathcal{H}), \dots, P_n(\mathcal{H}))$ a vector of enabled event sets, such that, for all i

$$I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \subseteq P_i(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \cup C_i(\mathcal{H}).$$

Then \mathcal{G} is a cut, where

1. H_i is a prefix of G_i and they differ by at most one event, and
2. $G_i \subseteq H_i \cup P_i(\mathcal{H})$.

Proof Outline: Clearly, \mathcal{G} satisfies the three conditions of a cut, that is,

1. \mathcal{G} is a partial order, since \mathcal{H} is a partial order, and if $g \in G \cap P(\mathcal{H})$ then $\exists g' \in G : g \rightarrow g'$,

2. $x.r^* \in G \Rightarrow x.s \in G$, since a receive event is added only if H contains the send event, and
3. $x.s \in G \Rightarrow x.s^* \in G$ and $x.r \in G \Rightarrow x.r^* \in G$, by definition of \mathcal{S} and \mathcal{R} . \square

It is desirable that any protocol allows the system to progress to satisfy the liveness property. That is, if a user requests a message then it is eventually sent and delivered. In other words, we want the protocol to eventually execute a cut \mathcal{H} such that, $S(\mathcal{H}) \cup R(\mathcal{H}) \cup D(\mathcal{H}) = \emptyset$, that is, all the messages requested by the user have been sent and delivered and there are no pending events. Thus, the protocol at each stage enables at least one of the pending events if the pending set is not empty. This condition can be formally stated as

$$R(\mathcal{H}) \cup C(\mathcal{H}) \neq \emptyset \Rightarrow P(\mathcal{H}) \cap (R(\mathcal{H}) \cup C(\mathcal{H})) \neq \emptyset. \quad \textbf{(Liveness)}$$

Consider the case when the above condition is not satisfied. If the user does not request any more messages, the system cannot make any progress and the pending events are never executed.

Next, we classify protocols based on the extent of information exchange possible among the processes. Consider first a protocol which allows processes to exchange information only using user messages. Then a process is limited to the causal past and intuitively, the class of such protocols can be implemented by tagging information to user messages. Next consider, a protocol which does not allow any information exchange using either user or control messages. Then a process has to enable/disable events based only on its local history. Such protocols belong to the class of tag-less protocols. Finally, consider a protocol that allows processes to exchange information using both control messages and user messages. Then a process is capable of delaying events based on events that appear concurrent, when events associated with the control messages are deleted. For example, in Figure 2.3, process i knows about the events $x.s^*$ and $x.s$ although they appear concurrent when the control message has been deleted.

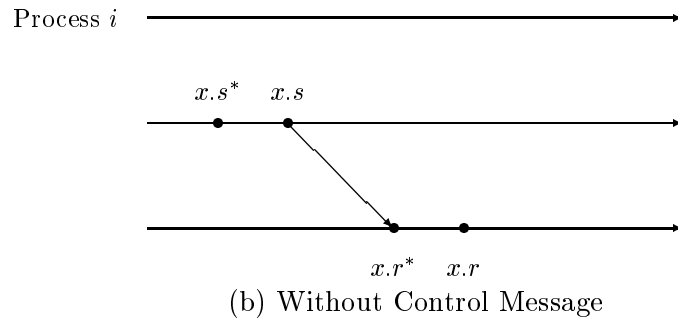
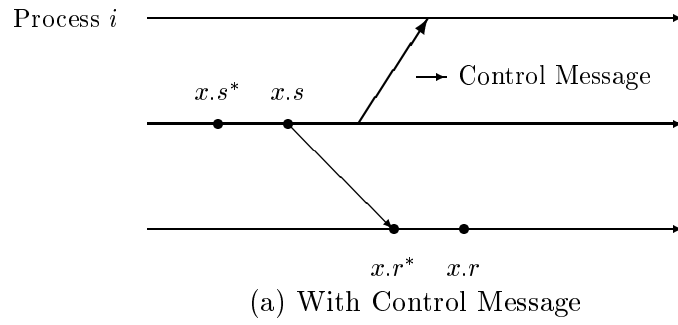


Figure 2.3: Knowledge of concurrent events.

Formally, the types of protocols and the condition satisfied by each type are:

General Protocols

The class of **general** protocols characterizes the environment where an action by a process can be known instantaneously to all processes in the group. Thus, each process enables and disables events based on the knowledge of both causal and concurrent events, but a process cannot enable or disable events based on

- future events, or
- timing of the events.

Therefore, a process takes the same action in any two executions, if the partial orders are the same.

Later, we show that if there exists a **general** protocol for a given specification then there exists an inhibitory protocol using control and user messages that implements the specification.

Tagged Protocols

The class of **tagged** protocols characterizes the environment where an action by a process can be known only in its causal future. Thus, each process enables and disables events based on the knowledge of events in the causal past. Therefore, if in two different executions, the causal past with respect to a process i , that is $CausalPast_i(\cdot)$, is the same then the action taken by the process i in the two cases is same. This can be formally stated as,

$$CausalPast_i(\mathcal{H}) = CausalPast_i(\mathcal{G}) \Rightarrow P_i(\mathcal{H}) = P_i(\mathcal{G}). \quad (\mathbf{Causal})$$

Tagless Protocols

The class of **tagless** protocols characterizes the environment where an action by a process can be known only in its local future. Thus, each process enables and

disables events based on the knowledge of local events. These protocols cannot tag information to the user messages and cannot use control messages. The condition satisfied by a **tagless** protocol is,

$$H_i = G_i \Rightarrow P_i(\mathcal{H}) = P_i(\mathcal{G}). \quad (\mathbf{Local})$$

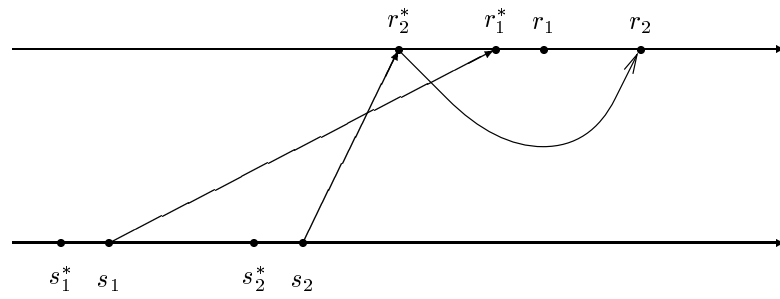
The condition states that if the local history is the same then the action taken by the process is the same.

The above conditions are used to describe the three classes of protocols we are interested in studying. For instance, the class of **general** protocols models the behavior of protocols with control messages in the absence of synchronized clocks or a global clock. Such a protocol cannot differentiate between two cuts that have the same partial order relation but may differ in physical global time.

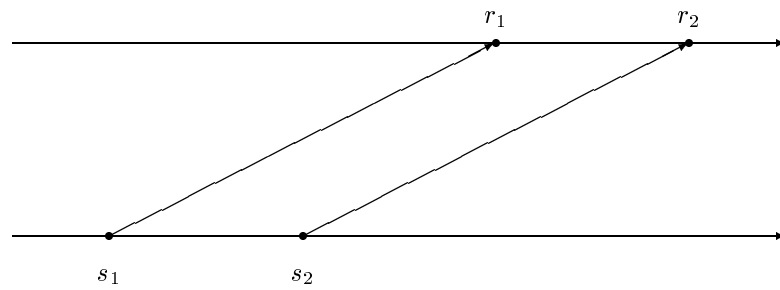
2.3 Specifications

A specification is the behavior as desired by the user. For example, a particular run may or may not be desirable. In this section, we expand on the concept of user's view and formally define a specification.

A user is interested in the send and delivery of a message and the order relation among them, rather than the invocation and receive events. For example, causal ordering is stated in terms of the relation between the send and delivery events. Thus, the causality relation between two events from the user's view can be different than the relation from the system's view. Figure 2.4 illustrates the difference in a system that implements FIFO ordering among the messages. In the system's view the event s_2 happened causally before the event r_1 , whereas from the user's view s_2 did not happen before the event r_1 . Thus, we define a relation from the system's view to the user's view of a cut, which is a projection of the events with the invocation and receive events removed.



(a) System's View



(b) User's View

Figure 2.4: Differences in causality relation between system's and user's views.

Definition 2.10 $UserView(\mathcal{H})$, a projection of the cut \mathcal{H} , is a partial order, denoted as a tuple $(\mathbb{H}, \triangleright)$, where

$$\mathbb{H} = \{ h : h \in H \wedge (h \text{ is a } \textit{send} \text{ or a } \textit{delivery} \text{ event}) \},$$

and \triangleright is the order relation on \mathbb{H} . For the *projected cut* $(\mathbb{H}, \triangleright)$, $h \triangleright h'$ if and only if

1. $\exists k$ such that $h, h' \in H_k$ and $h \rightarrow h'$, or
2. $\exists x \in M$, such that $x.s = h$ and $x.r = h'$, or
3. $\exists g \in H$ such that $h \triangleright g$ and $g \triangleright h'$.

The projected cut also satisfies the usual notion of a cut, that is there are no spurious messages and the relation \triangleright is a partial order.

A distributed run is a execution of a program with all messages invoked have been delivered, since we assume reliable message delivery.

Definition 2.11 A *run* $(\mathbb{H}, \triangleright)$ is a projection of a cut \mathcal{H} such that

1. $(\mathbb{H}, \triangleright) = UserView(\mathcal{H})$, and
2. all messages invoked in the cut \mathcal{H} have been delivered, that is, $x.s^* \in H \Leftrightarrow x.r \in H$.

We can represent the distributed system (Z, M, \mathcal{X}) from a user's view as a 3-tuple (Z, M, \mathbb{X}) , where \mathbb{X} is the set of all runs possible in user's view. That is,

$$\mathbb{X} = \{ (\mathbb{H}, \triangleright) : \exists \mathcal{H} \text{ such that } (\mathcal{H} \in \mathcal{X}) \wedge ((\mathbb{H}, \triangleright) = UserView(\mathcal{H})) \wedge (x.s^* \in H \Leftrightarrow x.r \in H) \}.$$

In the rest of the dissertation we use \mathbb{X} to represent the above set.

Definition 2.12 A *specification* \mathbb{Y} is a subset of \mathbb{X} .

A protocol \mathcal{P} is characterized by the set of cuts $\mathcal{X}_{\mathcal{P}}$ that are possible under the protocol. We say that a protocol \mathcal{P} guarantees safety if the projection of a cut $\mathcal{H} \in \mathcal{X}_{\mathcal{P}}$ is valid in the user's view. In other words, a protocol \mathcal{P} guarantees safety if

for all \mathcal{H} in the set \mathcal{X}_P satisfying the condition $x.s^* \in H \Leftrightarrow x.r \in H$, the projection $UserView(\mathcal{H})$ belongs to \mathbb{Y} .

A protocol \mathcal{P} is characterized by the set of cuts \mathcal{X}_P on a system (Z, M, \mathcal{X}) . The protocol can be similarly characterized in the user's view of the system, as a set of runs \mathbb{X}_P , that is

$$\mathbb{X}_P = \{ (\mathbb{H}, \triangleright) : \exists \mathcal{H} \text{ such that} \\ (\mathcal{H} \in \mathcal{X}_P) \wedge ((\mathbb{H}, \triangleright) = UserView(\mathcal{H})) \wedge (x.s^* \in H \Leftrightarrow x.r \in H) \}.$$

Thus, we can state that a protocol \mathcal{P} guarantees safety for the specification \mathbb{Y} , if and only if

$$\mathbb{X}_P \subseteq \mathbb{Y}. \quad \textbf{(Safety)}$$

In summary, given a system (Z, M, \mathcal{X}) and a specification \mathbb{Y} we say a protocol \mathcal{P} implements the specification if and only if

$$\forall \mathcal{H} \in \mathcal{X} \quad : \quad R(\mathcal{H}) \cup C(\mathcal{H}) \neq \emptyset \Rightarrow P(\mathcal{H}) \cap (R(\mathcal{H}) \cup C(\mathcal{H})) \neq \emptyset, \quad \text{and} \\ \mathbb{X}_P \subseteq \mathbb{Y}.$$

2.4 Limitations of Protocols

In this section we explore the limitations of each type of protocol. We answer questions of the form, 'If protocol \mathcal{P} is a **tagless** protocol, then does \mathcal{H} necessarily belong to the set \mathcal{X}_P ?'. These questions provide us with insight into the type of protocol necessary to implement the desired specification. For example, if the cut \mathcal{H} is undesirable and $\mathcal{H} \in \mathcal{X}_P$, then \mathcal{P} cannot implement the specification.

Given a protocol \mathcal{P} , the set of possible cuts under the protocol \mathcal{X}_P is defined inductively. Thus, the proof of the inclusion of a cut \mathcal{H} in the set \mathcal{X}_P is also done inductively. Given a cut \mathcal{H} we construct a series of prefixes $\mathcal{H}^0, \mathcal{H}^1, \dots, \mathcal{H}^i, \dots$, such that \mathcal{H}^i is a prefix of \mathcal{H} and $H^i \subset H^{i+1} \subseteq H$, and \mathcal{H}^0 is an empty cut, that is $H^0 = \emptyset$.

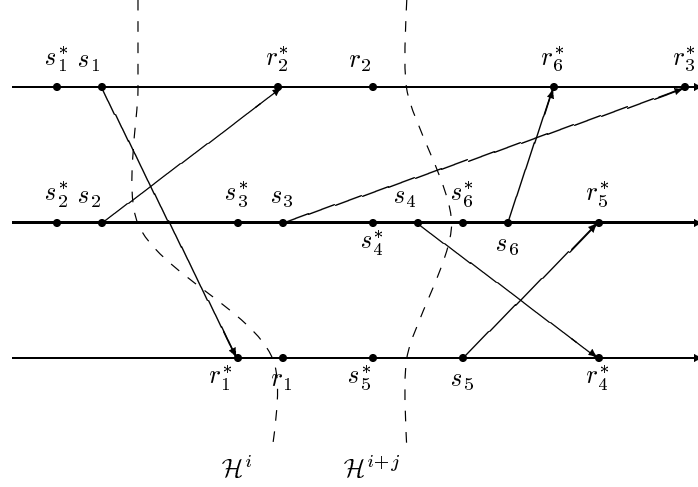


Figure 2.5: Prefixes of \mathcal{H} .

Base Case: The base case, that is $\mathcal{H}^0 \in \mathcal{X}_P$ follows, since \mathcal{H}^0 is an empty cut.

Inductive Case: We have to show that if $\mathcal{H}^i \in \mathcal{X}_P$ then $\mathcal{H}^{i+1} \in \mathcal{X}_P$. Given $\mathcal{H}^i \in \mathcal{X}_P$,

the conditions for $\mathcal{H}^{i+1} \in \mathcal{X}_P$ are given by Definition 2.9,

for all $j \in I$,

I1 : H_j^i is a prefix of H_j^{i+1} and they differ by at most one event, and

I2 : $H_j^{i+1} \subseteq H_j^i \cup P_j(\mathcal{H}^i)$.

2.4.1 General Protocols

In this section, we define a set of cuts that necessarily belong to the set \mathcal{X}_P , where \mathcal{P} is a **general** protocol. Let the set be denoted as \mathcal{X}_{gn} . A cut \mathcal{H} belongs to the set \mathcal{X}_{gn} if the following conditions hold.

1. For all messages x in $Msg(H)$, $x.s^*$ immediately precedes $x.s$ and $x.r^*$ immediately precedes $x.r$.
2. All messages requested have been delivered, that is $x.s^* \in H \Rightarrow x.r \in H$.
3. There exists a numbering scheme Num that assigns a unique number to each event such that

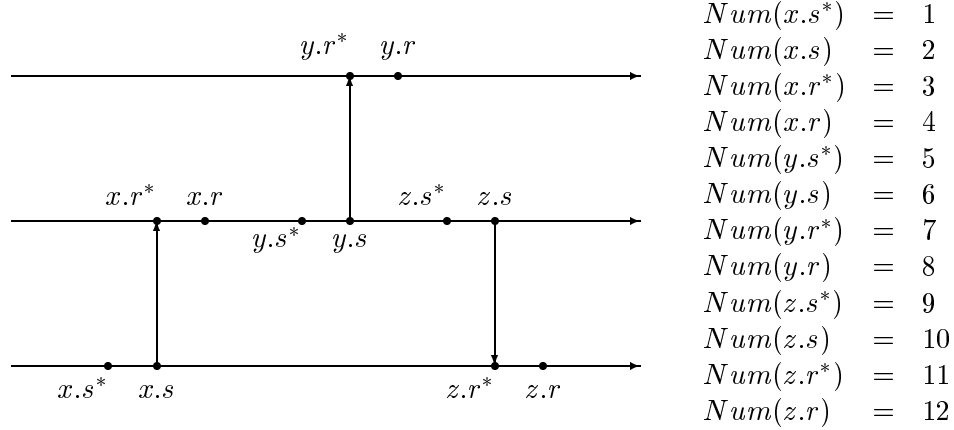


Figure 2.6: A cut belonging to \mathcal{H}_{gn} .

- (a) for any two events $h, g \in H$, if $h \rightarrow g$ then $Num(h) < Num(g)$, and
- (b) for any message $x \in Msg(H)$, $Num(x.r) = Num(x.r^*) + 1 = Num(x.s) + 2 = Num(x.s^*) + 3$.

The time diagram of any element in the set \mathcal{X}_{gn} can be drawn in such a way that all message arrows are vertical. Figure 2.6 shows an example cut in \mathcal{X}_{gn} .

Lemma 2.2 Let \mathcal{P} be a protocol satisfying the liveness property and $\mathcal{X}_{\mathcal{P}}$ the set of all cuts possible under the protocol. If \mathcal{P} is a **general** protocol, then $\mathcal{X}_{gn} \subseteq \mathcal{X}_{\mathcal{P}}$.

Proof: A **general** protocol \mathcal{P} satisfies the following properties:

Controllability : $I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \subseteq P_i(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \cup C_i(\mathcal{H})$.

Liveness : $R(\mathcal{H}) \cup C(\mathcal{H}) \neq \emptyset \Rightarrow P(\mathcal{H}) \cap (R(\mathcal{H}) \cup C(\mathcal{H})) \neq \emptyset$.

Let $\mathcal{H} \in \mathcal{X}_{gn}$. By the definition of \mathcal{X}_{gn} , there exists a numbering scheme Num that assigns a unique number to each event, such that

$$Num(x.r) = Num(x.r^*) + 1 = Num(x.s) + 2 = Num(x.s^*) + 3$$

and $(g \rightarrow h) \Rightarrow (Num(g) < Num(h))$. Using this numbering, we can define a total order in the messages and construct the required prefixes, that is, $\mathcal{H}^0, \mathcal{H}^1, \dots$, as shown in Figure 2.7. Since $\mathcal{H}^0 \in \mathcal{X}_{\mathcal{P}}$ it is sufficient to show that $\mathcal{H}^{i+1} \in \mathcal{X}_{\mathcal{P}}$, given

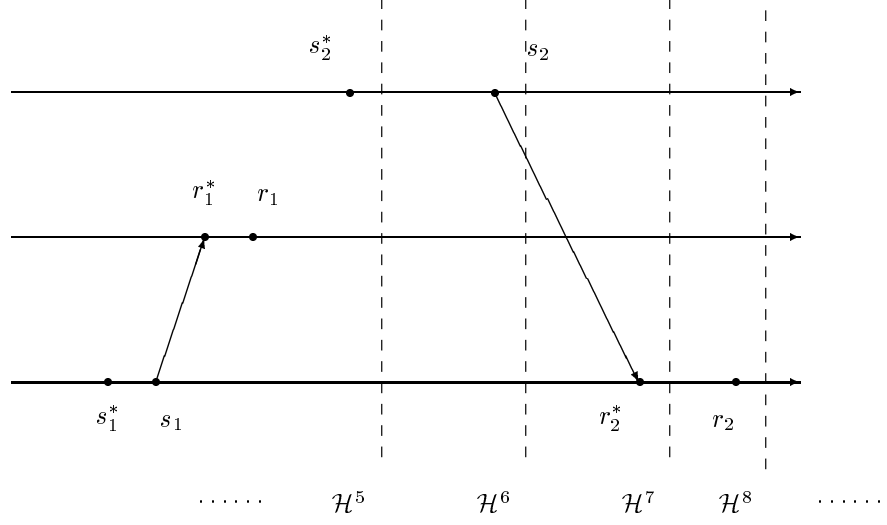


Figure 2.7: Numbering scheme for an element $\mathcal{H} \in \mathcal{X}_{gn}$.

$\mathcal{H}^i \in \mathcal{X}_P$. Clearly, H^{i+1} and H^i differ at most by one event. Therefore, for each j , H_j^i is a prefix of H_j^{i+1} and they differ by at most one event. Thus, **I1** is satisfied for all j .

We have to show that **I2** is satisfied, that is, $(H^{i+1} - H^i) \subseteq P(\mathcal{H}^i)$. There are four possible cases:

- Let $\underline{i = 4m}$. Then $H^{i+1} - H^i = \{s_{m+1}^*\}$ and $s_{m+1}^* \in I(\mathcal{H}^i)$, since only up to m messages have been executed. Due to controllability property, $s_{m+1}^* \in I(\mathcal{H}^i)$ implies $s_{m+1}^* \in P(\mathcal{H}^i)$.
- Let $\underline{i = 4m + 1}$. Then $H^{i+1} - H^i = \{s_{m+1}\}$ and $S(\mathcal{H}^i) = \{s_{m+1}\}$, $R(\mathcal{H}^i) = \emptyset$ and $D(\mathcal{H}^i) = \emptyset$. Due to liveness property, the singleton set $C(\mathcal{H}^i) \cup R(\mathcal{H}^i)$ implies $S(\mathcal{H}^i) \subseteq P(\mathcal{H}^i)$. Therefore, $s_{m+1} \in P(\mathcal{H}^i)$.
- Let $\underline{i = 4m + 2}$. Then $H^{i+1} - H^i = \{r_{m+1}^*\}$ and $S(\mathcal{H}^i) = \emptyset$, $R(\mathcal{H}^i) = \{r_{m+1}^*\}$ and $D(\mathcal{H}^i) = \emptyset$. Due to controllability property, $r_{m+1}^* \in R(\mathcal{H}^i)$ implies $r_{m+1}^* \in P(\mathcal{H}^i)$.

- Let $i = 4m + 3$. Then $H^{i+1} - H^i = \{r_{m+1}\}$ and $S(\mathcal{H}^i) = \emptyset$, $R(\mathcal{H}^i) = \emptyset$ and $D(\mathcal{H}^i) = \{r_{m+1}\}$. Due to liveness property, the singleton set $C(\mathcal{H}^i) \cup R(\mathcal{H}^i)$ implies $R(\mathcal{H}^i) \subseteq P(\mathcal{H}^i)$. Therefore, $r_{m+1} \in P(\mathcal{H}^i)$.

Therefore, in each case we have $H^{i+1} = H^i \cup P(\mathcal{H}^i)$, or, for each $j \in I$, we have $H_j^{i+1} \subseteq H_j^i \cup P_j(\mathcal{H}^i)$. \square

2.4.2 Tagged Protocols

In this section, we define a set of cuts, denoted as \mathcal{X}_{td} , that necessarily belong to the set $\mathcal{X}_{\mathcal{P}}$, where \mathcal{P} is a **tagged** protocol. A cut \mathcal{H} belongs to the set \mathcal{X}_{td} if the following conditions hold.

1. For all messages x in $Msg(H)$, $x.s^*$ immediately precedes $x.s$ and $x.r^*$ immediately precedes $x.r$.
2. All messages requested have been delivered, that is $x.s^* \in H \Rightarrow x.r \in H$.
3. All pairs of messages are causally ordered, that is, $x.s \rightarrow y.s \Rightarrow \neg(y.r^* \rightarrow x.r^*)$.

Lemma 2.3 Let \mathcal{P} be a protocol satisfying the liveness property and $\mathcal{X}_{\mathcal{P}}$ be the set of all cuts possible under the protocol. If \mathcal{P} is a **tagged** protocol, then $\mathcal{X}_{td} \subseteq \mathcal{X}_{\mathcal{P}}$.

Proof: A **tagged** protocol \mathcal{P} satisfies the following properties:

Controllability : $I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \subseteq P_i(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup R_i(\mathcal{H}) \cup C_i(\mathcal{H})$.

Liveness : $R(\mathcal{H}) \cup C(\mathcal{H}) \neq \emptyset \Rightarrow P(\mathcal{H}) \cap (R(\mathcal{H}) \cup C(\mathcal{H})) \neq \emptyset$.

Causal : $CausalPast_i(\mathcal{H}) = CausalPast_i(\mathcal{G}) \Rightarrow P_i(\mathcal{H}) = P_i(\mathcal{G})$.

To prove that a cut $\mathcal{H} \in \mathcal{X}_{\mathcal{P}}$, we have to construct a sequence of cuts $\mathcal{H}^0, \mathcal{H}^1, \dots$, that are prefixes of \mathcal{H} . We construct the sequence such that if the longest path from start of the computation to an event h is k , then $h \notin H^{k-1}$ and $h \in H^k$.

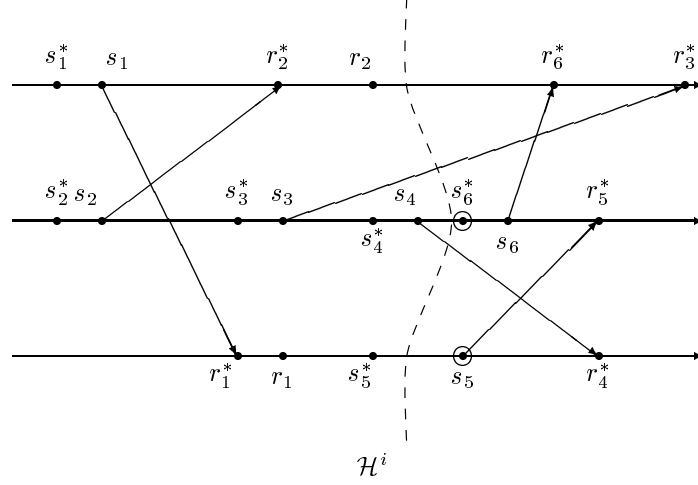


Figure 2.8: Constructing the next prefix given \mathcal{H}^i .

Let \mathcal{H}^i be a prefix of \mathcal{H} , then \mathcal{H}^{i+1} contains the bottom elements among the events that do not belong to \mathcal{H}^i . For example, in Figure 2.8, $H^{i+1} = H^i \cup \{s_5, s_6^*\}$. Formally,

$$H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i), \quad \text{for all } j$$

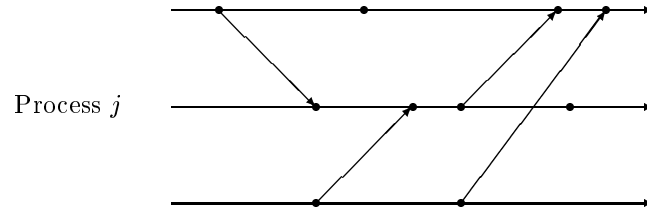
where $B_j(\mathcal{H}, \mathcal{H}^i) = \left\{ h \in H_j - H_j^i : (g \rightarrow h) \Rightarrow g \in H^i \right\}$ and satisfies the following properties:

1. $B_j(\mathcal{H}, \mathcal{H}^i)$ is a singleton or an empty set.
2. $B_j(\mathcal{H}, \mathcal{H}^i) \subseteq I_j(\mathcal{H}^i) \cup R_j(\mathcal{H}^i) \cup C_j(\mathcal{H}^i)$.

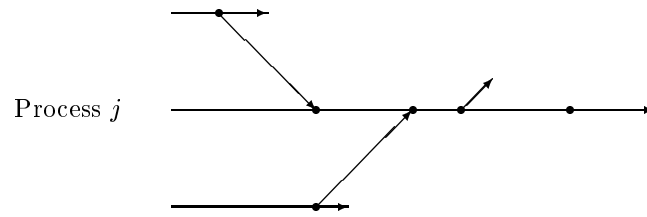
We have to show that if $\mathcal{H} \in \mathcal{X}_{id}$, then $\mathcal{H} \in \mathcal{X}_P$. Clearly, $\mathcal{H}^0 \in \mathcal{X}_P$, since it is the empty cut. Let $\mathcal{H}^i \in \mathcal{X}_P$, we have to show that $\mathcal{H}^{i+1} \in \mathcal{X}_P$. The prefixes satisfy **I1**, since $B_j(\mathcal{H}, \mathcal{H}^i)$ is a singleton or an empty set and $H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i)$ for all j . Further, we have to show that $\mathcal{H}_j^{i+1} \subseteq \mathcal{H}_j^i \cup P_j(\mathcal{H}^i)$ for all j (**I2**).

Given \mathcal{H}^i and j , construct a cut \mathcal{G} as shown in Figure 2.9. Pick $CausalPast_j(\mathcal{H}^i)$ and extend all messages (with destination process not being j) in transit². Therefore, $CausalPast_j(\mathcal{H}^i) = CausalPast_j(\mathcal{G})$. We make the following claims:

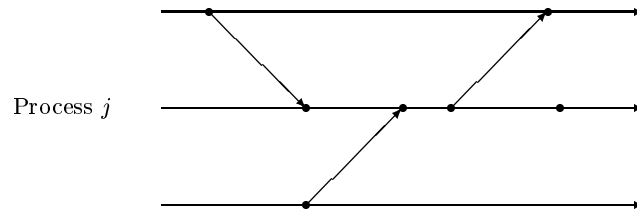
²pick any possible extension



(a) Cut \mathcal{H}^i



(b) $\text{CausalPast}_2(\mathcal{H}^i)$



(c) Cut \mathcal{G}

Figure 2.9: Construction of \mathcal{G} given \mathcal{H}^i for process j .

1. $P_j(\mathcal{G}) = P_j(\mathcal{H}^i)$.

Since $CausalPast_j(\mathcal{H}^i) = CausalPast_j(\mathcal{G})$ by construction, therefore using the causal property, we get $P_j(\mathcal{H}^i) = P_j(\mathcal{G})$.

2. $C_j(\mathcal{G}) = C_j(\mathcal{H}^i)$.

For any cut \mathcal{H} we have

$$\begin{aligned} S_j(\mathcal{H}) &= \{x.s : (x.s^* \in H_j) \wedge ;(x.s \notin H_j)\}, \text{ and} \\ D_j(\mathcal{H}) &= \{x.r : (x.r^* \in H_j) \wedge (x.r \notin H_j)\}. \end{aligned}$$

By construction of \mathcal{G} we have $H_j^i = G_j$. Therefore $S_j(\mathcal{G}) = S_j(\mathcal{H}^i)$ and $D_j(\mathcal{G}) = D_j(\mathcal{H}^i)$. Since for any cut $C_j(\mathcal{H}) = S_j(\mathcal{H}) \cup D_j(\mathcal{H})$, we have $C_j(\mathcal{G}) = C_j(\mathcal{H}^i)$.

3. $R_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

For any cut \mathcal{H} , $R_k(\mathcal{H})$ represents the messages in transit destined for process k . By construction of \mathcal{G} , we have $R_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

4. $R_j(\mathcal{G}) = \emptyset$. (Proof by contradiction)

Let $x.r^* \in R_j(\mathcal{G})$, therefore $\exists k : (x.r^* \notin G_j) \wedge (x.s \in G_k) \wedge (x \in M_{kj})$. Since $(x.s \in G_k)$ and $(x \in M_{kj})$, we have by the definition of *CausalPast* $\exists h : (x.s \rightarrow h) \wedge (h \in G_j)$.

Since $(x.s \rightarrow h)$ and $x.s, h$ are in different processes, we have either

- (a) $\exists y \in M : (x.s \rightarrow y.s) \wedge ((y.r^* \rightarrow h) \vee (y, r^* \equiv h))$, or
- (b) $(x.s \rightarrow x.r^*) \wedge (x.r^* \rightarrow h)$.

But $x.r^* \notin G$, therefore $\neg(x.r^* \rightarrow h)$. Since $h, x.r^* \in H_j$, either $(x.r^* \rightarrow h)$ or $(h \rightarrow x.r^*)$. Therefore,

$$\exists x, y \in M : (x.s \rightarrow y.s) \wedge (y.r^* \rightarrow x.r^*).$$

This implies $\mathcal{H} \notin \mathcal{X}_{id}$.

5. $C_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

Let $x.s^* \in G_k$. Since $x.s^*$ immediately precedes $x.s$, therefore if there exists an event h such that $x.s^* \rightarrow h$, then $x.s \in G_k$. From the construction of \mathcal{G} , we have $x.s^* \in G_k$, then $x.s^*$ is also an event in the cut $CausalPast_j(\mathcal{H}^i)$. From the definition of $CausalPast_j(\cdot)$, if $x.s^* \in CausalPast_j(\mathcal{H}^i)$, then either $x.s^* \in H_j^i$ or there exists an event h such that $h \in H_j^i$ and $x.s^* \rightarrow h$. Therefore, $x.s \in G_k \Rightarrow x.s^* \in G_k$.

Since $x.s^* \in G_k \Leftrightarrow x.s \in G_k$, we have $S_k(\mathcal{G}) = \emptyset$ for all $k \neq j$ and by construction of \mathcal{G} , $D_k(\mathcal{G}) = \emptyset$ for all $k \neq j$.

6. $C_j(\mathcal{G})$ is a singleton or an empty set.

Since $x.s^*$ immediately precedes $x.s$ and $x.r^*$ immediately precedes $x.r$.

From (3), (4), (5) and (6), we have

$$R(\mathcal{G}) \cup C(\mathcal{G}) = \left(\bigcup_k R_k(\mathcal{G}) \right) \cup \left(\bigcup_{k \neq j} C_k(\mathcal{G}) \right) \cup C_j(\mathcal{G})$$

is a singleton or empty set. Using liveness property and $R(\mathcal{G}) \cup C(\mathcal{G})$ being either a singleton or empty set, we get

$$C_j(\mathcal{G}) \subseteq P_j(\mathcal{G}).$$

Substituting for $C_j(\mathcal{G})$ and $P_j(\mathcal{G})$ from (1) and (2), we get

$$C_j(\mathcal{H}^i) \subseteq P_j(\mathcal{H}^i).$$

From the controllability property and $C_j(\mathcal{H}^i) \subseteq P_j(\mathcal{H}^i)$, we get

$$P_j(\mathcal{H}^i) = I_j(\mathcal{H}^i) \cup R_j(\mathcal{H}^i) \cup C_j(\mathcal{H}^i).$$

Therefore,

$$B_j(\mathcal{H}, \mathcal{H}^i) \subseteq P_j(\mathcal{H}^i),$$

or

$$H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i) \subseteq H_j^i \cup P_j(\mathcal{H}^i).$$

Thus, **I2** is satisfied. □

2.4.3 Tagless Protocols

In this section, we define a set of cuts that necessarily belong to the set \mathcal{X}_P , where \mathcal{P} is a **tagless** protocol. Let the set be denoted as \mathcal{X}_{tl} . A cut \mathcal{H} belongs to the set \mathcal{X}_{tl} if the following conditions hold.

1. For all messages x in $Msg(H)$, $x.s^*$ immediately precedes $x.s$ and $x.r^*$ immediately precedes $x.r$.
2. All messages requested have been delivered, that is $x.s^* \in H \Rightarrow x.r \in H$.

Lemma 2.4 Let \mathcal{P} be a protocol satisfying the liveness property and \mathcal{X}_P is the set of all cuts possible under the protocol. If \mathcal{P} is a **tagless** protocol, then $\mathcal{X}_{tl} \subseteq \mathcal{X}_P$.

Proof: A **tagless** protocol \mathcal{P} satisfies the following properties:

Controllability : $I_i(\mathcal{H}) \cup D_i(\mathcal{H}) \subseteq P_i(\mathcal{H}) \subseteq I_i(\mathcal{H}) \cup D_i(\mathcal{H}) \cup C_i(\mathcal{H})$.

Liveness : $R(\mathcal{H}) \cup C(\mathcal{H}) \neq \emptyset \Rightarrow P(\mathcal{H}) \cap (R(\mathcal{H}) \cup C(\mathcal{H})) \neq \emptyset$.

Local : $H_i = G_i \Rightarrow P_i(\mathcal{H}) = P_i(\mathcal{G})$.

To prove that a cut $\mathcal{H} \in \mathcal{X}_P$, we have to construct a sequence of cuts $\mathcal{H}^0, \mathcal{H}^1, \dots$, that are prefixes of \mathcal{H} .

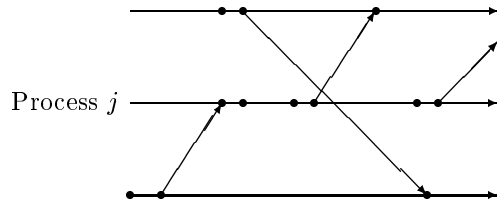
We construct the prefixes as in Lemma 2.3. The construction satisfies

$$H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i), \quad \text{for all } j$$

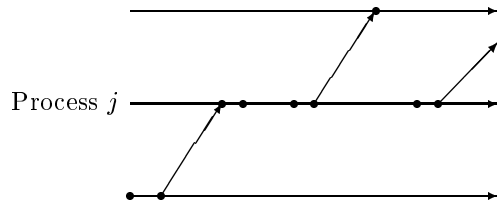
where $B_j(\mathcal{H}, \mathcal{H}^i) = \left\{ h \in H_j - H_j^i : (g \rightarrow h) \Rightarrow g \in H^i \right\}$, and satisfies the following properties:

1. $B_j(\mathcal{H}, \mathcal{H}^i)$ is a singleton or an empty set.
2. $B_j(\mathcal{H}, \mathcal{H}^i) \subseteq I_j(\mathcal{H}^i) \cup R_j(\mathcal{H}^i) \cup C_j(\mathcal{H}^i)$.

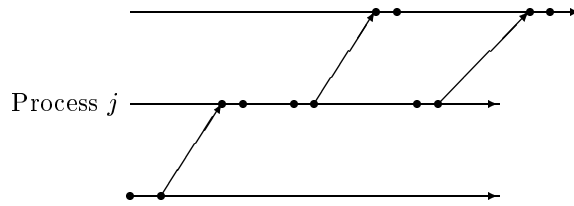
We have to show that if $\mathcal{H} \in \mathcal{X}_{tl}$, then $\mathcal{H} \in \mathcal{X}_P$. Clearly, $\mathcal{H}^0 \in \mathcal{X}_P$, since it is the empty cut. Let $\mathcal{H}^i \in \mathcal{X}_P$, we have to show that $\mathcal{H}^{i+1} \in \mathcal{X}_P$. The prefixes satisfy



(a) Cut \mathcal{H}^i



(a) Cut \mathcal{H}^i with events removed.



(a) Cut \mathcal{G}

Figure 2.10: Construction of \mathcal{G} given \mathcal{H}^i for process j .

I1, since $B_j(\mathcal{H}, \mathcal{H}^i)$ is a singleton or an empty set and $H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i)$ for all j . Further, we have to show that $\mathcal{H}_j^{i+1} \subseteq \mathcal{H}_j^i \cup P_j(\mathcal{H}^i)$ for all j (**I2**).

Given \mathcal{H}^i and some j , construct a cut \mathcal{G} as shown in Figure 2.10. Remove and add events such that $G_j = H_j^i$ using the following two steps.

1. Delete all messages from the cut \mathcal{H}^i preserving the condition $G_j = H_j^i$.
2. The messages sent to k ($k \neq j$) are delivered and received.

Therefore, for all $k \neq j$ $R_k(\mathcal{G}) = \emptyset$ and $D_k(\mathcal{G}) = \emptyset$, and $G_j = H_j^i$. We make the following claims:

1. $P_j(\mathcal{H}^i) = P_j(\mathcal{G})$.

Since $G_j = H_j^i$ by construction therefore, using the property **P3**, we get $P_j(\mathcal{H}^i) = P_j(\mathcal{G})$.

2. $C_j(\mathcal{G}) = C_j(\mathcal{H}^i)$.

For any cut \mathcal{H} we have

$$\begin{aligned} S_j(\mathcal{H}) &= \{x.s : (x.s^* \in H_j) \wedge ;(x.s \notin H_j)\}, \text{ and} \\ D_j(\mathcal{H}) &= \{x.r : (x.r^* \in H_j) \wedge (x.r \notin H_j)\}. \end{aligned}$$

By construction of \mathcal{G} we have $H_j^i = G_j$. Therefore $S_j(\mathcal{G}) = S_j(\mathcal{H}^i)$ and $D_j(\mathcal{G}) = D_j(\mathcal{H}^i)$. Since for any cut $C_j(\mathcal{H}) = S_j(\mathcal{H}) \cup D_j(\mathcal{H})$, we have $C_j(\mathcal{G}) = C_j(\mathcal{H}^i)$.

3. $R_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

For any cut \mathcal{H} , $R_k(\mathcal{H})$ represents the messages in transit destined for process k . By construction of \mathcal{G} , we have $R_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

4. $R_j(\mathcal{G}) = \emptyset$.

There are no messages with destination process being j , since by step 1 of the construction the corresponding invocation and/or send event is removed.

5. $C_k(\mathcal{G}) = \emptyset$, where $k \neq j$.

By construction of \mathcal{G} , $D_k(\mathcal{G}) = \emptyset$ and $S_k(\mathcal{G}) = \emptyset$ for all $k \neq j$.

6. $C_j(\mathcal{G})$ is a singleton or an empty set.

Since $x.s^*$ immediately precedes $x.s$ and $x.r^*$ immediately precedes $x.r$.

Following the steps in the proof of Lemma 2.3, we get

$$B_j(\mathcal{H}, \mathcal{H}^i) \subseteq P_j(\mathcal{H}^i),$$

or

$$H_j^{i+1} = H_j^i \cup B_j(\mathcal{H}, \mathcal{H}^i) \subseteq H_j^i \cup P_j(\mathcal{H}^i).$$

Thus, **I2** is satisfied. □

2.5 Limit Sets

In this section, we consider the problem of finding the type of protocol necessary and sufficient to implement a given specification.

In Section 2.4, we investigated the question of whether a cut necessarily belongs to \mathcal{X}_P , given a protocol \mathcal{P} . In this section, we pose the same question but in a different setting; that is, given a run (H, \triangleright) does it necessarily belong to \mathbb{X}_P ? Given a specification \mathbb{Y} , this gives us lower bounds on the specification \mathbb{Y} that is *necessary* for the existence of a **general**, a **tagged** or a **tagless** protocol. For example, if a **general** protocol implements the specification \mathbb{Y} then $\mathbb{X}_n \subseteq \mathbb{Y}$, where \mathbb{X}_n is the lower bound for the class of **general** protocols. In this section, we present results in the other direction, that is, does there exist a limit \mathbb{X}_s that is *sufficient* for the existence of a **general** protocol?

We define three subsets of \mathbb{X} (or specifications) similar to ones in Section 2.4 that are used to provide an answer to the problem stated in this section. The three subsets of \mathbb{X} are:

Asynchronous ordering (ASYNC)

This is the same as the ground set \mathbb{X} . Therefore, it includes all possible runs. There exists a **tagless** algorithm (i.e., enable all pending events) that guarantees safety

and liveness for this specification. Formally, we can state \mathbb{X}_{async} , the set of all partial orders as

$$\mathbb{X}_{async} = \{ (\mathbb{H}, \triangleright) : (x.s \in \mathbb{H} \Leftrightarrow x.r \in \mathbb{H}) \text{ and } \triangleright \text{ is a partial order} \}.$$

Causal Ordering (CO)

Causal ordering can be stated as $s_1 \triangleright s_2 \Rightarrow \neg(r_2 \triangleright r_1)$. There exists a **tagged** algorithm where with each message a matrix of size $n \times n$ is tagged to the message [32, 33]. Formally, we can state \mathbb{X}_{co} , the set of partial orders satisfying causal ordering as

$$\mathbb{X}_{co} = \{ (\mathbb{H}, \triangleright) : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)) \forall x, y \in M \}.$$

Logically Synchronous (SYNC)

A run is logically synchronous if its time diagram can be drawn such that all message arrows are vertical. Formally, a run $(\mathbb{H}, \triangleright)$ is logically synchronous, that is $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$, if there exists a function $T : Msg(\mathbb{H}) \rightarrow \{1, 2, 3, \dots\}$, such that for any two events $h, g \in \mathbb{H}$, if $h \triangleright g$ and $Msg(h) \neq Msg(g)$ then $T(Msg(h)) < T(Msg(g))$.

Definition 2.13 (Crown [14]) A crown (of size k) in a run is a sequence of messages $\langle x_1, x_2, \dots, x_k \rangle$ such that

$$(x_1.s \triangleright x_2.r) \wedge (x_2.s \triangleright x_3.r) \cdots (x_k.s \triangleright x_1.r).$$

Theorem 2.1 If a run is logically synchronous, that is $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$, then there does not exist a sequence of messages $\langle x_1, x_2, \dots, x_k \rangle$ belonging to the run, such that

$$(x_1.s \triangleright x_2.r) \wedge (x_2.s \triangleright x_3.r) \cdots (x_k.s \triangleright x_1.r).$$

Proof:

Synchronous $\Rightarrow \neg$ **Crown**

Since the run is synchronous there exists a function T_e such that for any two events

h and g

$$\begin{aligned} (h \triangleright g) \wedge \text{Msg}(h) \neq \text{Msg}(g) &\Rightarrow \text{T}_e(h) < \text{T}_e(g), \text{ and} \\ (h \triangleright g) \wedge \text{Msg}(h) = \text{Msg}(g) &\Rightarrow \text{T}_e(h) = \text{T}_e(g). \end{aligned}$$

Suppose the computation has a crown of size k ,

$$(x_{1.s} \triangleright x_{2.r}) \wedge (x_{2.s} \triangleright x_{3.r}) \cdots (x_{k.s} \triangleright x_{1.r}).$$

Therefore,

$$\begin{aligned} \forall i \in \{1, \dots, k\} \quad \text{T}_e(x_{i.s}) &< \text{T}_e(x_{(i+1) \bmod k.r}) & (*) \\ \forall i \in \{2, \dots, k\} \quad \text{T}_e(x_{i.s}) &= \text{T}_e(x_{i.r}). & (**) \end{aligned}$$

Therefore, from equations (*) and (**),

$$\text{T}_e(x_{1.s}) < \text{T}_e(x_{1.r}).$$

which is a contradiction.

\neg Crown \Rightarrow Synchronous

Given a run $(\mathbb{H}, \triangleright)$, we form a directed graph $G(V, E)$, as follows. The vertex set V consists of all messages $\{x_1, x_2, \dots\}$ in the computation. Thus, each vertex v_i represents a set of two events: the send event $x_{i.s}$ and the corresponding receive event $x_{i.r}$. That is

$$v_i = \{x_{i.s}, x_{i.r}\}.$$

There is an edge from v_i to v_j if there is an event $h \in v_i$ and an event $h \in v_j$ such that $h \triangleright g$. Thus, $(v_i, v_j) \in E$ iff $(x_{i.s} \triangleright x_{j.s}) \vee (x_{i.s} \triangleright x_{j.r}) \vee (x_{i.r} \triangleright x_{j.s}) \vee (x_{i.r} \triangleright x_{j.r})$. It is easy to see that each of the four disjuncts implies $x_{i.s} \triangleright x_{j.r}$. Hence, $(v_i, v_j) \in E$ iff $x_{i.s} \triangleright x_{j.r}$. Since the computation does not have any crown, it follows that the graph G is acyclic and the graph can be topologically sorted. Therefore, if we pick the same ordering, we get the desired function $T : \text{Msg}(\mathbb{H}) \rightarrow \{1, 2, 3, \dots\}$, such that for any two events $h, g \in \mathbb{H}$, if $h \triangleright g$ and $\text{Msg}(h) \neq \text{Msg}(g)$ then $T(\text{Msg}(h)) < T(\text{Msg}(g))$. Therefore the run $(\mathbb{H}, \triangleright)$ is synchronous. \square

It is easy to see that [14, 36]

$$\mathbb{X}_{sync} \subseteq \mathbb{X}_{co} \subseteq \mathbb{X}_{async}.$$

The sets \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} exhibit an important property, i.e., they are the limiting specifications, in terms of whether there exists a protocol that can guarantee safety and liveness, for each of the three classes of protocols. For example, there exists a **tagged** protocol (i.e., no control messages) that guarantees safety and liveness for the specification \mathbb{X}_{co} . Further, given a specification \mathbb{Y} , there exists a **tagged** protocol that guarantees safety and liveness, if and only if $\mathbb{X}_{co} \subseteq \mathbb{Y}$. Thus, given a specification, i.e., the set of acceptable runs, the type of protocol necessary and sufficient can be easily checked by testing the containment of the three limit sets, \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} .

Theorem 2.2 Let \mathbb{Y} be a specification. Then

1. A **general** protocol can guarantee safety and liveness iff $\mathbb{X}_{sync} \subseteq \mathbb{Y}$.
2. A **tagged** protocol can guarantee safety and liveness iff $\mathbb{X}_{co} \subseteq \mathbb{Y}$.
3. A **tagless** protocol can guarantee safety and liveness iff $\mathbb{X}_{async} \subseteq \mathbb{Y}$.

Proof: It is easy to show the “if part” in each of the cases. We use the fact that if a protocol \mathcal{P} implements the specification \mathbb{Y} , then $\mathbb{X}_{\mathcal{P}} \subseteq \mathbb{Y}$.

1. There exists a **general** protocol \mathcal{P} such that $\mathbb{X}_{\mathcal{P}} = \mathbb{X}_{sync}$ [4, 29].
2. There exists a **tagged** protocol \mathcal{P} such that $\mathbb{X}_{\mathcal{P}} = \mathbb{X}_{co}$ [32, 33].
3. There exists a **tagless** protocol \mathcal{P} such that $\mathbb{X}_{\mathcal{P}} = \mathbb{X}_{async}$ (enable all events).

We now proceed to show the “only if part”.

Part 1. Let \mathcal{P} be a **general** protocol. From lemma 2.2, we have $\mathcal{X}_{gn} \subseteq \mathcal{X}_{\mathcal{P}}$. We have to show that if $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$ then $\exists \mathcal{H} \in \mathcal{X}_{gn}$ such that $(\mathbb{H}, \triangleright) = UserView(\mathcal{H})$.

Given $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$ we construct \mathcal{H} such that $(\mathbb{H}, \triangleright) = UserView(\mathcal{H})$ and $\mathcal{H} \in \mathcal{X}_{sync}$, as shown in Figure 2.11. For each event $x.s$ add $x.s^*$ such that $x.s^*$



Figure 2.11: Construction of \mathcal{H} from $(\mathbb{H}, \triangleright)$.

immediately precedes $x.s$. Similarly, for each event $x.r$ add $x.r^*$ such that $x.r^*$ immediately precedes $x.r$. We claim that $\mathcal{H} \in \mathcal{X}_{gn}$, that is, \mathcal{H} satisfies the conditions satisfied by elements of \mathcal{X}_{gn} .

1. $x.s^*$ immediately precedes $x.s$, and $x.r^*$ immediately precedes $x.r$, by construction of \mathcal{H} .
2. All messages invoked have been delivered, that is, $x.s^* \in H \Rightarrow x.r \in H$, since $(\mathbb{H}, \triangleright)$ is a run.
3. Since $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$, there exists a function $T : Msg(\mathbb{H}) \rightarrow \{1, 2, 3, \dots\}$, such that for any two events $h, g \in \mathbb{H}$, if $h \triangleright g$ and $Msg(h) \neq Msg(g)$ then $T(Msg(h)) < T(Msg(g))$. We can derive the numbering scheme Num , where for each message x , $Num(x.s^*) = 4 * T(x)$, $Num(x.s) = 4 * T(x) + 1$, $Num(x.r^*) = 4 * T(x) + 2$ and $Num(x.r) = 4 * T(x) + 3$. Thus, we have a numbering scheme that assigns a unique number to each event satisfying desired properties, that is, for any two events h, g if $h \rightarrow g$ then $Num(h) < Num(g)$ and for any message x , $Num(x.r) = Num(x.r^*) + 1 = Num(x.s) + 2 = Num(x.s^*) + 3$.

Thus, if \mathcal{P} is a **general** protocol and $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$ then $(\mathbb{H}, \triangleright) = \mathbb{X}_{\mathcal{P}}$.

Part 2. Let \mathcal{P} be a **tagged** protocol. From lemma 2.3, we have $\mathcal{X}_{td} \subseteq \mathcal{X}_{\mathcal{P}}$. We have to show that if $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$ then $\exists \mathcal{H} \in \mathcal{X}_{td}$ such that $(\mathbb{H}, \triangleright) = UserView(\mathcal{H})$. We claim that $\mathcal{H} \in \mathcal{X}_{td}$, that is, \mathcal{H} satisfies the conditions satisfied by elements of \mathcal{X}_{td} .

The proof is similar to the previous case. We construct \mathcal{H} as above and show that $\mathcal{H} \in \mathcal{X}_{td}$.

1. $x.s^*$ immediately precedes $x.s$, and $x.r^*$ immediately precedes $x.r$, by construction of \mathcal{H} .
2. All messages invoked have been delivered, that is, $x.s^* \in H \Rightarrow x.r \in H$, since $(\mathbb{H}, \triangleright)$ is a run.
3. Since $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$ we have, for any two messages x, y the relation $(x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$ is false. By construction, we have $(x.s \triangleright y.s) \Leftrightarrow (x.s \rightarrow y.s)$ and $(x.r \triangleright y.r) \Leftrightarrow (x.r^* \rightarrow y.r^*)$. Therefore, in the cut \mathcal{H} for any two messages x, y the relation $(x.s \rightarrow y.s) \wedge (y.r^* \rightarrow x.r^*)$ is false. In other words, for any two messages x, y we have $(x.s \rightarrow y.s) \Rightarrow \neg(y.r^* \rightarrow x.r^*)$.

Thus, if \mathcal{P} is a **tagged** protocol and $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$ then $(\mathbb{H}, \triangleright) = \mathbb{X}_{\mathcal{P}}$.

Part 3. Let \mathcal{P} be a **tagless** protocol. From lemma 2.4, we have $\mathcal{X}_{tl} \subseteq \mathcal{X}_{\mathcal{P}}$. We have to show that if $(\mathbb{H}, \triangleright) \in \mathbb{X}_{async}$ then $\exists \mathcal{H} \in \mathcal{X}_{tl}$ such that $(\mathbb{H}, \triangleright) = UserView(\mathcal{H})$. We claim that $\mathcal{H} \in \mathcal{X}_{tl}$, that is, \mathcal{H} satisfies the conditions satisfied by elements of \mathcal{X}_{tl} . The proof is similar to the previous case. We construct \mathcal{H} as above and show that $\mathcal{H} \in \mathcal{X}_{tl}$.

1. $x.s^*$ immediately precedes $x.s$, and $x.r^*$ immediately precedes $x.r$, by construction of \mathcal{H} .
2. All messages invoked have been delivered, that is, $x.s^* \in H \Rightarrow x.r \in H$, since $(\mathbb{H}, \triangleright)$ is a run.

Thus, if \mathcal{P} is a **tagless** protocol and $(\mathbb{H}, \triangleright) \in \mathbb{X}_{async}$ then $(\mathbb{H}, \triangleright) = \mathbb{X}_{\mathcal{P}}$. □

Corollary 2.1 A specification \mathbb{Y} is implementable, that is, there exists a **tagless**, **tagged**, or **general** protocol, if and only if $\mathbb{X}_{sync} \subseteq \mathbb{Y}$.

2.6 Related Work

In [8], Bougé and Francez studied inhibition (called freezing) based protocols as a superimposition of a set of control processes P on another set of user processes Q . A user process can send/receive a message only if (and when) the corresponding control process sends/receives a “similar” message. They considered syntactic representation of inhibition in CSP. The first fundamental work on the properties of inhibition was done by Taylor and Critchlow [18, 37]. They studied the relationships between inhibition and the existence of specific protocols, and distinguished local versus global inhibition.

The major difference between our model of inhibition protocols and the model used by Taylor and Critchlow is that protocol events in our model are not used to define enabling relations. They modeled enabling relations as a function of the local state of the process, where the local state is composed of a sequence of system and protocol events. We, on the other hand, define enabling relations with respect to the partial order formed by the system events in the global system thus eliminating the need to model protocol events. They studied the necessity of inhibition in consistent-cut protocols and the extent of inhibition – local versus global inhibition, inhibition of send events versus receive events, and the number of protocol messages.

Our interest is in the existence of protocol with global inhibition to implement message orderings. We classified global inhibition based on the amount of knowledge required, that is, local, causal, or concurrent, to describe the enabling relation.

In [14], Charron-Bost, Mattern, and Tel study the structural aspects of three synchronization schemes – FIFO, causal, and synchronous orderings, and the hierarchy relation, that is, synchronous \Rightarrow causal \Rightarrow FIFO.

2.7 Summary

In this chapter, we presented a new characterization of *inhibition* based protocols and *message ordering* specifications.

An *inhibition* based protocol for a distributed system specifies for each process the events it can perform. A protocol can delay the normal execution of an event until the occurrence of prerequisite events. We distinguish three kinds of inhibition based protocols:

- protocols that require control messages and tagging of user messages, called **general** protocols.
- protocols that do not require control messages, but require tagging of user messages, called **tagged** protocols, and
- protocols that do not require control messages or tagging of user messages, called **tagless** protocols.

A *message ordering specification* can be characterized as a set of acceptable runs, that is, a subset of \mathbb{X} , where \mathbb{X} is the set of all runs. The three specifications that play a key role in determining the existence of each type of protocol are:

$$\begin{aligned} \mathbb{X}_{sync} &= \{ (\mathbf{H}, \triangleright) : \neg((x_1.s \triangleright x_2.r) \wedge (x_2.s \triangleright x_3.r) \cdots (x_k.s \triangleright x_1.r)) \}, \\ &\quad \text{for any subset } \{x_1, x_2, \dots, x_k\} \text{ of } \text{Msg}(\mathbf{H}) \} \\ \mathbb{X}_{co} &= \{ (\mathbf{H}, \triangleright) : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)) \text{ for any subset } \{x, y\} \text{ of } \text{Msg}(\mathbf{H}) \} \\ \mathbb{X}_{async} &= \{ (\mathbf{H}, \triangleright) : \neg((x.s \triangleright x.s) \wedge (x.r \triangleright x.r)) \text{ for any message } x \in \text{Msg}(\mathbf{H}) \} \end{aligned}$$

Given a specification \mathbb{Y} there exists

- a **general** protocol if and only if $\mathbb{X}_{sync} \subseteq \mathbb{Y}$,
- a **tagged** protocol if and only if $\mathbb{X}_{co} \subseteq \mathbb{Y}$, and
- a **tagless** protocol if and only if $\mathbb{X}_{async} \subseteq \mathbb{Y}$.

Thus, given a specification the type of protocol necessary and sufficient can be easily checked by the containment of the three sets \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} .

It is natural to ask about the fourth type of protocols, that is, those that can use control messages but cannot tag the user messages. There exists a protocol of this type that can implement a specification if there exists a **general** protocol. A protocol of this type can send a control message before a user message including all the information that is supposed to be tagged along with the user message, thus simulating tagging.

Chapter 3

Protocol for Message-Orderings

In this chapter, we discuss the protocols implementing the three specifications: Asynchronous Ordering \mathbb{X}_{async} , Causal Ordering \mathbb{X}_{co} , and Synchronous Ordering \mathbb{X}_{sync} .

Asynchronous Ordering can be provided by a protocol that does not delay any events. Since any run (H, \triangleright) is a partial order, thus any run is a valid run under the Asynchronous Ordering specification.

A fair amount of research has been done for efficient algorithms to implement Causal Ordering. Birman and Joseph [5], Raynal, Schiper, and Toueg [32], Schiper, Eggli, and Sandoz [33], have presented algorithms for the causal ordering of messages. These algorithms tag knowledge of processes about messages sent in the system with the message. For example, process p_i in the algorithm by Raynal, Schiper, and Toueg [32] tags a message with the matrix m where $m[j, k]$ is the knowledge of process p_i about the messages sent from p_j to p_k . On receiving a message a process p_k delivers the message only if all the messages represented by $m[j, k]$ for all j have been received and delivered.

Variants of Synchronous Ordering have been studied as guarded statements in CSP [10, 4], binary interaction problem [3], and logically instantaneous message passing [36]. In the rest of the chapter, we will present an efficient protocol to

implement Synchronous Ordering.

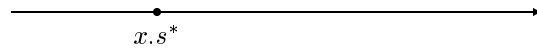
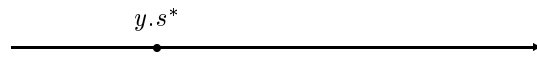
3.1 Algorithm

An algorithm that implements synchronous ordering must be asymmetric with respect to the processes. This can be easily demonstrated by a simple example. Consider a case where each of the two processes p_1, p_2 want to send one message x and y to the other, respectively. Figure 3.1(a) shows the scenario. If the protocols are symmetric with respect to the processes and the messages, then the only possible completion of the run is shown in Figure 3.1(b) but it violates the synchronous ordering conditions. Figure 3.1(c) shows the only two possible completions that do not violate synchronous ordering and these can only be achieved by imposing some order among the processes or the messages. This asymmetry can be imposed either by ordering the processes [29, 5, 36] or by imposing an order among the messages [4].

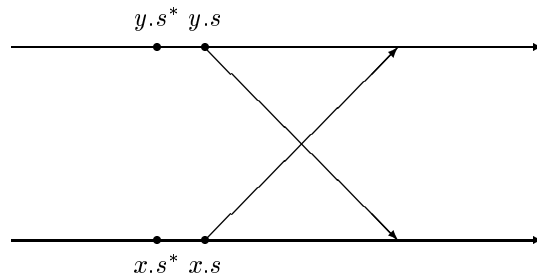
In this chapter, we present an algorithm that breaks the symmetry using the natural order among the processes. Therefore, the protocol followed by a process to send a message to a bigger process is different from the protocol followed to send a message to a smaller process. When a process wants to send a message x to a smaller process, it sends the message. On the other hand, if the process wants to send a message x to a bigger process then it has to request permission from the bigger processes before sending the message.

Figure 3.2 illustrates the messages required to send a message x from a bigger to a smaller process. The protocol is summarized in Table 3.1. Figure 3.3 illustrates the messages required to send a message x from a smaller to a bigger process. The protocol is summarized in Table 3.2.

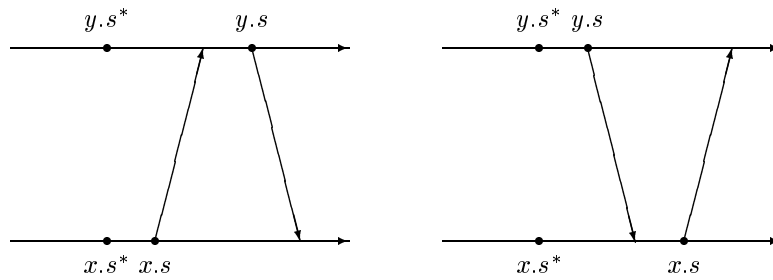
The *request* message is used to inform the bigger process of the desire for synchronization. Neither the sending nor the receiving of a request message changes the state of the process (from *active* to *passive*, or vice-versa) and the request message does not take part in the synchronization. Thus, we can neglect the *request*



(a) Request of messages by process.



(b) Completion of run under a symmetric protocol.



(c) Only possible completions valid under synchronous ordering.

Figure 3.1: Asymmetric property of an algorithm implementing SYNC.

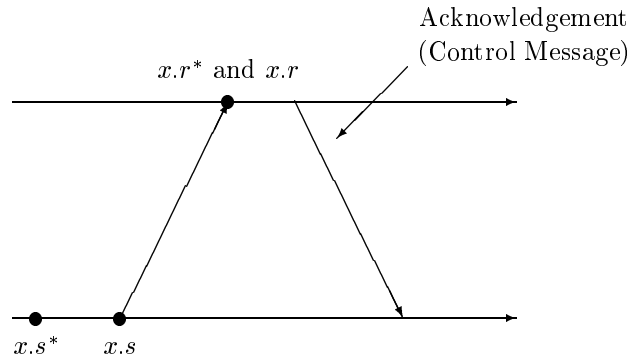


Figure 3.2: A bigger process sending a message to a smaller process.

Bigger Process	Smaller Process
wait until active <ul style="list-style-type: none"> • Send the message x, and • turn <i>passive</i> 	
	<ul style="list-style-type: none"> • Receive the message x.
	wait until active <ul style="list-style-type: none"> • Send an <i>ack</i> message for x.
<ul style="list-style-type: none"> • Receive the <i>ack</i> message, and • turn <i>active</i>. 	

Table 3.1: Protocol to send a message from a bigger to a smaller process.

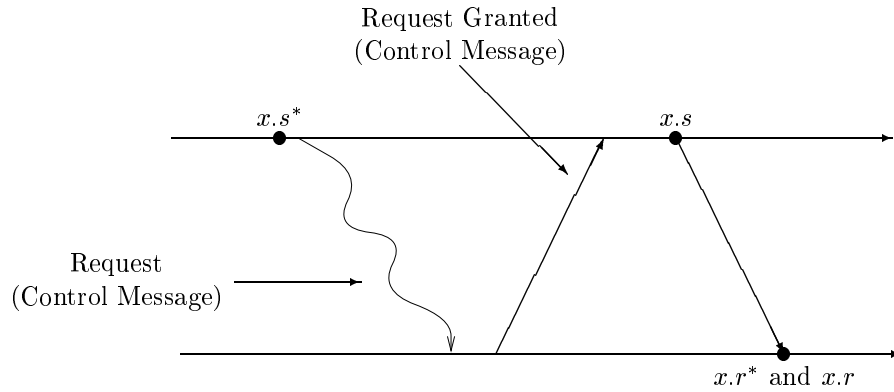


Figure 3.3: A smaller process sending a message to a bigger process.

Smaller Process	Bigger Process
<ul style="list-style-type: none"> • Send a <i>request</i> message 	
	wait until active <ul style="list-style-type: none"> • Send <i>grant</i> message for x, and • turn <i>passive</i>
<ul style="list-style-type: none"> • Receive the <i>grant</i> message. 	
wait until active <ul style="list-style-type: none"> • Send the message x. 	
	<ul style="list-style-type: none"> • Receive the message x, and • turn <i>active</i>.

Table 3.2: Protocol to send a message from a smaller to a bigger process.

messages in our arguments. Both parts of the protocol involve two messages, the first sent from the bigger to the smaller process, and second from the smaller to the bigger process.

Therefore, synchronization is achieved using two messages: initiation and acknowledgment messages. The initiation message is always from a bigger process to a smaller process. The acknowledgment message is sent by the smaller process to a bigger process in response to the initiation message. A process is in one of two states: *passive* or *active*. Initially all processes are *active*. In the *active* state, a process sends an initiation message and turns *passive* until it receives the corresponding acknowledgment. The user message x is tagged along with either the initiation or the acknowledgment message. Thus, we have two types of protocol messages denoted by the set M_b and M_s , representing the messages from bigger to smaller, and the messages from smaller to bigger processes respectively.

When a process wants to send a message x to a smaller process it sends an initiation message, with x tagged to it, and turns *passive* until it receives the corresponding acknowledgment message. When a process wants to send a message x to a bigger process it sends a request message to the bigger process. In turn, the bigger process sends back an initiation message (when it is *active*) and turns *passive* until it receives the corresponding acknowledgment message. The smaller process tags the user message x along with the acknowledgment message.

During the *passive* state a process cannot send any message, neither an initiation nor an acknowledgment message. In addition, if the synchronization is for a message from a smaller to a bigger process, then the processes involved cannot receive initiation messages.

3.2 Proof of Correctness

We have a set of n processes p_1, p_2, \dots, p_n that communicate using three sets of messages:

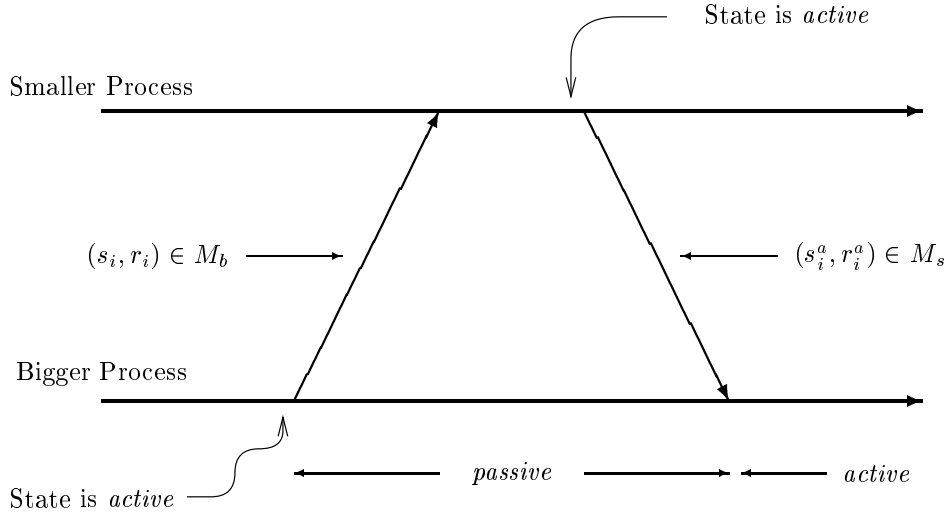


Figure 3.4: Protocol messages to implement SYNC.

1. User messages $M = \bigcup_i \{x_i\}$,
2. Initiation messages $M_b = \bigcup_i \{(s_i, r_i)\}$, and
3. Acknowledgment messages $M_s = \bigcup_i \{(s_i^a, r_i^a)\}$.

We use x_i to represent a user message and (s_i, r_i) and (s_i^a, r_i^a) for the corresponding protocol messages. If x_i is a message from a bigger to a smaller process then x_i and (s_i, r_i) represent the same message, otherwise x_i and (s_i^a, r_i^a) represent the same message. Thus, $M = M_b^u \cup M_s^u$, where M_b^u are initiation messages that are also user messages and M_s^u are acknowledgment messages that are also user messages.

We have to show that there exists a function $T : M \rightarrow \{1, 2, \dots\}$ such that

$$(g \triangleright h) \Rightarrow T(\text{Msg}(g)) < T(\text{Msg}(h)),$$

where messages $\text{Msg}(g)$ and $\text{Msg}(h)$ belong to M . Alternatively, we have to show that there exists a function $T : M_b^u \cup M_s^u \rightarrow \{1, 2, \dots\}$ such that

$$(g \prec h) \Rightarrow T(\text{Msg}(g)) < T(\text{Msg}(h)),$$

where, $(g \prec h)$ implies the event g and h happened in the same process and g

happened before h . We use the term SYNC to refer to the property of the existence of such a function for a set of messages.

A process can be in one of the two states: *active* or *passive*. The system uses the following rules:

- A process can send a message belonging to $M_b \cup M_s$ only if it is *active*.
- On sending a message $(s, r) \in M_b$ a process turns *passive*.
- On receiving a message $(s, r) \in M_b$ a process acknowledges by sending the message $(s^a, r^a) \in M_s$.
- On receiving a message $(s^a, r^a) \in M_s$ a process turns *active*.
- If the corresponding user message is from a smaller process to a bigger process, then during the *passive* state both the processes cannot receive any message belonging to M_b .

Using the first four rules we show that the protocol messages M_b satisfy the SYNC condition, that is, there exists a function $T_p : M_b \rightarrow \{1, 2, \dots\}$ such that

$$(g \prec h) \Rightarrow T_p(\text{Msg}(g)) < T_p(\text{Msg}(h)).$$

The fifth rule guarantees that for protocol messages $(s_i, r_i) \in M_b$ and $(s_i^a, r_i^a) \in M_s$ if the corresponding user message x_i is from a smaller to a bigger process, then for any event g (not the same as $s_i, r_i, s_i^a,$ or r_i^a),

$$\begin{aligned} (g \prec s_i) &\Leftrightarrow (g \prec r_i^a) \quad \wedge \quad (s_i \prec g) \Leftrightarrow (r_i^a \prec g) \quad \wedge \\ (g \prec r_i) &\Leftrightarrow (g \prec s_i^a) \quad \wedge \quad (r_i \prec g) \Leftrightarrow (s_i^a \prec g). \end{aligned}$$

Thus, given the function T_p we get the function $T : M_b^u \cup M_s^u \rightarrow \{1, 2, \dots\}$ satisfying the SYNC property, where $T((s_i^a, r_i^a)) = T_p((s_i, r_i))$ and $T((s_i, r_i)) = T_p((s_i, r_i))$.

In the rest of the section we present the properties satisfied by the protocol messages and prove the existence of a function T_p satisfying the SYNC property. The conditions satisfied by the protocol messages are:

Send Condition (SC) : A process can only send a message in *active* state. If $(s_1, r_1) \in M_b$ and $(s_2, r_2) \in M_b$ then

$$(s_1 \prec s_2) \Rightarrow (r_1 \rightarrow r_2).$$

Ack Condition (AC) : A process can only send an acknowledgment in *active* state. If $(s_1, r_1) \in M_b$ and $(s_2, r_2) \in M_b$ then

$$(s_1 \prec r_2) \Rightarrow \neg(r_2 \rightarrow r_1).$$

Priority Rule (PR) : Symmetry is broken by assuming a total order among the processes and that protocol messages (s_i, r_i) can only be sent to a smaller process. Let $(s, r) \in M_b$ then

$$\text{proc}(s) > \text{proc}(r).$$

3.2.1 Proof of Safety

Given a run, we form a directed graph $G(V, E)$ as follows. The vertex set V consists of all M_b messages in the computation. Thus, each vertex v_i represents a set of two events: send event s_i and the corresponding receive r_i . That is, $v_i = \{s_i, r_i\}$. There is an edge from v_i to v_j if

$$(s_i \prec s_j) \vee (s_i \prec r_j) \vee (r_i \prec s_j) \vee (r_i \prec r_j).$$

In each of the cases either we have $(s_i \rightarrow r_j)$. Thus, if there exists a cycle in the graph formed by the vertices v_1, v_2, \dots, v_k then

$$(s_1 \rightarrow r_2) \wedge (s_2 \rightarrow r_3) \wedge \dots \wedge (s_k \rightarrow r_1)$$

is true. The graph can be topologically sorted if and only if there are no cycles in the graph. The topological ordering satisfies the condition T_p .

Lemma 3.1 Let $(s_1, r_1) \in M_b$ and $(s_2, r_2) \in M_b$. If $(s_1 \rightarrow r_2)$ then

$$(r_1 \rightarrow r_2) \vee (s_1 \prec r_2).$$

Proof: If $s_1 \rightarrow r_2$ then

- (i) $(s_1 \prec r_2) \vee$
- (ii) $((s_1 \rightarrow r_1) \wedge (r_1 \rightarrow r_2)) \vee$
- (iii) $(\exists (s, r) \in M_b \cup M_s : (s_1 \prec s) \wedge (r \rightarrow r_2))$.

The first two cases directly satisfy the lemma. In the third case the process sends the message (s, r) only when it is active. Therefore, it would have received an acknowledgment (s_1^a, r_1^a) . Thus,

$$s_1 \rightarrow r_1 \rightarrow s_1^a \rightarrow r_1^a \rightarrow s \rightarrow r \rightarrow r_2.$$

□

Lemma 3.2 Let $(s_1, r_1) \in M_b$ and $(s_2, r_2) \in M_b$. Then

$$(s_1 \rightarrow r_2) \wedge (s_2 \rightarrow r_1) \Rightarrow (s_1 \prec r_2) \wedge (s_2 \prec r_1).$$

Proof: Let $s_1 \rightarrow r_2, s_2 \rightarrow r_1$.

Assume, without loss of generality, $\neg(s_1 \prec r_2)$. From lemma 3.1 and $(s_1 \rightarrow r_2)$, we get that $(r_1 \rightarrow r_2)$.

Since $(r_1 \rightarrow r_2)$ and the property AC of the protocol, we get $\neg(s_2 \prec r_1)$. Applying lemma 3.1 again to $\neg(s_2 \prec r_1)$ and $s_2 \rightarrow r_1$, we get that $r_2 \rightarrow r_1$, which is a contradiction. □

Lemma 3.3 If there exists a sequence of messages belonging to M_b such that

$$(s_1 \rightarrow r_2) \wedge (s_2 \rightarrow r_3) \wedge \cdots \wedge (s_k \rightarrow r_1),$$

then there also exists a sequence of messages belonging to M_b such that

$$(s'_1 \prec r'_2) \wedge (s'_2 \prec r'_3) \wedge \cdots \wedge (s'_{k'} \prec r'_1).$$

Proof: If $k \leq 2$, then the lemma follows directly from lemma 3.2. Assume $k > 2$. Pick any part of the crown starting from any index $i \bmod k$,

$$s_{i-1} \rightarrow r_i, s_i \rightarrow r_{i+1} \quad (*)$$

such that, $\neg(s_i \prec r_{i+1})$, such an i exists otherwise we have

$$(s_1 \prec r_2) \wedge (s_2 \prec r_3) \wedge \cdots \wedge (s_k \prec r_1).$$

Since $s_i \rightarrow r_{i+1}$ (by lemma 3.1), $r_i \rightarrow r_{i+1}$. Since $s_{i-1} \rightarrow r_i$ and $r_i \rightarrow r_{i+1}$, equation (*) can be reduced to $s_{i-1} \rightarrow r_{i+1}$, giving a smaller crown. Therefore, in the sequence ($k > 2$)

$$(s_1 \rightarrow r_2) \wedge (s_2 \rightarrow r_3) \wedge \cdots \wedge (s_k \rightarrow r_1),$$

if $\neg(s_i \prec r_{i+1})$ then the sequence can be reduced by removing the (s_i, r_i) message. On repeating the process the resulting sequence will eventually be one of the following:

- For some $k' \leq k$

$$(s'_1 \prec r'_2) \wedge (s'_2 \prec r'_3) \wedge \cdots \wedge (s'_{k'} \prec r'_1).$$

- The sequence length $k = 2$, that is, $(s'_1 \rightarrow r'_2) \wedge (s'_2 \rightarrow r'_1)$. By lemma 3.2,

$$(s'_1 \rightarrow r'_2) \wedge (s'_2 \rightarrow r'_1) \Rightarrow (s'_1 \prec r'_2) \wedge (s'_2 \prec r'_1). \quad \square$$

Lemma 3.4 There does not exist a sequence of messages belonging to M_b such that

$$(s_1 \prec r_2) \wedge (s_2 \prec r_3) \wedge \cdots \wedge (s_k \prec r_1).$$

Proof: The proof is by contradiction. If there exists a sequence of events such that

$$(s_1 \prec r_2) \wedge (s_2 \prec r_3) \wedge \cdots \wedge (s_k \prec r_1),$$

then

$$\left\{ \begin{array}{l} \forall i \in \{1, \dots, k\} : \text{proc}(s_i) = \text{proc}(r_{i+1 \bmod k}), \quad (*) \\ \forall i \in \{2, \dots, k\} : \text{proc}(s_i) > \text{proc}(r_i). \quad (**) \text{ (by PR)} \end{array} \right\}$$

Combining (*) and (**) we get, $\text{proc}(r_1) > \text{proc}(s_1)$, which is a contradiction to PR. □

Hence, there exists a function $T_p : M_b \rightarrow \{1, 2, \dots\}$ such that

$$(g \prec h) \Rightarrow T_p(\text{Msg}(g)) < T_p(\text{Msg}(h)).$$

3.2.2 Proof of Liveness

In a distributed computation (that implements the algorithm), we have to show that every process p_k that wants to send a message will eventually be able to send it.

If a process wants to send a message to a bigger process, then it sends a *request* message. When the bigger process is *active* the permission is granted and the process can send the message. If a process wants to send a message to a smaller process then it sends the message as soon as it becomes *active*. Thus, we show by induction that eventually all processes p_k become *active* and are able to send a message or grant permission for a message to be sent.

Base Case $k = 1$. The smallest process p_1 does not send any initiation message therefore it is always *active*. It sends the acknowledgment as soon as it gets a message $(s, r) \in M_g$.

Induction Case Now on applying induction, given that k smallest processes will eventually be in the *active* state, then the $(k + 1)$ th process if *passive* will eventually be *active*. The process p_{k+1} is *passive* at time t if

1. there exists a send of message $(s, r) \in M_g$ at time $t_0 < t$ and
2. the process is *passive* between the time interval from t_0 to t .

Therefore, there exists an acknowledgment from a process $p_{k'}$ receiving (s, r) to p_{k+1} such that,

1. the message $(s^a, r^a) \in M_s$ is in transit, or
2. send of the message will eventually be executed when the process $p_{k'}$ is *active*, where $k' \leq k$.

If the message is in transit then process p_{k+1} will eventually receive the acknowledgment and become *active*. If the second condition is true, then because $p_{k'} < p_{k+1}$, $p_{k'}$ will eventually turn *active* and execute the send of an acknowledgment. Therefore, process p_{k+1} will eventually be *active*.

3.3 Related Work

The *synchronous* communication primitives have been extensively studied as binary rendezvous that have been used in CSP [23] and Ada [1]. A number of algorithms have been suggested to implement rendezvous [10, 35, 4, 15]. In binary rendezvous the synchronization takes place with respect to time, i.e., both processes should simultaneously commit to an interaction. A similar property, synchronous ordering has been studied that is weaker than binary rendezvous. In it, the synchronization takes place with respect to concurrency [14, 29, 36].

In a binary rendezvous, a communication involves synchronization of exactly two processes. This kind of primitive was later generalized to allow communication and synchronization between an arbitrary number of processes [15, 3]. This general setting has been abstracted by Chandy and Misra [12] as the Committee Coordination problem. Other algorithms for multiway rendezvous have been suggested in [15, 3]. A similar synchronization property weaker than multiway rendezvous has been studied in [5, 17] as ABCAST and in [22] as synchronous multicast. In [5, 13, 17], the main concern is ordering of message in a faulty environment.

The communication mechanisms for asynchronous distributed systems that can implement synchronous ordering of messages operate either by assuming asymmetry in the underlying systems, or inducing asymmetry by ordering the messages or the processes. In Remote Procedure Call (RPC) [7] there is an assumption that the underlying system is a client/server model thus all the initiation messages are always from the client to the server. In a general asynchronous distributed system it will result in a deadlock. Bagrodia's rendezvous algorithm [4] imposed an order among the messages thus breaking the symmetry. This results in a $O(n)$ number of messages and a response time of $O(n^2)$. In the algorithm presented in this chapter and in [36], the symmetry is broken by using the natural order among the processes. The message complexity is $O(1)$ and the response time of $O(n)$.

The protocol based on the SYNC property resulted in a more efficient algo-

rithm then the existing ones such that Bagrodia's rendezvous [4] and Soneoka and Ibaraki [36]. The algorithm presented results in 2 or 3 messages for every user message with a time response of $O(n)$, where n is the number of processes. The message and time complexity for the rendezvous message passing are $O(n)$ and $O(n^2)$, and for the protocol presented by Soneoka and Ibaraki are 3 and $O(n)$.

3.4 Summary

In this chapter, we presented a protocol that guarantees synchronous ordering of messages. The protocol breaks the symmetry using the natural order among the processes. Therefore, the protocol followed by a process sending a message to a bigger process is different from the protocol followed for sending a message to a smaller process. When a process wants to send a message x to a smaller process it sends the message. But, if the process wants to send a message x to a bigger process then it has to request permission from the bigger process before sending the message.

Chapter 4

Forbidden Predicates

Generally, a *message ordering* specification can be characterized as a set of acceptable complete runs, that is, a subset of \mathbb{X} , where \mathbb{X} is the set of all complete runs. In Chapter 2, we studied the limitations of the three types of inhibition based protocols, that is, **general**, **tagged**, and **tagless**. The three specifications that play a key role in determining the existence of each type of protocol were:

$$\begin{aligned}\mathbb{X}_{sync} &= \{ (\mathbf{H}, \triangleright) : \neg((x_1.s \triangleright x_2.r) \wedge (x_2.s \triangleright x_3.r) \cdots (x_k.s \triangleright x_1.r)) \}, \\ &\quad \text{for any subset } \{x_1, x_2, \dots, x_k\} \text{ of } \text{Msg}(\mathbf{H}) \} \\ \mathbb{X}_{co} &= \{ (\mathbf{H}, \triangleright) : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)) \text{ for any subset } \{x, y\} \text{ of } \text{Msg}(\mathbf{H}) \} \\ \mathbb{X}_{async} &= \{ (\mathbf{H}, \triangleright) : \neg((x.s \triangleright x.s) \wedge (x.r \triangleright x.r)) \text{ for any message } x \in \text{Msg}(\mathbf{H}) \}\end{aligned}$$

That is, given a specification \mathbb{Y} there exists:

- a **general** protocol if and only if $\mathbb{X}_{sync} \subseteq \mathbb{Y}$,
- a **tagged** protocol if and only if $\mathbb{X}_{co} \subseteq \mathbb{Y}$, and
- a **tagless** protocol if and only if $\mathbb{X}_{async} \subseteq \mathbb{Y}$.

Thus, given a specification the type of protocol necessary and sufficient to implement the specification can be easily checked by the containment of the three sets \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} .

Since \mathbb{X} is an infinite set, we need a finite representation for its subsets that specify message ordering. We present a method called *forbidden predicates* that can be used to describe a large class of message ordering specifications. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering as well as others can be concisely specified using forbidden predicates. For example, the specification for causal ordering \mathbb{X}_{co} can be stated as: for all runs in \mathbb{X}_{co} , and for all pairs of messages, $\neg((s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1))$. The forbidden predicate for \mathbb{X}_{co} is $\exists (s_1, r_1), (s_2, r_2) : (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1)$. In general, a forbidden predicate can be stated as a conjunction of causality relationships between the events (send and receive).

4.1 Forbidden Predicates

In this chapter we describe *forbidden predicates* and present an algorithm to address the necessary and sufficient conditions for the existence of a protocol of each type.

Definition 4.1 A *forbidden predicate* B is defined as

$$B \equiv \exists x_1, x_2, \dots, x_m \in M : B(x_1, x_2, \dots, x_m)$$

where

$$B(x_1, x_2, \dots, x_m) = \bigwedge_{(j,k) \in J \times K} (x_j.p \triangleright x_k.q),$$

p and q represent s or r , and J, K are subsets of $\{1, 2, \dots, m\}$.

Definition 4.2 Given a forbidden predicate B , the corresponding *specification set* $\mathbb{X}_B \subseteq \mathbb{X}$ is defined as

$$\mathbb{X}_B = \{ (\mathbb{H}, \triangleright) : \neg B(x_1, \dots, x_m), \forall x_1, x_2, \dots, x_m \in M \}.$$

Notation: Let $B \equiv \exists x, y \in M : (x.s \triangleright y.s)$. We write the predicate B as $(x.s \triangleright y.s)$ dropping the quantifier \exists for ease of use. $B(a, b)$ implies the evaluation of $(x.s \triangleright y.s)$ for the instances a and b in M . Therefore, $B(a, b)$ is true if and only if $a.s \triangleright b.s$. In case of ambiguity we express the predicate as $B \equiv \exists x, y \in M : B(x, y)$.

Given two forbidden predicates B and B' for the sets \mathbb{X}_B and $\mathbb{X}_{B'}$, respectively, $B' \Rightarrow B$ iff $\mathbb{X}_B \subseteq \mathbb{X}_{B'}$. If a protocol for B guarantees that all the allowable partial orders belong to the set \mathbb{X}_B , then the same protocol guarantees that all the allowable partial orders belong to the set $\mathbb{X}_{B'}$.

Consider the example of causal ordering. The predicate can be stated as $B \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$. For each element $(\mathbb{H}, \triangleright)$ of \mathbb{X}_{co} (the corresponding specification set),

$$\forall x, y \in M : : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)).$$

Further, we can define three attributes for each message: receiving process, sending process, and color. We can use these attributes to define a range for the variables of the predicate. For example, we may be interested in runs where messages should not overtake the red marker message, that is

$$\forall x, y \in M : \text{color}(y) = \text{red} : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)).$$

In this chapter we are interested in predicates where the variables range over all messages.

Now, we characterize limit sets using forbidden predicates. For example, \mathbb{X}_{co} corresponds to the forbidden predicate $B \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$.

Lemma 4.1

1. The specification set for each of the following predicates contain \mathbb{X}_{sync} .
 - a) $B \equiv ((x_1.s \triangleright x_2.r) \wedge (x_2.s \triangleright x_3.r) \cdots (x_k.s \triangleright x_1.r))$ for any $k = 2, 3 \dots$
2. The specification set for the following predicates is \mathbb{X}_{co} .
 - a) $B_1 \equiv (x.s \triangleright y.r) \wedge (y.r \triangleright x.r)$.
 - b) $B_2 \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$.
 - c) $B_3 \equiv (x.s \triangleright y.s) \wedge (y.s \triangleright x.r)$.

3. The specification set for the following predicates is \mathbb{X}_{async} .

- a) $B \equiv (x.s \triangleright y.s) \wedge (y.s \triangleright x.s)$. b) $B \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.s)$.
c) $B \equiv (x.s \triangleright y.r) \wedge (y.r \triangleright x.s)$. d) $B \equiv (x.r \triangleright y.s) \wedge (y.r \triangleright x.s)$.
e) $B \equiv (x.r \triangleright y.r) \wedge (y.r \triangleright x.s)$. f) $B \equiv (x.r \triangleright y.r) \wedge (y.r \triangleright x.r)$.

Proof: In the first part, the intersection of all specification sets is \mathbb{X}_{sync} , follows from Theorem 2.1. For the third part, each of the predicates implies the existence of an event $h \in H$ such that $h \triangleright h$. No run in \mathbb{X}_{async} satisfies such a predicate. Therefore, the specification set for the predicates is \mathbb{X}_{async} .

In the second part, B_2 corresponds to \mathbb{X}_{co} by definition. We will show $B_1 \Leftrightarrow B_2$; the proof of $B_2 \Leftrightarrow B_3$ is similar. Let the corresponding specification sets be \mathbb{X}_1 and \mathbb{X}_2 , respectively. We have to show that $\mathbb{X}_1 = \mathbb{X}_2$. It is easy to see that $B_2 \Rightarrow B_1$. Since $B_2 \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright x.r)$ and $y.s \triangleright y.r$ is true, $B_2 = (x.s \triangleright y.s) \wedge (y.r \triangleright x.r) \wedge (y.s \triangleright y.r)$. Combining the first and third conjuncts, we get $B_2 \Rightarrow (x.s \triangleright y.r) \wedge (y.r \triangleright x.r) \equiv B_1$. Therefore, $\mathbb{X}_1 \subseteq \mathbb{X}_2$.

We now show that $\overline{\mathbb{X}}_1 \subseteq \overline{\mathbb{X}}_2$ where, $\overline{\mathbb{X}}_1 = \mathbb{X} - \mathbb{X}_1$. Using the definition of \mathbb{X}_1 , we get the complement of \mathbb{X}_1 as

$$\overline{\mathbb{X}}_1 = \{ (\mathbb{H}, \triangleright) : \exists x, y \in M \text{ such that } B_1(x, y) \}.$$

Let $(\mathbb{H}, \triangleright) \in \overline{\mathbb{X}}_1$. We have to show $(\mathbb{H}, \triangleright) \in \overline{\mathbb{X}}_2$. In the run $(\mathbb{H}, \triangleright)$, we have at least two messages x and y such that $(x.s \triangleright y.s) \wedge (y.s \triangleright x.r)$.

1. Let $x.s$ and $y.s$ be in different processes.

Since $(x.s \triangleright y.s)$, and $x.s$ and $y.s$ are in different processes, there exists a message z such that $(x.s \triangleright z.s)$, $(z.s \triangleright z.r)$, and $(z.r \triangleright y.s)$. Since $(y.s \triangleright x.r)$ and $(z.r \triangleright y.s)$, $z.r \triangleright x.r$. Therefore, $x.s \triangleright z.s$ and $z.r \triangleright x.r$, thus $B_2(x, z)$ is true.

2. Assume $x.s$ and $y.s$ are in the same process (therefore, $x.r$ and $y.s$ are in different processes).

Since $(y.s \triangleright x.r)$ and $x.r$ and $y.s$ are in different processes, $y.r \triangleright x.r$ or $\exists z \in M$, such that $(y.s \triangleright z.s)$, $(z.s \triangleright z.r)$, and $(z.r \triangleright x.r)$.

- (a) If $y.r \triangleright x.r$, then $(x.s \triangleright y.s)$ and $(y.r \triangleright x.r)$. Thus $B_2(x, y)$ is true.
- (b) If $\exists z : (y.s \triangleright z.s), (z.s \triangleright z.r)$, and $(z.r \triangleright x.r)$; then $(x.s \triangleright y.s) \wedge (y.s \triangleright z.s) \Rightarrow (x.s \triangleright z.s)$ and $(z.r \triangleright x.r)$, thus $B_2(x, z)$ is true.

Therefore, $\exists x, z \in M$ such that $B_2(x, z)$ is true. Thus, $(H, \triangleright) \in \overline{X}_2$. □

4.2 Specification Graph

In this section we classify the forbidden predicates to determine the type of algorithm necessary and sufficient to guarantee safety and liveness.

Definition 4.3 Let $B \equiv \exists x_1, \dots, x_m \in M : B(x_1, \dots, x_m)$ be a forbidden predicate. A *predicate graph* $G_B(V, E)$ is a multi-graph such that

$$V = \{x_1, \dots, x_m\}$$

$$E = \{(x_j, x_k) \mid (x_j.p \triangleright x_k.q) \text{ is a conjunct of } B \text{ where each } p, q \text{ is } s \text{ or } r \}$$

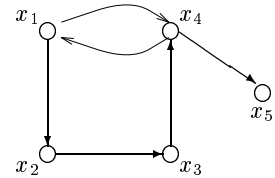
Example 4.1 Let a predicate be

$$B \equiv \left\{ \begin{array}{l} (x_1.r \triangleright x_2.s) \wedge (x_2.s \triangleright x_3.s) \wedge (x_3.r \triangleright x_4.r) \wedge \\ (x_4.s \triangleright x_1.r) \wedge (x_4.s \triangleright x_5.r) \wedge (x_1.s \triangleright x_4.r) \end{array} \right\},$$

then $G_B(V, E)$ is

$$V = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6\}, \text{ and}$$

$$E = \{(x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_1), (x_4, x_5), (x_1, x_4)\}.$$



Using the graph, we can determine whether the specification is implementable, and if it is, the type of protocol necessary and sufficient to guarantee safety and liveness.

Theorem 4.1 A specification \mathbb{X}_B (or forbidden predicate B) is implementable if and only if there exists a cycle in the predicate graph $G_B(V, E)$.

Proof: We first prove the “only if” part. Let the predicate be $B \equiv \exists x_1, \dots, x_m \in M : B(x_1, \dots, x_m)$ such that the predicate graph $G_B(V, E)$ does not have a cycle and let the corresponding specification set be \mathbb{X}_B . Consider a run $(\mathbb{H}, \triangleright)$ such that the set of messages is $Msg(\mathbb{H}) = \{x_1, \dots, x_m\}$. The run is constructed such that if $x_j.p \triangleright x_k.q$ is a conjunct of $B(x_1, \dots, x_m)$ then $(x_j.p, x_k.q) \in \triangleright$. For each message $x \in Msg(\mathbb{H})$, $(x.s, x.r) \in \triangleright$. Now take the transitive closure ($^+$) to make it a run. Therefore, $(h, h') \in \triangleright$ if one of the following conditions hold.

1. The events h and h' are $x_j.p$ and $x_k.q$, respectively, and $x_j.p \triangleright x_k.q$ is a conjunct of $B(x_1, x_2, \dots, x_m)$.
2. The events h and h' are send and delivery events of a message, that is, there exists a message $y \in Msg(\mathbb{H})$ such that $h \equiv y.s$ and $h' \equiv y.r$.
3. There exists another event $g \in \mathbb{H}$ such that $(h, g) \in \triangleright$ and $(g, h') \in \triangleright$.

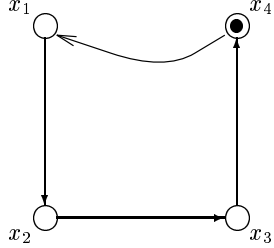
It is easy to see that the predicate B is true in the run $(\mathbb{H}, \triangleright)$, therefore, $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$. We claim that $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$, hence the theorem (only if) follows. Since the predicate graph does not have any cycles, it can be linearly ordered. Using the same ordering we define a function $T : Msg(\mathbb{H}) \rightarrow \{1, 2, 3, \dots\}$, such that for any two events $h, g \in \mathbb{H}$, if $h \triangleright g$ and $Msg(h) \neq Msg(g)$ then $T(Msg(h)) < T(Msg(g))$. Therefore, $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$ and $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$. From corollary 2.1, we have that there exists a protocol only if $\mathbb{X}_{sync} \subseteq \mathbb{X}_B$.

The “if” part follows from theorem 4.2 which will be proved in Section 4.3.

□

Since a specification graph without cycles is not implementable, thus we are interested in the specification graphs with cycles. Pick any cycle $G_c(V^c, E^c) \subseteq G(V, E)$ in the specification graph and let the corresponding forbidden predicate be B_c .

Example 4.2 Consider the forbidden predicate and the graph from example 4.1. A possible cycle and the corresponding predicate is shown below. It is easy to see that $B \Rightarrow B_c$, since B_c is the same as B with some conjuncts removed.

$$\begin{aligned}
 V^c &= \{x_1, x_2, x_3, x_4\} \\
 E^c &= \{(x_1, x_2), (x_2, x_3), (x_3, x_4), (x_4, x_1)\} \\
 P_c &= \left\{ \begin{array}{l} (x_1.r \triangleright x_2.s) \wedge (x_2.s \triangleright x_3.s) \wedge \\ (x_3.r \triangleright x_4.r) \wedge (x_4.s \triangleright x_1.s) \end{array} \right\}
 \end{aligned}$$


The specification graphs can contain a number of cycles. We classify a cycle into different categories based on the number of β vertices (defined next) it contains.

Definition 4.4 Given a cycle $G_c(V^c, E^c)$ in the graph $G(V, E)$, we say $x \in V^c$ is a β vertex with respect to the cycle $G_c(V^c, E^c)$ if the incoming edge is $\zeta \triangleright x.r$ where ζ is either $y.s$ or $y.r$ and the outgoing edge is $x.s \triangleright \delta$ where δ is either $z.s$ or $z.r$. The *order* of a cycle is equal to the number of β vertices it contains.

Example 4.3 (Continuing with the previous example.) With respect to the cycle $G_c(V^c, E^c)$, only x_4 is a β vertex, thus the order of the cycle is 1. Consider a non- β vertex, say x_3 . Consider the conjuncts that result in the input and output edges of the vertex x_3 . They are, $x_2.s \triangleright x_3.s$ and $x_3.r \triangleright x_4.r$. Since $x_3.s \triangleright x_3.r$, combining the three conjuncts we get, $x_2.s \triangleright x_4.r$. We can get a predicate B' ,

$$B' \equiv (x_1.r \triangleright x_2.s) \wedge (x_2.s \triangleright x_4.r) \wedge (x_4.s \triangleright x_1.r),$$

such that $B_c \Rightarrow B'$. Since $B \Rightarrow B_c$ and $B_c \Rightarrow B'$, $B \Rightarrow B'$. If we consider the predicate graph $G_{B'}(V', E')$, it is a cycle of order 1 and the β vertex is x_4 , thus maintaining the order and the β vertex of the cycle.

Lemma 4.2 Let B be a predicate and $G_B(V, E)$ be the corresponding predicate graph with a cycle of order k . Then there exists a predicate B' weaker than B whose predicate graph $G_{B'}(V', E')$ is a cycle of order k such that

1. $|V'| = 2$, or
2. all the vertices are β vertices.

Proof: Let $G(V^c, E^c) \subseteq G(V, E)$ be a cycle in the predicate graph with the corresponding predicate B_c . We know that $B \Rightarrow B_c$.

If the graph $G(V', E') = G(V^c, E^c)$ and predicate $B' = B_c$ satisfy the condition of the lemma, we are done. If not, pick a vertex, say y , that is not a β vertex. Then one of the following is true, with $x \neq z$,

1. $B' \equiv \dots (x.p \triangleright y.s) \wedge (y.s \triangleright z.q) \wedge \dots$, or
2. $B' \equiv \dots (x.p \triangleright y.s) \wedge (y.r \triangleright z.q) \wedge \dots$, or
3. $B' \equiv \dots (x.p \triangleright y.r) \wedge (y.r \triangleright z.q) \wedge \dots$.

Such a vertex exists since the graph (cycle) has more than two vertices and has at least one non- β vertex. In each case, $B' \Rightarrow B''$, where $B'' \equiv \dots \wedge (x.p \triangleright z.q) \wedge \dots$. Since $B \Rightarrow B'$ and $B' \Rightarrow B''$, $B \Rightarrow B''$. Let the graph predicate for B'' be $G(V'', E'')$. The graph $G(V'', E'')$ satisfies the condition $|V''| = |V'| - 1$, and the number of β vertices in $G(V'', E'')$ is k .

If the graph $G(V'', E'')$ and the corresponding predicate B'' satisfy the conditions of the lemma, we are done, otherwise repeat the above process. \square

4.3 Impossibility and Lower-Bounds

In this section we prove the necessary and sufficient conditions for a specification to be implementable by a protocol of a given class. The next theorem proves the sufficient condition for a protocol to implement a given specification. Theorem 4.3 presents the necessary conditions to be satisfied by the specifications to be implementable by a protocol of a given class.

Theorem 4.2 (Sufficient Conditions) Let \mathbb{X}_B be a specification with B as the corresponding forbidden predicate. Let the predicate graph be $G_B(V, E)$ with a

cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$.

1. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, then $\mathbb{X}_{async} \subseteq \mathbb{X}_B$.
2. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 1, then $\mathbb{X}_{co} \subseteq \mathbb{X}_B$.
3. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order $k (> 1)$, then $\mathbb{X}_{sync} \subseteq \mathbb{X}_B$.

Proof:

Part 1. Let $G_c(V^c, E^c) \subseteq G_B(V, E)$ be a cycle of order 0. Then from Lemma 4.2, there exists a predicate B' such that $B \Rightarrow B'$ and the corresponding graph $G_{B'}(V', E')$ is a cycle of order 0 with $|V'| = 2$. Since $B \Rightarrow B'$, $\mathbb{X}_{B'} \subseteq \mathbb{X}_B$.

Since the graph is a cycle with two vertices (both non- β), the predicate $B' \equiv \exists x, y : B'(x, y)$ can only be one of the predicates in the statement of Lemma 4.1.3. From Lemma 4.1 we have that the specification corresponding to the above predicates are equivalent to \mathbb{X}_{async} . Therefore, $\mathbb{X}_{async} = \mathbb{X}_{B'}$, and $\mathbb{X}_{async} \subseteq \mathbb{X}_B$.

Part 2. We have to show $\mathbb{X}_{co} \subseteq \mathbb{X}_B$. Let $G_c(V^c, E^c) \subseteq G_B(V, E)$ be a cycle of order 1. Then from Lemma 4.2, there exists a predicate $B \Rightarrow B'$. The corresponding graph $G_{B'}(V', E')$ is a cycle of order 1, such that $|V'| = 2$. Since $B \Rightarrow B'$, $\mathbb{X}_{B'} \subseteq \mathbb{X}_B$.

Since the graph is a cycle with two vertices (one β), the predicate $B' \equiv \exists x, y : B'(x, y)$ can only be one of the predicates in the statement of Lemma 4.1.2. From Lemma 4.1, the specification corresponding to the above predicates is equivalent to \mathbb{X}_{co} . Therefore, $\mathbb{X}_{co} = \mathbb{X}_{B'}$, and $\mathbb{X}_{co} \subseteq \mathbb{X}_B$.

Part 3. We have to show $\mathbb{X}_{sync} \subseteq \mathbb{X}_B$. Let $G_c(V^c, E^c) \subseteq G_B(V, E)$ be a cycle of order $k (> 1)$. Then from Lemma 4.2, there exists a predicate $B \Rightarrow B'$. The corresponding graph $G_{B'}(V', E')$ is a cycle of order k , such that $|V'| = k$. Since $B \Rightarrow B'$, $\mathbb{X}_{B'} \subseteq \mathbb{X}_B$.

Since the graph is a cycle with k β vertices, the predicate B' is

$$B' \equiv (x_{1.s} \triangleright x_{2.r}) \wedge (x_{2.s} \triangleright x_{3.r}) \cdots (x_{k.s} \triangleright x_{1.r}).$$

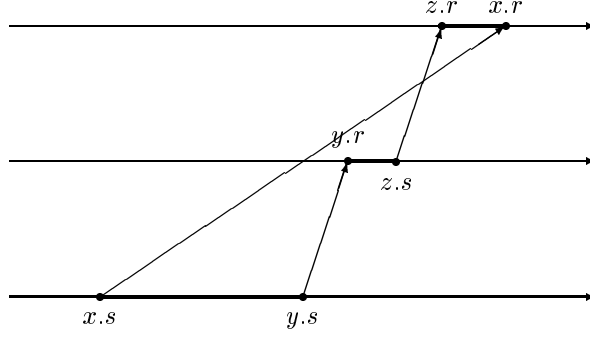


Figure 4.1: Construction of a run using a forbidden predicate.

This implies the predicate in the statement of Lemma 4.1.1. Therefore, $\mathbb{X}_{sync} \subseteq \mathbb{X}_{B'} \subseteq \mathbb{X}_B$. \square

The next theorem employs a technique of constructing a run $(\mathbb{H}, \triangleright)$ using a forbidden predicate. We illustrate the methodology used by considering the following example.

Example 4.4 Let us consider the following forbidden predicate

$$B(x, y, z) \equiv (x.s \triangleright y.s) \wedge (y.r \triangleright z.s) \wedge (z.r \triangleright x.r).$$

We construct a run $(\mathbb{H}, \triangleright)$ such that the set of messages is $Msg(\mathbb{H}) = \{x, y, z\}$. The causality relation \triangleright is a transitive closure of the following set.

$$\{ (x.s \triangleright y.s), (y.r \triangleright z.s), (z.r \triangleright x.r), (x.s \triangleright x.r), (y.s \triangleright y.r), (z.s \triangleright z.r) \}.$$

Note that the first three elements are conjuncts of the forbidden predicate and next three elements are the causality induced by the fact that x, y , and z are messages. Figure 4.1 shows one possible run given by the above construction. In the figure, the thick lines correspond to the conjuncts of B .

Now consider the two events $x.s$ and $z.s$, clearly $(x.s \triangleright z.s)$. Which can be rewritten as

$$(x.s \triangleright y.s) \wedge (y.s \triangleright y.r) \wedge (y.r \triangleright z.s),$$

or $C_1 \wedge C_2 \wedge C_3$, where

$$C_1 \equiv (x.s \triangleright y.s), C_2 \equiv (y.s \triangleright y.r), \text{ and } C_3 \equiv (y.r \triangleright z.s).$$

Each of the C s is either a conjunct of B or is of the form $(a.s \triangleright a.r)$. In the above case, C_1 and C_3 are conjuncts of B and C_2 is of the form $(a.s \triangleright a.r)$.

Theorem 4.3 (Necessary Conditions) Let \mathbb{X}_B be a specification with B as the corresponding forbidden predicate. Let the predicate graph be $G_B(V, E)$ with a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$.

1. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, 1 or n , then $\mathbb{X}_{sync} \not\subseteq \mathbb{X}_B$.
2. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0 or 1, then $\mathbb{X}_{co} \not\subseteq \mathbb{X}_B$.
3. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, then $\mathbb{X}_{async} \not\subseteq \mathbb{X}_B$.

Proof:

Part 1. If there does not exist a cycle of order 0, 1, or n , then there does not exist a cycle. From theorem 4.1 it follows that $\mathbb{X}_{sync} \not\subseteq \mathbb{X}_B$.

Part 2. We have to show that $\mathbb{X}_{co} \not\subseteq \mathbb{X}_B$, given there does not exist a cycle of order 0 or 1 in the predicate graph. We will construct a run $(\mathbb{H}, \triangleright)$ and show that $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$, but $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$.

Let the forbidden predicate be $B(x_1, \dots, x_m)$ with the corresponding predicate graph having no cycles of order 0 or 1. Consider a run $(\mathbb{H}, \triangleright)$ such that the set of messages is $Msg(\mathbb{H}) = \{x_1, \dots, x_m\}$. The run is constructed such that if $(x_j.p \triangleright x_k.q)$ is a conjunct of $B(x_1, \dots, x_m)$ then $(x_j.p, x_k.q) \in \triangleright$. For each message $x \in Msg(\mathbb{H})$, $(x.s, x.r) \in \triangleright$. Now take the transitive closure $(+)$ to make it a run. Therefore, $(h, h') \in \triangleright$ if one of the following conditions hold.

1. The events h and h' are $x_j.p$ and $x_k.q$, respectively, and $(x_j.p \triangleright x_k.q)$ is a conjunct of $B(x_1, x_2, \dots, x_m)$.
2. The events h and h' are send and delivery events of a message, that is, there exists a message $y \in \text{Msg}(\mathbb{H})$ such that $h \equiv y.s$ and $h' \equiv y.r$.
3. There exists another event $g \in \mathbb{H}$ such that $(h, g) \in \triangleright$ and $(g, h') \in \triangleright$.

Since for this run $B(x_1, \dots, x_n)$ is true, $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$. The claim is $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$. We will show $(\mathbb{H}, \triangleright) \in \mathbb{X}_{co}$ by contradiction.

Assume $(\mathbb{H}, \triangleright) \notin \mathbb{X}_{co}$. From the definition of \mathbb{X}_{co} ,

$$\mathbb{X}_{co} = \{ (\mathbb{H}, \triangleright) : \neg((x.s \triangleright y.s) \wedge (y.r \triangleright x.r)), \forall x, y \in \text{Msg}(\mathbb{H}) \}.$$

Therefore, $\exists x_i, x_j \in \text{Msg}(\mathbb{H})$ such that $(x_i.s \triangleright x_j.s) \wedge (x_j.r \triangleright x_i.r)$ is true.

We can rewrite $(x_i.s \triangleright x_j.s) \wedge (x_j.r \triangleright x_i.r)$ as

$$(C_1^1 \wedge C_2^1 \wedge \dots \wedge C_p^1) \bigwedge (C_1^2 \wedge C_2^2 \wedge \dots \wedge C_q^2),$$

where C_j^i s are either a conjunct of B or of the form $(x_k.s \triangleright x_k.r)$ for some k . The first group of C s, i.e., $(C_1^1 \wedge C_2^1 \wedge \dots \wedge C_p^1)$ is equal to $(x_i.s \triangleright x_j.s)$ and the second group of C s, i.e., $(C_1^2 \wedge C_2^2 \wedge \dots \wedge C_q^2)$ is equal to $(x_j.r \triangleright x_i.r)$.

We form a graph (cycle) from the C s. Drop all the C s which are not conjuncts of B , since they do not contribute to the cycle. Since the remaining C are conjuncts of the predicate B , every edge in the graph formed by the C s has a corresponding edge in the predicate graph $G_B(V, E)$.

Consider the predicate graph formed by the resulting C s. Let the predicate be B_c and $G_{B_c}(V^c, E^c)$. It is a cycle and $G_{B_c}(V^c, E^c) \subseteq G_B(V, E)$. For each C remaining there is an edge. We have to analyze the vertex formed by two C s. If C_i is of form $(x.s \triangleright x.r)$ (thus dropped) then $C_{i-1} = (\zeta \triangleright x.s)$ and $C_{i+1} = (x.r \triangleright \delta)$, thus the vertex formed by C_{i-1} and C_{i+1} is not a β vertex.

Let us consider the case when C_i and C_{i+1} are parts of the conjunct. Then, $C_i = (\zeta \triangleright x.h)$ and $C_{i+1} = (x.h \triangleright \delta)$. Thus the vertex formed by C_i and C_{i+1} is not a β vertex.

Therefore the vertices formed by the C 's in the same group do not result in any β vertex. There are two more vertices to be considered – the vertices formed by the group joining. Note that C_p^1 and C_1^2 are conjuncts of B since they cannot be of the form $(x.s \triangleright x.r)$. The conjuncts are of the form $(\zeta \triangleright x_j.s)$ and $(x_j.r \triangleright \delta)$, respectively. Therefore, the vertex formed by joining C_p^1 and C_1^2 results in a non- β vertex.

Therefore, the number of vertices left to be considered is one (it may or may not be a β vertex). Thus the resulting graph is of order 0 or 1. Thus, there exists a cycle $G_{B^c}(V^c, E^c)$ in the predicate graph of the predicate B of order 0 or 1, which is a contradiction.

Part 3. We have to show that $\mathbb{X}_{async} \not\subseteq \mathbb{X}_B$ given that there does not exist a cycle of order 0 in the predicate graph $G_B(V, E)$.

Let us assume that the predicate graph does not have a cycle of order 0. We construct a run $(\mathbb{H}, \triangleright)$ and show that $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$ but $(\mathbb{H}, \triangleright) \in \mathbb{X}_{async}$.

Let the forbidden predicate be $B(x_1, \dots, x_n)$. Consider a run $(\mathbb{H}, \triangleright)$ such that the set of messages is $Msg(\mathbb{H}) = \{x_1, \dots, x_m\}$. The run is constructed such that if $(x_j.p \triangleright x_k.q)$ is a conjunct of $B(x_1, \dots, x_m)$ then $(x_j.p, x_k.q) \in \triangleright$. For each message $x \in Msg(\mathbb{H})$, $(x.s, x.r) \in \triangleright$. Now take the transitive closure $(^+)$ to make it a run. Therefore, $(h, h') \in \triangleright$ if one of the following conditions hold.

1. The events h and h' are $x_j.p$ and $x_k.q$, respectively, and $(x_j.p \triangleright x_k.q)$ is a conjunct of $B(x_1, x_2, \dots, x_m)$.
2. The events h and h' are send and delivery events of a message, that is, there exists a message $y \in Msg(\mathbb{H})$ such that $h \equiv y.s$ and $h' \equiv y.r$.
3. There exists another event $g \in \mathbb{H}$ such that $(h, g) \in \triangleright$ and $(g, h') \in \triangleright$.

Since for this run $B(x_1, \dots, x_n)$ is true, $(\mathbb{H}, \triangleright) \notin \mathbb{X}_B$. The claim is $(\mathbb{H}, \triangleright) \in \mathbb{X}_{async}$. We will show $(\mathbb{H}, \triangleright) \in \mathbb{X}_{async}$ by contradiction.

Assume $(\mathbf{H}, \triangleright) \notin \mathbb{X}_{async}$. From the definition of \mathbb{X}_{async} ,

$$\mathbb{X}_{async} = \{ (\mathbf{H}, \triangleright) : \nexists h \in \mathbf{H} \text{ such that } h \triangleright h \}.$$

Therefore, $\exists h \in \mathbf{H}$ such that $(h \triangleright h)$.

We can rewrite $(h \triangleright h)$ as $(C_1 \wedge C_2 \wedge \dots \wedge C_p)$ where C is either a conjunct of B or of the form $(x_k.s \triangleright x_k.r)$ for some k .

We form a graph (cycle) from the C s. Drop all the C s which are not a conjunct of B , since they do not contribute to the cycle.

Consider the predicate graph formed by the resulting C s. Let the predicate be B_c and $G_c(V^c, E^c)$. It is a cycle and $G_c(V^c, E^c) \subseteq G(V, E)$. By similar reasoning as in the previous case the only β vertex that can be possible is between C_1 and C_p .

1. C_1 is of the form $(x.s \triangleright x.r)$ (thus dropped), then C_p and C_2 are of the form $(\zeta \triangleright x.s)$ and $(x.r \triangleright \delta)$, respectively. The vertex formed by C_p and C_2 is not a β vertex.
2. C_p is of the form $(x.s \triangleright x.r)$ (thus dropped), then C_{p-1} and C_1 are of the form $(\zeta \triangleright x.s)$ and $(x.r \triangleright \delta)$, respectively. The vertex formed by C_{p-1} and C_1 is not a β vertex.
3. C_1 and C_p are not of the form $(x.s \triangleright x.r)$. Then C_p and C_1 are of the form $(\zeta \triangleright x.f)$ and $(x.f \triangleright \delta)$, respectively. The vertex formed by C_p and C_1 is not a β vertex.

Thus the resulting graph is of order 0. Thus, there exist a cycle $G_{B_c}(V^c, E^c)$ in the predicate graph of the predicate B of order 0, which is a contradiction. \square

4.4 Related Work

Many communication and synchronization schemes for distributed systems have been proposed in the literature. Examples include: various broadcast and multicast schemes like ABCAST, CBCAST, FBCAST [5, 34, 6]; flush channels as weakening

of the FIFO-protocol [2]; global flush channels as weakening of causal ordering [21]; causal ordering [5, 32, 33]; and synchronous ordering [36, 14, 29]. In addition, many protocols induce some message orderings among the user messages or control messages [11, 20, 27, 19, 25, 24, 9]. Although there has been a fair amount of research in the area of message orderings, neither a succinct representation for all the message orderings nor a formal treatment to study the relationship between the orderings has been done. The method of forbidden predicates characterizes these and many new orderings.

4.5 Summary

A distributed computation or run describes an execution of a distributed program. At an abstract level a run is a partial order (H, \triangleright) , where H is the set of events in the system and \triangleright the happened-before relation between events. Generally, a *message ordering* specification can be characterized as a set of acceptable complete runs, that is, a subset of \mathbb{X} , where \mathbb{X} is the set of all complete runs.

Since \mathbb{X} is an infinite set, we presented a method called *forbidden predicates* that can be used to describe a large class of message ordering specifications. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering as well as others can be concisely specified using forbidden predicates. For example, the specification for causal ordering \mathbb{X}_{co} can be stated as: for all runs in \mathbb{X}_{co} , and for all pairs of messages, $\neg((s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1))$. The forbidden predicate for \mathbb{X}_{co} is $\exists(s_1, r_1), (s_2, r_2) : (s_1 \triangleright s_2) \wedge (r_2 \triangleright r_1)$. In general, a forbidden predicate can be stated as a conjunction of causality relationships between the events (send and receive).

Given a message ordering specification using forbidden predicates, we present an algorithm that determines the type of protocol necessary to implement that specification. The algorithm converts the forbidden predicate into a predicate graph. It is shown that the specification can be implemented if and only if there is a cycle

in this graph. Further, to determine the nature of the protocol required for the specification, it is sufficient to examine vertices of the graph. We define the notion of β vertices. If the cycle has two or more β vertices with respect to that cycle, then control messages are necessary. If the cycle has one β vertex, then tagging user messages is sufficient. If the cycle has no β vertex, then no action from the protocol is required. Thus, given any message ordering specification using forbidden predicates, the nature of the protocol necessary for implementing it can easily be determined. This can be summed up by the following table which is a consequence of the last two theorems proved in this chapter:

Specification graph has a cycle	\Leftrightarrow	specification is implementable
and if there exists a cycle with		
– zero or more β vertices	\Leftrightarrow	tagging and control messages are sufficient,
– zero or one β vertex	\Leftrightarrow	tagging is sufficient, and
– zero β vertices	\Leftrightarrow	trivial protocol is sufficient.

Chapter 5

Algorithm to Implement Message Ordering

The focus of this chapter is to find a general algorithm that can implement a large class of message orderings without using control messages. In the previous chapter we showed that this is possible only if the predicate graph, derived from the forbidden predicate, has a cycle with zero or one β vertices. We will not consider the trivial case where there exists a cycle with zero β vertices.

5.1 Extensions to Forbidden Predicates

In this chapter, we extend the concept of forbidden predicates by defining three attributes for each message x , they are: $color(x)$ – the color of a message, $proc(x.s)$ – the process identifier of the sending process, and $proc(x.r)$ – the process identifier of the receiving process.

The predicate graph for the specifications considered in this chapter are cycles with one β vertex. Without loss of generality, let $V = \{x_1, x_2, \dots, x_m\}$ and $E = \{(x_i, x_{(i \bmod m)+1}) : i = 1, 2, \dots, m\}$ be the vertex and edge set of the resulting graph, with x_1 being the β vertex. In addition, each message x_i has to

satisfy $c_i(x_i)$ where $c_i(\cdot)$ is a trivial predicate, i.e., a function of color or the sending process identifier of the message. Thus a general forbidden predicate can be written as:

$$B \equiv \exists x_1, \dots, x_m \in M : c_1(x_1) \wedge \dots \wedge c_m(x_m) \\ : (x_1.s \triangleright x_2.p) \wedge \left(\bigwedge_{i=2, \dots, m-1} (x_i.p' \triangleright x_{i+1}.q') \right) \wedge (x_m.q \triangleright x_1.r),$$

where each p, p', q' and q stand for either s or r . The $(i-1)$ th and i th clauses in the predicate can be written as

$$(\zeta \triangleright x_i.s) \wedge (x_i.s \triangleright \delta), \quad (\zeta \triangleright x_i.s) \wedge (x_i.r \triangleright \delta), \quad \text{or} \quad (\zeta \triangleright x_i.r) \wedge (x_i.r \triangleright \delta),$$

otherwise, x_i is a β vertex, and we say, the vertex x_i is of the type (s, s) , (s, r) , or (r, r) . We introduce a shorthand to represent the $(i-1)$ th and i th clauses as $\zeta \triangleright l_i(x_i) \triangleright \delta$, where

$$l_i(a) = \begin{cases} a.s & \text{if vertex } x_i \text{ is of type } (s, s) \\ a.s \triangleright a.r & \text{if vertex } x_i \text{ is of type } (s, r) \\ a.r & \text{if vertex } x_i \text{ is of type } (r, r). \end{cases}$$

Thus, if a run (H, \triangleright) is invalid under a specification B , then there exists a set of messages $\{a_1, a_2, \dots, a_m\} \subseteq \text{Msg}(H)$ such that $c_i(a_i)$ is true for all i and

$$a_1.s \triangleright l_2(a_2) \triangleright l_3(a_3) \triangleright \dots \triangleright l_m(a_m) \triangleright a_1.r.$$

Some examples of forbidden predicates that can be implemented without using control messages are:

FIFO: Messages are received in the order that they are sent between any pair of processes:

$$\exists x, y : \\ (\text{proc}(x.s) = \text{proc}(y.s)) \wedge (\text{proc}(x.r) = \text{proc}(y.r)) : \\ (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

Colored FIFO: Messages of different kinds (or different colors) are received in the same order that they are sent between any pair of processes:

$$\begin{aligned} & \exists x, y : \\ & (\text{proc}(x.s) = \text{proc}(y.s)) \wedge (\text{proc}(x.r) = \text{proc}(y.r)) \wedge (\text{color}(x) \neq \text{color}(y)) : \\ & (x.s \triangleright y.s) \wedge (y.r \triangleright x.r). \end{aligned}$$

Causal Ordering: A series of messages cannot overtake another message:

$$\exists x, y : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

Colored Causal Ordering: Ordering among messages with different colors is maintained:

$$\exists x, y : (\text{color}(x) \neq \text{color}(y)) : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

Similar predicates are used in many consistent-cut protocols.

k -Weaker Causal Ordering: Messages can be out of order by at most k messages:

$$\exists x_1, \dots, x_{k+1} : (x_1.s \triangleright x_2.s) \wedge (x_2.s \triangleright x_3.s) \wedge \dots \wedge (x_{k+2}.r \triangleright x_1.r).$$

Local Forward-Flush: All messages sent before a *red* message are received before the *red* message between any pair of processes:

$$\begin{aligned} & \exists x, y : \\ & (\text{proc}(x.s) = \text{proc}(y.s)) \wedge (\text{proc}(x.r) = \text{proc}(y.r)) \wedge (\text{color}(y) = \text{red}) : \\ & (x.s \triangleright y.s) \wedge (y.r \triangleright x.r). \end{aligned}$$

Local Backward-Flush : All messages sent after a *red* message are received after the *red* message between any pair of processes:

$$\begin{aligned} & \exists x, y : \\ & (\text{proc}(x.s) = \text{proc}(y.s)) \wedge (\text{proc}(x.r) = \text{proc}(y.r)) \wedge (\text{color}(x) = \text{red}) : \\ & (x.s \triangleright y.s) \wedge (y.r \triangleright x.r). \end{aligned}$$

Global Forward–Flush: All messages sent before a *red* message are received before the *red* message:

$$\exists x, y : (\text{color}(y) = \text{red}) : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

Global Backward–Flush: All messages sent after a *red* message are received after the *red* message:

$$\exists x, y : (\text{color}(x) = \text{red}) : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

5.2 Algorithm for a Two Clause Predicate

Consider the following specification:

a *red* message sent before a *green* message should not be received after the *green* message, i.e.,

$$(x.s \triangleright y.s) \Rightarrow \neg(y.r \triangleright x.r),$$

where $\text{color}(x) = \text{red}$ and $\text{color}(y) = \text{green}$.

In this case, upon receiving a *green* message a process waits for only those *red* messages in transit that were sent before the *green* message. Thus a process has to keep track of two types of *red* messages: first, those *red* messages in transit that were sent before a *green* message, and second those *red* messages that might precede a future *green* message. This is done by keeping two *level* sets L_1 and L_2 , where

$L_1 \equiv$ set of all *red* messages, and

$L_2 \equiv$ set of *red* messages that are preceded by a past *green* message.

The above condition can be rewritten using forbidden predicates as,

$$B \equiv \exists x, y : \text{color}(x) = \text{red} \wedge \text{color}(y) = \text{green} : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

Informally, a forbidden predicate states that a run is illegal if a set of events satisfies certain causality relations. Therefore using sets L_1 and L_2 , the algorithm tracks all

$$\exists x, y : \text{color}(x) = \text{red} \wedge \text{color}(y) = \text{green} : \underbrace{(x.s \triangleright y.s)}_{L_1} \quad \wedge \quad \underbrace{(y.r \triangleright x.r)}_{L_2}$$

For any event g ,

$$a \in L_1[g] \Rightarrow (a.s \succeq g),$$

where $\text{color}(a) = \text{red}$

$$a \in L_2[g] \Rightarrow \exists b : (a.s \triangleright b.s) \wedge (b.r \succeq g),$$

where $\text{color}(b) = \text{green}$

Figure 5.1: Detection of different stages of the predicate.

possible combinations of events that may eventually satisfy the causality relations. To guarantee safety a process delays an event until it has received all messages in L_2 . This ensures the second clause of the predicate is never true for any x, y . Therefore, the two steps in the algorithm are:

1. Detection of different stages of the predicate.
2. Avoidance of one of the clauses of the predicate.

5.2.1 Detection

The predicate can be viewed as a causality chain, and the first step keeps track of all events that have satisfied part of the causality chain. Figure 5.1 illustrates the first step for the above specification. In this chapter, the notation \succeq stands for $\triangleright \cup \equiv$, that is, if $h \succeq h'$ then either h and h' are the same events or $h \triangleright h'$. Set L_1 tracks the messages that satisfy the causality relation left of the perpendicular line associated with L_1 in the figure, that is $a \in L_1[g]$ if $(a.s \succeq g)$ and a is a *red* message. Similarly, set L_2 tracks the messages that satisfy the causality relation left of the perpendicular line associated with L_2 , that is, $a \in L_2[g]$ implies there exists a message b such that $(a.s \triangleright b.s) \wedge (b.r \succeq g)$, where a is a *red* message and b is a *green* message.

Whenever a process sends a message x it tags its local information along with the message. The new local information of the receiving process is a function of its old local information (L_i s), and the information tagged along with the message x

$(x.L_i.s)$. The value of the set L_i just after the event g is $L_i[g]$. The detection of the predicate is done in stages. For example, in the above predicate the two stages are:

1. Detection of the first part of the predicate. Therefore,

$$a \in L_1[g] \Leftrightarrow ((\text{color}(a) = \text{red}) \wedge (a.s \succeq g)).$$

Thus, L_1 keeps track of all messages that are *red*. Therefore, whenever a process sends or receives a *red* message it adds the message to the set L_1 .

2. Detection of the first part followed by the second part, i.e.,

$$a \in L_2[g] \Leftrightarrow (\exists b : \text{color}(a) = \text{red} \wedge \text{color}(b) = \text{green} : (a.s \triangleright b.s) \wedge (b.r \succeq g)).$$

Note that part of the predicate is the same as the definition of L_1 . Using the identity,

$$(a.s \triangleright b.s) \equiv \exists \text{ an event } f : (a.s \succeq f) \wedge (f \triangleright b.s),$$

we can rewrite L_2 using L_1 as,

$$\begin{aligned} a \in L_2[g] \Leftrightarrow & \exists \text{ an event } f, \text{ and a message } b : \\ & (\text{color}(b) = \text{green}) \wedge (f \triangleright b.s) \wedge (b.r \succeq g) \wedge (a \in L_1[f]). \end{aligned}$$

The condition specifies when a message from any level set is added to a higher level set (e.g., from L_1 to L_2).

If a process executes a message b such that $\text{color}(b) = \text{green}$ then the sending process can update the level sets L_1 and L_2 using its old values and the knowledge that message b satisfies the condition. Similarly, the receiving process can update its level sets based on its level sets and the level sets tagged with the message b .

5.2.2 Safety

The second step guarantees the safety property. The algorithm maintains the safety property by the following invariant:

$$\text{wait}(g) \equiv (a \in L_2[g]) \wedge (\text{proc}(a.r) = \text{proc}(g)) \Rightarrow (a.r \triangleright g).$$

Informally, this states the waiting condition for the event g . That is, the execution of g should wait until all the messages have been received that will belong to the set $L_2[g]$.

Let us consider the above forbidden predicate. The L_i s are

$$a \in L_1[g] \Leftrightarrow \text{color}(a) = \text{red} : (a.s \succeq g),$$

$$a \in L_2[g] \Leftrightarrow \exists f, b : \text{color}(b) = \text{green} : (f \triangleright b.s) \wedge (b.r \triangleright g) \wedge (a \in L_1[f]).$$

Consider the case when the invariant $\text{wait}(g)$ is not maintained. That is,

$$a \in L_2[g] \wedge (\text{proc}(a.r) = \text{proc}(g) \wedge \neg(a.r \triangleright g)).$$

Since the events $a.r$ and g are in the same process, $\neg(a.r \triangleright g) \Rightarrow (g \succeq a.r)$. From $(g \succeq a.r)$ and $a \in L_2[g]$, that is,

$$\exists b : (\text{color}(a) = \text{red}) \wedge (\text{color}(b) = \text{green}) \wedge (a.s \triangleright b.s) \wedge (b.r \succeq g),$$

we have $\exists a, b : (a.s \triangleright b.s) \wedge (b.r \succeq g) \wedge (g \succeq a.r)$. Therefore,

$$\exists a, b \in M : (\text{color}(a) = \text{red}) \wedge (\text{color}(b) = \text{green}) \wedge (a.s \triangleright b.s) \wedge (b.r \triangleright a.r).$$

Thus, $B(a, b)$ is true and the run is illegal.

5.3 Discussion of the General Algorithm

Consider a predicate B with the corresponding graph $G_B(V, E)$. Let the vertex set be $V = \{x_1, x_2, \dots, x_m\}$ and the edge set be $E = \{(x_i, x_{(i \bmod m)+1}) : i = 1, \dots, m\}$. Without loss of generality we will assume x_1 is the β vertex. The conditions satisfied by messages are $\{c_1, c_2, \dots, c_m\}$. The forbidden predicate can be written as:

$$\exists x_1, \dots, x_m \in M : c_1(x_1) \wedge \dots \wedge c_m(x_m) : (x_1.s \triangleright x_2.p) \wedge \dots \wedge (x_m.q \triangleright x_1.r).$$

We will drop the quantifier \exists for ease of use. For example, consider the forbidden predicate from the last section¹:

$$\text{color}(x_1) = \text{red} \wedge \text{color}(x_2) = \text{green} : (x_1.s \triangleright x_2.s) \wedge (x_2.r \triangleright x_1.r),$$

¹Instead of using x, y as free variables we are using x_1, x_2 respectively.

the vertex set $V = \{x_1, x_2\}$ and $E = \{(x_1, x_2), (x_2, x_1)\}$. The conditions satisfied by the messages are $c_1(x) \equiv (\text{color}(x) = \text{red})$ and $c_2(x) \equiv (\text{color}(x) = \text{green})$.

The definition for the level set L_1 for the example predicate is:

$$a \in L_1[g] \Leftrightarrow \text{color}(a) = \text{red} : (a.s \succeq g).$$

For the general predicate we get:

$$a \in L_1[g] \Leftrightarrow c_1(a) : (a.s \succeq g).$$

The definition of the level set L_2 for the example predicate is:

$$a \in L_2[g] \Leftrightarrow \exists f, b : (\text{color}(b) = \text{green}) \wedge (f \triangleright l_2(b) \succeq g) \wedge (a \in L_1[f]).$$

Similarly, for the general predicate we get:

$$a \in L_2[g] \Leftrightarrow \exists f, b : c_2(b) \wedge (f \triangleright l_2(b) \succeq g) \wedge (a \in L_1[f]).$$

This can be generalized to other level sets and we get:

$$a \in L_i[g] \Leftrightarrow \exists f, b : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (a \in L_{i-1}[f]).$$

Informally, $L_i[g]$ is a set of messages where $a \in L_i[g]$ if there exists a set of messages $\{b_2, b_3, \dots, b_i\}$, such that $a.s \triangleright l_1(b_1) \triangleright l_2(b_2) \triangleright \dots \triangleright l_i(b_i) \succeq g$ and $c_1(a) \wedge c_2(b_2) \wedge c_3(b_3) \wedge \dots \wedge c_i(b_i)$. A message a is moved up the level sets from L_1 to L_2 and so forth. Eventually $a \in L_m$ and an event g has to wait for the message a before executing, satisfying the waiting condition

$$\text{wait}(g) \equiv (x \in L_m[g]) \wedge (\text{proc}(x.r) = \text{proc}(g)) \Rightarrow (x.r \triangleright g).$$

The conditions met by the algorithm are as follows:

Waiting Condition (WC) : When is an event delayed ?

$$(x \in L_m[g]) \wedge (\text{proc}(x.r) = \text{proc}(g)) \Rightarrow (x.r \triangleright g).$$

Enabling condition for event g :

$$E1 \quad (x \in L_m[g]) \wedge (\text{proc}(a.r) = \text{proc}(g)) \Rightarrow (x.r \triangleright g).$$

sending message x :

```

S1   Message Tag =  $\{L_1, L_2, \dots, L_m\}$ 
S2   for  $i$  in  $\{m, m-1, \dots, 2\}$  do
S3       if  $(c_i(x))$  then
S4           if  $v_i$  is of type  $(s, s)$  then
S5                $L_i = L_i \cup L_{i-1}$ .
S6   if  $(c_1(x))$  then
S7        $L_1 = L_1 \cup \{x\}$ 

```

Receiving message x :

```

R1    $L_i = L_i \cup x.L_i$ 
R2   for  $i$  in  $\{m, m-1, \dots, 2\}$  do
R3       if  $(c_i(x))$  then
R4           if  $v_i$  is of type  $(s, s)$  then
R5                $L_i = L_i \cup x.L_{i-1}$ .
R6           if  $v_i$  is of type  $(s, r)$  then
R7                $L_i = L_i \cup x.L_{i-1}$ .
R8           if  $v_i$  is of type  $(r, r)$  then
R9                $L_i = L_i \cup L_{i-1} \cup x.L_{i-1}$ .
R10  if  $(c_1(x))$  then
R11   $L_1 = L_1 \cup \{x\}$ 

```

Figure 5.2: Pseudo-code for an algorithm implementing WC, EC, and UC.

Entry Condition (EC) : Which messages are of interest ?

$$c_1(a) \wedge (a.s \succeq h) \Leftrightarrow a \in L_1[h].$$

Update Condition (UC) : When does a message a become a member of level set $L_i, i > 1$?

$$(a \in L_i[g]) \Leftrightarrow \exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (a \in L_{i-1}[f]).$$

The algorithm satisfying the above conditions (WC, EC, and UC) is given in Figure 5.2. Here we give an informal argument to show that the algorithm

satisfies the above three properties. Statement E1 implements WC. Statements S6-S7 for the sending process and R10-R11 for the receiving process implement EC, and statements S2-S5 for the sending process and R1-R9 for the receiving process implement UC.

5.4 Proof of the Correctness of the Algorithm

Lemma 5.1 (Monotonic Property (MP)) UC and EC \Rightarrow MP where MP is

$$x \in L_i[g] \Rightarrow \forall h : g \triangleright h : x \in L_i[h].$$

Proof: Let $a \in L_1[g]$ and $g \triangleright h$.

$$\begin{aligned} a \in L_1[g] &\Rightarrow c_1(a) \wedge (a.s \succeq g) && \text{Using EC} \\ &\Rightarrow c_1(a) \wedge (a.s \triangleright h) && \text{Using } g \triangleright h \\ &\Rightarrow a \in L_1[h] && \text{Using EC} \end{aligned}$$

Let $a \in L_i[g], i > 1$ and $g \triangleright h$.

$$\begin{aligned} a \in L_i[g] &\Rightarrow \exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (a \in L_{i-1}[f]) && \text{Using UC} \\ &\Rightarrow \exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \triangleright h) \wedge (a \in L_{i-1}[f]) && \text{Using } g \triangleright h \\ &\Rightarrow a \in L_i[h] && \text{Using UC} \end{aligned}$$

□

Lemma 5.2 Given a forbidden predicate B, data structures L_i s satisfying UC and EC, and an event g and a message a such that $(a \in L_m[g]) \wedge (g \triangleright a.r)$. Then $\exists a_1, a_2, \dots, a_m : B(a_1, a_2, \dots, a_m)$ is true.

Proof: We use the following properties of UC and EC:

Prop. 1: $x \in L_i[g] \Rightarrow (\exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (x \in L_{i-1}[f]))$, and

Prop. 2: $x \in L_1[g] \Rightarrow (x.s \succeq g) \wedge c_1(x)$.

Let $a_1 \equiv a$, then from the statement of this lemma $a_1 \in L_m[g]$. On expanding $L_m[g]$ using Prop. 1, we have

$$a_1 \in L_m[g] \Rightarrow \exists a_m, f : c_m(a_m) \wedge (f \triangleright l_m(a_m) \succeq g) \wedge (a_1 \in L_{m-1}[f]).$$

On expanding $L_{m-1}[f]$ we get

$$a_1 \in L_m[g] \Rightarrow \exists a_m, f : c_m(a_m) \wedge (f \triangleright l_m(a_m) \succeq g) \wedge (\exists a_{m-1}, h : c_{m-1}(a_{m-1}) \wedge (h \triangleright l_{m-1}(a_{m-1}) \succeq f) \wedge (a_1 \in L_{m-2}[h])).$$

On simplification we get

$$a_1 \in L_m[g] \Rightarrow \exists a_m, a_{m-1}, h : c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge (h \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \succeq g) \wedge (a_1 \in L_{m-2}[h]).$$

After repeatedly expanding L_i s we eventually get

$$a_1 \in L_m[g] \Rightarrow \exists a_m, a_{m-1}, \dots, a_2, h : c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \dots \wedge c_2(a_2) \wedge (h \triangleright l_2(a_2) \triangleright \dots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \succeq g) \wedge (a_1 \in L_1[h]).$$

If $(a_1 \in L_1[h])$ then from Prop. 2, we have $(a_1.s \succeq h) \wedge c_1(a_1)$. Therefore,

$$a_1 \in L_m[g] \Rightarrow \exists a_m, a_{m-1}, \dots, a_2 : c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \dots \wedge c_1(a_1) \wedge (a_1.s \triangleright l_2(a_2) \triangleright \dots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \succeq g)$$

Since it is given that $(a_1 \in L_m[g]) \wedge (g \triangleright a_1.r)$, therefore

$$\exists a_m, a_{m-1}, \dots, a_1 : c_m(a_m) \wedge c_{m-1}(a_{m-1}) \wedge \dots \wedge c_1(a_1) \wedge (a_1.s \triangleright l_2(a_2) \triangleright \dots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \triangleright a_1.r)$$

Therefore $P(a_1, a_2, \dots, a_m)$ is true. \square

Lemma 5.3 Given a forbidden predicate B , data structures L_i s satisfying UC and EC, and messages a_1, a_2, \dots, a_m such that $B(a_1, a_2, \dots, a_m)$ is true. Then $\exists g : \text{wait}(g)$ is false.

Proof: We use the following properties satisfied by EC, UC, and MP:

Prop. 1: $c_1(a) \Rightarrow a \in L_1[a.s]$.

Prop. 2: $\exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (a \in L_{i-1}[f]) \wedge \neg(a.r \triangleright g) \Rightarrow a \in L_j[g]$.

Prop. 3: $a \in L_m[g] \Rightarrow \forall h : (g \triangleright h) \wedge \neg(h \triangleright a.r) : a \in L_m[h]$.

Since $B(a_1, a_2, \dots, a_m)$, we have

$$c_1(a_1) \wedge c_2(a_2) \wedge \dots \wedge c_m(a_m), \text{ and}$$

$$(a_1.s \triangleright l_2(a_2) \triangleright \dots \triangleright l_{m-1}(a_{m-1}) \triangleright l_m(a_m) \triangleright a_1.r).$$

Let $t_i(x)$ represent the top element of $l_i(x)$, that is,

$$t_i(x) = \begin{cases} x.s & \text{if } l_i(x) = x.s \\ x.r & \text{if } l_i(x) = x.s \triangleright x.r \\ x.r & \text{if } l_i(x) = x.r \end{cases}$$

Prop. 1 and $c_1(a_1) \Rightarrow a_1 \in L_1[a_1.s]$.

Since $c_2(a_2) \wedge (a_1.s \triangleright l_2(a_2) \succeq t_2(a_2) \triangleright a_1.r) \wedge (a_1 \in L_1[a_1.s])$, therefore from Prop. 2, we have $a_1 \in L_2[t_2(a_2)]$.

Similarly, $c_2(a_2) \wedge (t_2(a_2) \triangleright l_2(a_2) \succeq t_2(a_2) \triangleright a_1.r) \wedge (a_1 \in L_2[t_2(a_2)])$, therefore from Prop. 2, we have $a_1 \in L_3[t_2(a_2)]$.

By repeated application of Prop. 2, we eventually get

$$a_1 \in L_m[t_m(a_m)].$$

Since $(t_m[a_m] \triangleright a_1.r)$ and $(a_1.s \triangleright t_m(a_m))$ we get

$$\text{proc}(t_m[a_m]) = \text{proc}(a_1.r) \quad \vee$$

$$\exists g : (t_m[a_m] \triangleright g \triangleright a_1.r) \wedge (\text{proc}(g) = \text{proc}(a_1.r)).$$

From $a_1 \in L_m[t_m(a_m)]$ and Prop. 3, we have $a_1 \in L_m[g]$ for all g such that $t_m(a_m) \succeq g$ and $\neg(a.r \triangleright g)$. Therefore,

$$\exists g : (a_1 \in L_m[g]) \wedge (\text{proc}(a_1.r) = \text{proc}(g)) \wedge (g \triangleright a_1.r).$$

Therefore $\text{wait}(g)$ is false. □

Theorem 5.1 (Safety) Let the data structures L_i s satisfy UC and EC. Then there exists a set of messages $\{a_1, a_2, \dots, a_m\}$ such that $B(a_1, \dots, a_m)$ is true if and only if $\exists g : \text{wait}(g)$ is false.

Proof: Follows from Lemma 5.2 and Lemma 5.3. □

Lemma 5.4 If UC and EC are true, then

1. $x \in L_i[g] \Rightarrow x \in L_{i-1}[g]$.
2. $x \in L_i[g] \Rightarrow (x.s \succeq g) \wedge c_1(x)$.

Proof: The first statement follows from MP and UC.

$$\begin{aligned} a \in L_i[g] &\Rightarrow \exists f : (f \triangleright g) \wedge (a \in L_{i-1}[f]) && \text{Using UC} \\ &\Rightarrow a \in L_{i-1}[g] && \text{Using } g \triangleright f \text{ and MP} \end{aligned}$$

The second statement follows from the first statement and EC.

$$a \in L_i[h] \Rightarrow a \in L_{i-1}[h] \Rightarrow \dots \Rightarrow L_1[h] \Rightarrow (x.s \succeq h) \wedge c_1(x).$$

□

Theorem 5.2 (Liveness) Every event is eventually executed.

Proof: An event g is delayed only if $a \in L_n[g]$ and $\text{proc}(a.r) = \text{proc}(g)$ and a has not been received. Since $a \in L_i[g]$ (from Lemma 5.4), $a.s \triangleright g$. Therefore either $a.r$ has been received or the message is in transit. □

5.5 Discussion

In this chapter we presented a general optimal algorithm to implement a class of message ordering specifications. The algorithm is optimal in the sense that it is least restrictive, it delays an event if and only if it will result in safety violation. In this section we present some ideas on implementing space efficient algorithms. In particular:

- The level sets satisfy the monotonic condition. Thus, a message identifier once added to a level set is never purged, even after that information is of no use.
- The general algorithm results in passing information along with the message that is either not necessary or can easily be reduced. For example, in case of causal ordering a $n \times n$ matrix (that is, n^2 message identifiers) is sufficient.

We will study the first issue in the following subsection, Garbage Collection, and the next issue in the next subsection under the heading Induction Argument.

5.5.1 Garbage Collection

In the algorithm there is no way to purge an entry from any of the level sets. In this section we discuss two methods to purge an entry from a level set.

In the algorithm an event g is delayed until it receives all the messages in the set $L_m[g]$. If a process knows that a message has been already received then the level sets L_i s need not keep track of the message in the causal future. Secondly, if a message $x \in L_j[g]$ it is redundant for a level set $L_i[g]$ where $i < j$ to keep track of that information, since the waiting condition of any event depends on the messages in the level set L_m . Thus, taking these ideas into consideration we get the following conditions to be met by the algorithm:

Waiting Condition (WC) : When is an event delayed ?

$$(x \in L_m[g]) \wedge (\text{proc}(x.r) = \text{proc}(g)) \Rightarrow (x.r \triangleright g).$$

Modified Entry Condition (MEC) : Which messages are of interest ?

$$c_i(a) \wedge (a.s \succeq h) \wedge \neg(a.r \succeq h) \iff (a \in L_1[h]).$$

Modified Update Condition (MUC) : When does a message a become a member of level set L_i , $i > 1$?

$$\exists b, f : c_i(b) \wedge (f \triangleright l_i(b) \succeq g) \wedge (a \in L_{i-1}[f]) \wedge \neg(a.r \triangleright g) \iff a \in L_j[g].$$

We can prove similar results using WC, MEC, and MUC as was done for WC, EC, and UC.

Lemma 5.5 If MUC and MEC are true, then

1. $x \in L_i[g] \Rightarrow x \in L_{i-1}[g]$.
2. $x \in L_i[g] \Rightarrow (x.s \succeq g) \wedge c_1(x)$.
3. $a \in L_i[h] \Rightarrow \neg(a.r \triangleright h)$.

Proof Outline: Similar to proof of lemma 5.4. The third part follows from MUC and MEC. □

Theorem 5.3 (Safety) Let the data structures L_i s satisfy MUC and MEC. Then there exists a set of messages $\{a_1, a_2, \dots, a_m\}$ such that $B(a_1, \dots, a_m)$ is true if and only if $\exists g : wait(g)$ is false.

Proof Outline: To prove “if” part show that MUC, MEC, and $\exists g : wait(g)$ is false, implies the two properties stated in the proof of lemma 5.2.

To prove “only if” part show that MUC, MEC and $\exists a_i s : P(a_1, a_2, \dots, a_m)$ is true, implies the three properties stated in the proof of lemma 5.3. □

Theorem 5.4 (Liveness) Every event is eventually executed.

Proof Outline: Similar to earlier liveness proof. □

It is interesting to note that if $x \in L_i[g]$ then $x \in L_{i-1}[g]$ (from lemma 5.5), therefore, it is sufficient to maintain x as an element of the largest indexed set. Therefore, an efficient algorithm maintains the following properties:

Uniqueness Property : $x \in L_i[g] \Rightarrow x \notin L_j[g], \forall j < i.$

Purge Condition : $x.r \triangleright g \Rightarrow x \notin L_i[g], \forall i.$

To maintain the uniqueness property is trivial. We present an algorithm (in Figure 5.3) to maintain the purge condition. This is done by keeping vector clocks, and assigning a number in increasing order to each message sent by a process.

5.5.2 Induction Argument

Many implementations for specifications, like causal ordering, FIFO and some marker type algorithms implicitly use the induction arguments. That is, say message x must be received after $\{y, z\}$ and message y must be received after $\{z\}$, then the desired objective is achieved even if we say message x should be received after $\{y\}$ and message y must be received after $\{z\}$. Thus, in the case of causal ordering, we get a $n \times n$ matrix, where each element is just a message identifier of the previous message that should have been received. This idea can be implemented using the following two steps.

1. Defining an order: for any two messages x, y : x is less than y if

$$(x \in L_m[y.r]) \wedge (\text{proc}(x.r) = \text{proc}(y.r)).$$

2. Keeping only the maximal elements in L_i .

We illustrate the procedure in the following example.

Example 5.1 Consider FIFO in the following run:

Enabling condition for event g :

$$E1 \quad (x \in L_m[g]) \wedge (\text{proc}(a.r) = \text{proc}(g)) \Rightarrow (x.r \triangleright g).$$

sending message x :

```

S1   Message Tag =  $\{L_1, L_2, \dots, L_m, V\}$ 
S2   for  $i$  in  $\{m, m-1, \dots, 2\}$  do
S3     if  $(c_i(x))$  then
S4       if  $v_i$  is of type  $(s, s)$  then
S5          $L_i = L_i \cup L_{i-1}, L_{i-1} = \emptyset$ 
S6   if  $(c_1(x))$  then
S7      $V[\text{proc}(x.s)] ++$ 
S8      $L_1 = L_1 \cup \{(x, V[\text{proc}(x.s)])\}$ 

```

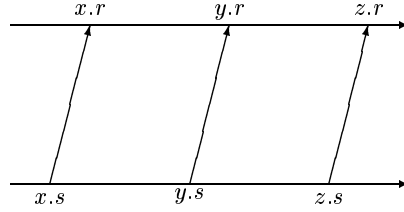
Receiving message x :

```

R1   for  $i$  in  $\{m, m-1, \dots, 1\}$  do
R2     if  $((y, n) \in L_i) \wedge ((y, n) \notin \cup_j x.L_j) \wedge (n \leq x.V[\text{proc}(x.s)])$  then
R3        $L_i = L_i - \{(y, n)\}$ 
R4     if  $((y, n) \in x.L_i) \wedge ((y, n) \notin \cup_j L_j) \wedge (n \leq V[\text{proc}(x.s)])$  then
R5        $x.L_i = x.L_i - \{(y, n)\}$ 
R6     if  $((y, n) \in L_i) \wedge ((y, n) \in \cup_{j>i} x.L_j)$  then
R7        $L_i = L_i - \{(y, n)\}$ 
R8     if  $((y, n) \in x.L_i) \wedge ((y, n) \in \cup_{j>i} L_j)$  then
R9        $x.L_i = x.L_i - \{(y, n)\}$ 
R10  for  $i$  in  $\{m, m-1, \dots, 1\}$  do
R11     $L_i = L_i \cup x.L_i$ 
R12  for  $i$  in  $\{m, m-1, \dots, 2\}$  do
R13    if  $(c_i(x))$  then
R14      if  $v_i$  is of type  $(s, s)$  then
R15         $L_i = L_i \cup x.L_{i-1}, L_{i-1} = L_{i-1} - x.L_{i-1}$ 
R16      if  $v_i$  is of type  $(s, r)$  then
R17         $L_i = L_i \cup x.L_{i-1}, L_{i-1} = L_{i-1} - x.L_{i-1}$ 
R18      if  $v_i$  is of type  $(r, r)$  then
R19         $L_i = L_i \cup L_{i-1} \cup x.L_{i-1}, L_{i-1} = \emptyset$ 
R20  if  $(c_1(x))$  then
R21     $L_i = L_i - \{(x, n)\}$ 
R22     $V[\text{proc}(x.s)] ++$ 

```

Figure 5.3: Pseudo-code for an algorithm implementing WC, MEC, and MUC.



The level set L_m has the following values:

$$L_2[x.r] = \{\} \quad \text{that is, message should be received after } \{\}.$$

$$L_2[y.r] = \{x\} \quad \text{that is, message should be received after } \{x\}.$$

$$L_2[z.r] = \{x, y\} \quad \text{that is, message should be received after } \{x, y\}.$$

Since $x \in L_2[y.r]$ therefore $x < y$, and if we take only the maximal element in L_m we get:

$$L_2[x.r] = \{\} \quad \text{that is, message should be received after } \{\}.$$

$$L_2[y.r] = \{x\} \quad \text{that is, message should be received after } \{x\}.$$

$$L_2[z.r] = \{y\} \quad \text{that is, message should be received after } \{y\}.$$

5.6 Related Work

A fair amount of research has been done for efficient algorithms to implement different message orderings. Birman and Joseph [5], Raynal, Schiper and Toueg [32], and Schiper, Eggli and Sandoz [33], have presented algorithms for the causal ordering of messages. These algorithms tag knowledge of processes about messages sent in the system with the message. Variants of FIFO ordering have been studied under F-channels [2]. The implementation of F-channels provides us with some basic synchronization primitives for sending messages: *two-way-flush* send, *forward-flush* send, *backward-flush* send, and *ordinary* send. Similar flush primitives can be defined for causal ordering. These message orderings can be specified using forbidden predicates. By constructing predicate graphs of these predicates it can be shown that these orderings can be implemented without using any control messages.

Many asynchronous consistent-cut protocols [37] such as global snapshot algorithms [11, 20, 27], check-pointing and rollback recovery [19, 25, 24], and deadlock detection [9] require special messages to find consistent-cuts in a computation. These protocols require some form of inhibition of the special messages in order to guarantee correctness.

5.7 Summary

In this chapter, we extended the concept of forbidden predicates by defining three attributes for each message: color, sending process, and receiving process. All existing message ordering specifications such as FIFO, flush channels, and causal ordering as well as many new message orderings can be concisely specified using forbidden predicates.

We presented a general algorithm to implement the message orderings that can be specified using forbidden predicate and that are implementable without control messages. We further presented techniques to generate efficient protocols for a given specification.

Chapter 6

Implementation

In the previous chapter we presented a general algorithm to implement an optimal protocol for a class of message orderings. The protocol generated is optimal in the sense that it is least restrictive. We considered two techniques – Garbage Collection and Induction Argument – to make the protocol space efficient.

In this chapter we discuss automatic generation of efficient protocols for the class of specifications studied in the previous chapter. The implementation consists of three layers: a distributed simulator modeling a distributed program that communicates using messages, the protocol layer automatically generated given a forbidden predicate, and the lowest layer facilitating the interprocess communication.

6.1 Interface to the Protocol

The protocol layer (being the middle layer) interfaces with the user program and the interprocess communication layer. The interface to the user program uses two functions. They are:

- `void send (const Msg& m)` – a function used by the user to send a message, and
- `Msg& deliver ()` – a function used by the user to receive a message.

```

class Msg {
public:
    virtual int sendProc () = 0;
    virtual int recvProc () = 0;

    virtual int color () { throw };

    virtual void* data () = 0;
    virtual unsigned int length () = 0;
};

```

Figure 6.1: Definition of the class `Msg`.

These functions are written in C++ and the definition of the class `Msg` is given in Figure 6.1. The parameter passed to the `send` function is an instance of a class derived from class `Msg`. Similarly, the return value of the function `deliver` is an instance of a class derived from class `Msg`. The definition of the functions `send ()` and `deliver ()` is given in Figure 6.2.

The protocol interfaces with the communication layer through two functions. They are:

- `SEND (int port, const Data& d)` – transfers the information `d.data()` of length `d.length()` to the destination process given by the first argument, and
- `Data RECV ()` – receives the information sent by another process using `SEND ()`.

The function `RECV ()` is a blocking receive function, we assume that the underlying system provides a reliable message delivery guarantee.

Figure 6.3 shows the interaction of the protocol with the user program and the communication layer.

6.2 Code Generator

We considered the following issues when writing the code generator.

```

Info local;
list<DataPacket> msgQueue;

void send (const Msg& m)
{
    SEND (m, local);           // send the message
    updateOnSend (m, local);   // update local info
}

const Msg& deliver ()
{
    list<DataPacket>::iterator i = msgQueue.begin ();
    for (; i != msgQueue.end (); i++)
    {
        Info& remote = (*i).info ();
        Msg& msg = (*i).msg ();

        if (wait (msg, remote, local) == false)
        {
            static MsgPacket m;

            m = msg;
            updateOnRecv (m, remote, local);

            msgQueue.erase (i);
            return (m);
        }
    }
    msgQueue.push_back (DataPacket (RECV ()));
    return (deliver ());
}

```

Figure 6.2: Send and Deliver functions.

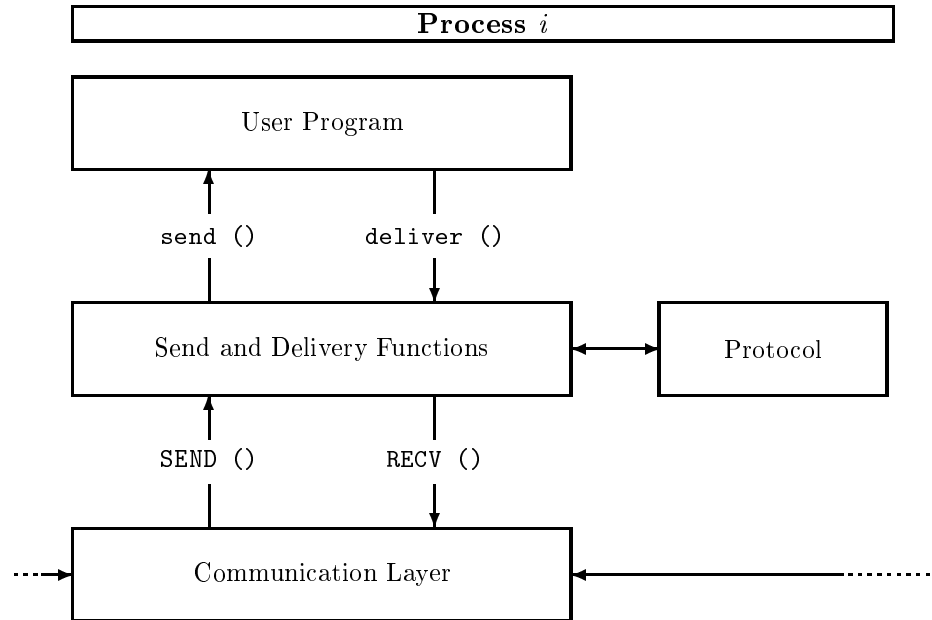


Figure 6.3: Architecture of the implementation.

Induction : We exploit the induction argument (explained in Section 5.5.2) when the conditions imposed on the first variable and the last variable are symmetric with respect to each other.

Large N : The existing algorithms for causal ordering usually tag control information as a $n \times n$ matrix with the messages. If the number of processes is large then the amount of control information can be very large and may not be all necessary. An alternative is to keep sparse matrices thus reducing the size of the control information. The protocol generator when invoked with a `-n` option generates code with sparse matrices.

Use of Counters : Consider the case of the marker message specification given in Figure 6.4. When a process sends a *red* message z after sending 5 *green* messages e.g., $\{a, b, c, d, e\}$, the process tags the *red* message with one of the following:

- Receive the following *green* messages: $\{a, b, c, d, e\}$ before receiving this *red* message.
- Receive 5 *green* of type $[a]$ before receiving this *red* message, where all the five messages fall in the equivalence class $[a]$.

The use of counters, as in the latter case, can result in space efficient protocols.

We have not implemented this optimization in our protocol generator.

Dimensions of Level Sets : In many cases, for example FIFO, the complete local information need not be tagged along with a message. In the case of FIFO, the local information consists of N integers, while only one integer is tagged along with every message. Similarly, in the following predicate:

$$\text{proc}(x.s) = \text{proc}(y.s) : (x.s \triangleright y.s) \wedge (y.r \triangleright \dots),$$

each process tags the level set L_1 with a message, but at the receiving end all the messages are moved up to a higher level. Thus, each process tracks only the messages sent by itself in the level set L_1 .

6.3 Input

Input to the protocol generator is a forbidden predicate as discussed in the previous chapter. Figure 6.4 shows an input file for the forbidden predicate:

$$\exists x, y \in M : \text{color}(x) \neq \text{color}(y) : (x.s \triangleright y.s) \wedge (y.r \triangleright x.r).$$

In an input specification, the process identifiers are given in the **Process** field, and the color identifiers for the messages are given in the **Colors** field. The forbidden predicate is specified by the two fields **Predicate** and **Filter**. The name in the **Specification** field is used to generate the output files containing the protocols. The files for the example in Figure 6.4 are:

- the header file is **Example.h** – represents the internal data structure, and
- the c-file is **Example.cc** – contains the protocol generated.


```

Specification: Example

Processes:      a, b, c, d, e

Variables:     x, y

Colors:        red, green

Filter:

                color (x) != color (y)

Predicate:

                (x.s < y.s) and
                (y.r < x.r)

```

Figure 6.4: An example input file.

```

Predicate ← PredicateClause [ and PredicateClause ]*
Filter    ← [ FilterClause [ and FilterClause ]* ]
PredicateClause ← (event < event)
FilterClause    ← ProcessFilter | ColorFilter
ProcessFilter  ← process (event) op process (event) |
                  process (event) op Processes
event         ← Variables.s | Variables.r
ColorFilter   ← color (Variables) op color (Variables) |
                  color (Variables) op Colors
op           ← == | !=
Variables    ← Identifier given in Variables field
Colors      ← Identifier given in Colors field
Processes   ← Identifier given in Processes field

```

Figure 6.5: Syntax for writing Filter and Predicate.

6.4 Output

The output for the specification given in Figure 6.4 are two files `Example.h` and `Example.cc`. The header file `Example.h` is:

```
#define N 4

enum Color {Red, Green};

class Info
{
    int LevelOne[2][N][N];
    int LevelTwo[2][N][N];

    friend bool wait (const Msg&, const Info&, const Info&);
    friend void updateOnRecv (const Msg&, Info&, Info&);
    friend void updateOnSend (const Msg&, Info&);
    friend void unionInfo (Info& local, const Info& remote);

public:
    Info ();
    Info (const void *data, const int len);

    const void *data () const;
    int length () const;
};

class RecvdMsgQueue {

    static list<int> ids[2][N];

    friend class Info;
public:
    static bool contains (const Element& e);
    static void received (const Msg& m, unsigned int id);
    static void init ();
};

inline int max (int x, int y){ return (x > y) ? x : y; }

bool wait (const Msg&, const Info&, const Info&);
void updateOnRecv (const Msg&, Info&, Info&);
void updateOnSend (const Msg&, Info&);
void unionInfo (Info& local, const Info& remote);
```

The c-file Example.cc is:

```
bool wait (const Msg& m, const Info& remote, const Info& local)
{
    int k;
    RecvdMsgQueue::init ();
    for (int i = 0; i < N; i++)
    {
        for (int color = 0; color < 2; color++)
        {
            int id = remote.LevelTwo[color][i][procId ()];
            Element e (i, procId (), id, color);
            if (RecvdMsgQueue::contains (e) == false)
            {
                return (true);
            }
        }
    }

    switch (m.color ())
    {
        case Red :
            for (k = 0; k < 2; k++)
            {
                if (k == 0) continue;
                for (int i = 0; i < N; i++)
                {
                    int id = remote.LevelOne[k][i][procId ()];
                    Element e (m.sendProc (), procId (), id, k);
                    if (RecvdMsgQueue::contains (e) == false)
                    {
                        return (true);
                    }
                }
            }
            break;
        case Green :
            for (k = 0; k < 2; k++)
            {
                if (k == 1) continue;
                for (int i = 0; i < N; i++)
                {
                    int id = remote.LevelOne[k][i][procId ()];
                    Element e (m.sendProc (), procId (), id, k);
                    if (RecvdMsgQueue::contains (e) == false)
                    {
                        return (true);
                    }
                }
            }
            break;
    }
}
```

```

    return (false);
}
void updateOnSend (const Msg& m, Info& local)
{
    RecvdMsgQueue::init ();

    /******
    /* Entry Condition(x)
    /******
    switch (m.color ())
    {
        case Red :
            local.LevelOne[Red][m.sendProc ()][m.recvProc ()]++;
            break;
        case Green :
            local.LevelOne[Green][m.sendProc ()][m.recvProc ()]++;
            break;
    }
}

void updateOnRecv (const Msg& m, Info& remote, Info& local)
{
    RecvdMsgQueue::init ();
    int k;

    /******
    /* Entry Condition(x)
    /******
    int id = 0;
    switch (m.color ())
    {
        case Red :
            id = ++(remote.LevelOne[Red][m.sendProc()][m.recvProc ()]);
            break;
        case Green :
            id = ++(remote.LevelOne[Green][m.sendProc()][m.recvProc ()]);
            break;
    }
    RecvdMsgQueue::received (m, id);

    unionInfo (local, remote);

    switch (m.color ())
    {
        case Red :
            for (k = 0; k < 2; k++)
            {
                if (k == 0) continue;
                for (int i = 0; i < N; i++)
                {
                    for (int j = 0; j < N; j++)

```

```

        {
            local.LevelTwo[k][i][j] =
                max (local.LevelTwo[k][i][j],
                    remote.LevelOne[k][i][j]);
        }
    }
}
break;
case Green :
    for (k = 0; k < 2; k++)
    {
        if (k == 1) continue;
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; j < N; j++)
            {
                local.LevelTwo[k][i][j] =
                    max (local.LevelTwo[k][i][j],
                        remote.LevelOne[k][i][j]);
            }
        }
    }
break;
}
}
void unionInfo (Info& local, const Info& remote)
{
    int k;
    for (k = 0; k < 2; k++)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; i < N; i++)
            {
                local.LevelOne[k][i][j] =
                    max (local.LevelOne[k][i][j],
                        remote.LevelOne[k][i][j]);
            }
        }
    }
    for (k = 0; k < 2; k++)
    {
        for (int i = 0; i < N; i++)
        {
            for (int j = 0; i < N; i++)
            {
                local.LevelTwo[k][i][j] =
                    max (local.LevelTwo[k][i][j],
                        remote.LevelTwo[k][i][j]);
            }
        }
    }
}
}

```

Acknowledgments

I would like to thank Mom-Ping Ng and Roger Mitchell for their help in implementing the distributed simulator modeling a distributed program and the communication layer facilitating in the interprocess communication.

Chapter 7

Conclusion and Future Work

7.1 Summary and Discussion

In this dissertation, we presented a new characterization of message ordering specifications. A message ordering specification is characterized as the set of acceptable runs, that is, a subset of \mathbb{X} where \mathbb{X} is the set of all runs. In this broad setting, where each message ordering is a subset of \mathbb{X} , we first determine whether a given specification can be implemented using an inhibition based protocol. We show that a message ordering specification can be implemented if and only if it includes all logically synchronous runs. Further, if it can be implemented then we determine the type of protocol necessary and sufficient to implement it, where the protocols are classified into three types, (1) **general**: those that can tag information and have control messages, (2) **tagged**: those that can tag information, and (3) **tagless**: those that do nothing. For example, we show that a message specification can be implemented by tagging user messages with some additional information if and only if it includes all causally ordered runs. It is an easy consequence of the results of this work that no additional tagging of information can restrict the message ordering further.

Formally, we define three subsets of \mathbb{X} , namely, \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} . We

show that given a specification $\mathbb{Y} \subseteq \mathbb{X}$, it is implementable (there exists a protocol with control messages) if and only if $\mathbb{X}_{sync} \subseteq \mathbb{Y}$. Similarly, there is a protocol without control messages if and only if $\mathbb{X}_{co} \subseteq \mathbb{Y}$. The “do nothing” protocol is sufficient to implement if and only if $\mathbb{X}_{async} \subseteq \mathbb{Y}$. Thus, given a specification (that is the set of acceptable runs) the type of protocol necessary and sufficient can be easily checked by testing the containment of the three limit sets \mathbb{X}_{async} , \mathbb{X}_{co} , and \mathbb{X}_{sync} .

Since \mathbb{X} is an infinite set, we also need a finite representation for its subsets that specify message ordering. We present a method called *forbidden predicates* that can be used to describe a large class of message ordering specifications. All existing message ordering guarantees such as FIFO, flush channels, causal ordering, and logically synchronous ordering as well as others can be concisely specified using forbidden predicates. Given a message ordering specification using forbidden predicates, we present an algorithm that determines the type of protocol necessary to implement that specification.

Lastly, we presented a general algorithm for a class of message orderings that can be implemented without control messages.

7.2 Future Work

In this chapter we present some generalizations to message ordering specifications. In the usual model of a distributed run $(\mathbb{H}, \triangleright)$, a message $x \in \text{Msg}(\mathbb{H})$ is a pair of local events, that is, $\{x.s, x.r\}$. These events are causally related, $x.s \triangleright x.r$ irrespective of the other events in the run. The order relation \triangleright among the events in \mathbb{H} is the transitive closure of the local ordering along a process and the relation between the send event and the corresponding receive event. Thus, we can view a message x as a global event with two local events $x.s$ and $x.r$ along with an order imposed on them. In general, a global event can be any set of local events, for example a broadcast x can be viewed as a global event, that is, a set of local events $\{x.s, x.r_0, x.r_1, \dots, x.r_n\}$ and the relation \triangleright where, for all $i = 0, 1, \dots, n$, $x.s \triangleright x.r_i$

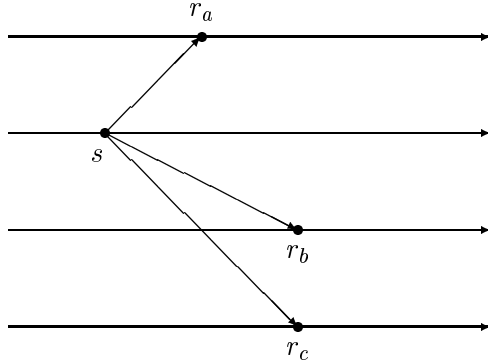


Figure 7.1: A multicast message $\{s, r_a, r_b, r_c\}$.

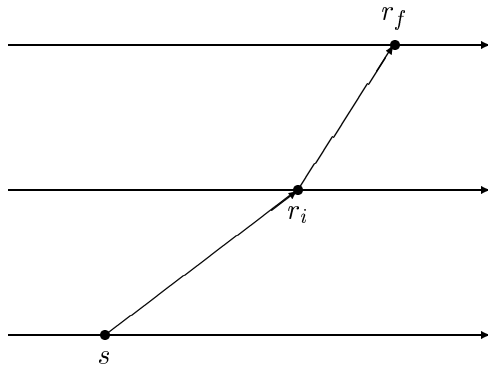


Figure 7.2: A collated message $\{s, r_i, r_f\}$.

holds. In general, we can view a distributed run as an execution of global events, where each global event is isomorphic to an element in S (a set of partial orders). For example, for a system with only two kinds of messages: point-to-point and broadcast, the set is

$$S = \{\{s \triangleright r\}, \{s \triangleright r_i : i = 1, 2, \dots, n\}\}.$$

Thus, in general a distributed system is a 4-tuple (Z, S, M, \mathcal{X}) , where Z is a set of processor identifiers, S the types of global events, M is set of global events each isomorphic to an element in S , and \mathcal{X} is the set of all runs over Z and M .

In the rest of this section, we present three applications of global events, illustrating some of the solved and open issues. First, we consider the case when the

system consists of point-to-point messages and multicast messages (see Figure 7.1). The work done on point-to-point messages can be easily extended to provide us with the answers. Second, we consider the case where the system consists (in addition to point-to-point messages) of messages where a message has an intermediate event that is nothing but a receive and send immediately following. We call such a message a collated message. Figure 7.2 shows one such message. The work presented in this dissertation does not address the basic issues in this framework. Third, we represent the implementation of protocols with control messages using global events.

In the rest of the section, a run $(\mathbb{H}, \rightarrow)$ is defined as an n -valued local state and the happened-before relation on the set of events satisfying the following conditions:

1. Two events occur in the same process.
2. The two events are part of the same global event, and one happens before the other. For example, $s \triangleright r_i$ (or $s \rightarrow r_i$) and $s \triangleright r_f$ (or $s \rightarrow r_f$) in Figure 7.2.
3. If there exists a third event, where the first event happened-before the third event, and the third happened-before the second event then the first event happened-before the second event.

Similarly, we extend \mathcal{X} as the set of all runs over Z and M and the projection of \mathcal{H} is $(\mathbb{H}, \triangleright)$.

Multicast Messages

In the case of multicast messages, we get the same results as in the case of point-to-point messages. We define three limit sets similar to the ones in Section 2.5. The three subsets of \mathbb{X} are:

Asynchronous ordering (ASYNC): This is the same as the ground set \mathbb{X} . Therefore, it includes all possible runs. There exists a `tagless` algorithm (i.e., enable all pending events) that guarantees safety and liveness for this specification.

Causal Ordering (CO): Causal ordering can be stated as $b_1.s \triangleright b_2.s \Rightarrow \neg(b_2.r_i \triangleright b_1.r_j)$ for all i, j . There exists a **tagged** algorithm, CBCAST [5] that implements the specification.

Logically Synchronous (SYNC): A run is logically synchronous if its time diagram can be drawn such that all message arrows are vertical. Formally, a run $(\mathbb{H}, \triangleright)$ is logically synchronous, that is $(\mathbb{H}, \triangleright) \in \mathbb{X}_{sync}$, if there exists a function $T : Msg(\mathbb{H}) \rightarrow \{1, 2, 3, \dots\}$, such that for any two events $h, g \in \mathbb{H}$, if $h \triangleright g$ and $Msg(h) \neq Msg(g)$ then $T(Msg(h)) < T(Msg(g))$. A protocol very similar to ABCAST [5] can implement the specification.

Using similar arguments as in Chapter 2, we get to the same theorem,

Theorem 7.1 Let \mathbb{Y} be a specification. Then

1. A **general** protocol can guarantee safety and liveness iff $\mathbb{X}_{sync} \subseteq \mathbb{Y}$.
2. A **tagged** protocol can guarantee safety and liveness iff $\mathbb{X}_{co} \subseteq \mathbb{Y}$.
3. A **tagless** protocol can guarantee safety and liveness iff $\mathbb{X}_{async} \subseteq \mathbb{Y}$.

Thus, given a specification the type of protocol necessary and sufficient can be easily checked by the containment of the three sets \mathbb{X}_{async} , \mathbb{X}_{co} and \mathbb{X}_{sync} .

Given a message ordering specification using forbidden predicates, we present an algorithm that determines the type of protocol necessary to implement that specification. The algorithm converts the forbidden predicate into a predicate graph. It is shown that the specification can be implemented if and only if there is a cycle in this graph. Further, to determine the nature of the protocol required for the specification, it is sufficient to examine vertices of the graph. We define the notion of β vertices. If the cycle has two or more β vertices with respect to that cycle, then control messages are necessary. If the cycle has one β vertex, then tagging user messages is sufficient. If the cycle has no β vertex, then no action from the protocol is required. Thus, given any message ordering specification using forbidden predicates, the nature of the protocol necessary for implementing it can easily be determined. The above results can be summarized using the following two theorems:

Theorem 7.2 (Sufficient Conditions) Let \mathbb{X}_B be a specification with B as the corresponding forbidden predicate. Let the predicate graph be $G_B(V, E)$ with a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$.

1. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, then $\mathbb{X}_{async} \subseteq \mathbb{X}_B$.
2. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 1, then $\mathbb{X}_{co} \subseteq \mathbb{X}_B$.
3. If there exists a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order $k (> 1)$, then $\mathbb{X}_{sync} \subseteq \mathbb{X}_B$.

Theorem 7.3 (Necessary Conditions) Let \mathbb{X}_B be a specification with B as the corresponding forbidden predicate. Let the predicate graph be $G_B(V, E)$ with a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$.

1. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, 1 or n , then $\mathbb{X}_{sync} \not\subseteq \mathbb{X}_B$.
2. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0 or 1, then $\mathbb{X}_{co} \not\subseteq \mathbb{X}_B$.
3. If there does not exist a cycle $G_c(V^c, E^c) \subseteq G_B(V, E)$ of order 0, then $\mathbb{X}_{async} \not\subseteq \mathbb{X}_B$.

The reason for the similar results between point-to-point and multicast messages is the consequence of the longest chain in a global event being one. In the next example, the results do not carry over to collated messages since the longest chain in a global event is two.

Collated Messages

In the case of collated messages, we cannot extend the results presented in this work. Although the question of the existence of a protocol can be answered, given a specification \mathbb{Y} the existence of a protocol that only tags information to user

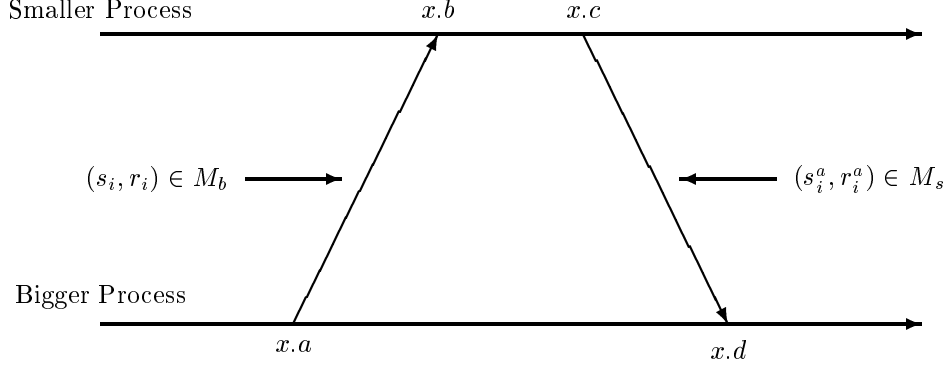


Figure 7.3: Global event $\{x.a, x.b, x.c, x.d\}$ to implement \mathbb{X}_{sync} .

messages remains to be answered. Using arguments similar to the ones in the proofs of Lemma 2.2 and Theorem 2.2, we get

Theorem 7.4 Let \mathbb{Y} be a specification. Then a **general** protocol can guarantee safety and liveness iff $\mathbb{X}_{sync} \subseteq \mathbb{Y}$.

Protocols with control messages

We have seen that there are specifications for example \mathbb{X}_{sync} that require control messages. The protocols implementing these specifications use a sequence of protocol messages to send one user message. Another way to view the sequence of protocol message is as a global event. For example, to implement \mathbb{X}_{sync} we had two messages: the initiation message and the acknowledgment message as shown in Figure 3.4. The protocol messages (s_i, r_i) and (s_i^a, r_i^a) can be viewed as a part of a global event shown in Figure 7.3. Then the conditions (SC, AC, and PR) satisfied by the global events can be stated as

$$(\text{proc}(x.a) = \text{proc}(y.a)) \wedge PR(x) \wedge PR(y) : (x.a \triangleright y.a) \wedge (y.a \triangleright x.d), \text{ and}$$

$$(\text{proc}(x.a) = \text{proc}(y.b)) \wedge PR(x) \wedge PR(y) : (x.a \triangleright y.b) \wedge (y.b \triangleright x.b),$$

where $PR(x) \equiv \text{proc}(x.a) > \text{proc}(x.b) \wedge \text{proc}(x.c) < \text{proc}(x.d)$.

Thus, a protocol with control messages maps a specification B in a system (Z, S, M, \mathcal{X}) to a specification B' in a system $(Z, S', M', \mathcal{X}')$.

Bibliography

- [1] *Reference Manual for the Ada Programming Language*, 1982.
- [2] M. Ahuja. An implementation of F-channels. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):658–667, June 1993.
- [3] R. Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms. *IEEE Transactions on Software Engineering*, 15(9):1053–1065, September 1989.
- [4] R. Bagrodia. Synchronization of asynchronous processes in CSP. *ACM Transactions on Programming Language Systems*, 11(4):585–597, October 1989.
- [5] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, January 1987.
- [6] K. P. Birman and R. V. Renesse, editors. *Reliable Distributed Computing with Isis Toolkit*. IEEE Computer Society Press, 1994.
- [7] A. D. Birrel and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February. 1984.
- [8] L. Bougé and N. Francez. A compositional approach to superimposition. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 240–249. ACM, 1988.
- [9] G. Bracha and S. Toeg. Distributed deadlock detection. *Distributed Computing*, 2(3):127–138, January 1987.

- [10] G. Buckley and A. Silbershatz. An effective implementation of the generalized input-output construct of CSP. *ACM Transactions on Programming Language Systems*, 2(2):223–235, April 1980.
- [11] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February. 1985.
- [12] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [13] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.
- [14] B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.
- [15] M. Choy and S. Ambuj K. Efficient implementation of synchronous communication over asynchronous networks. *Journal of Parallel and Distributed Computing*, 26:166–180, July 1995.
- [16] T. Connolly, P. Amer, and P. Conrad. An extension to TCP: Partial Order Service. Technical Report RFC 1693, Network Working Group, November 1994.
- [17] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. Technical Report RJ 4540 (48668), IBM, October 1984.
- [18] C. Critchlow. On inhibition and atomicity in asynchronous consistent-cut protocols. Technical Report TR 89-1069, Department of Computer Science, Cornell University, December 1989.
- [19] O. P. Damani and V. K. Garg. How to recover efficiently and asynchronously when optimism fails. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 108–115. IEEE, 1996.

- [20] E. W. Dijkstra. The distributed snapshot of K.M Chandy and L. Lamport. In M. Broy, editor, *Control Flow and Data Flow: Concepts of Distributed Programming*. Springer-Verlag, 1985.
- [21] A. Gahlot, M. Ahuja, and T. Carlson. Global flush communication primitive for interprocess communication. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 111–120. ACM, 1994.
- [22] K. J. Goldman. Highly concurrent logically synchronous multicast. Technical Report MIT/LCS/TM-401, M.I.T. Laboratory for Computer Science, July 1989.
- [23] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
- [24] D. B. Johnson and W. Zwaenepool. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181. ACM, 1988.
- [25] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, 13(1):23–31, January 1987.
- [26] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):95–114, July 1978.
- [27] F. Mattern. Efficient distributed snapshots and global virtual time algorithms for non-FIFO systems. Draft Version, March 1990.
- [28] V. V. Murty and V. K. Garg. Limits of protocols based on inhibition to implement message ordering specifications. Submitted to *Distributed Computing*.

- [29] V. V. Murty and V. K. Garg. An algorithm to guarantee synchronous ordering of messages. In *Proceedings of Second International Symposium on Autonomous Decentralized Systems*, pages 208–214. IEEE Computer Society Press, 1995.
- [30] V. V. Murty and V. K. Garg. Characterization of message ordering specifications and protocols. To appear in the Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97), May 1997.
- [31] V. V. Murty and V. K. Garg. Message ordering based on colorful forbidden predicates. Technical Report TR-PDS-1997-005, Parallel and Distributed Systems Laboratory, The University of Texas at Austin, April 1997.
- [32] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, July 1991.
- [33] A. Schiper, J. Egli, and A. Sandoz. A new algorithm to implement causal ordering. In *Proceedings of the Third International Workshop on Distributed Algorithms*, pages 219–232. Springer-Verlay, 1989.
- [34] F. Schmuck. Efficient broadcast primitives in asynchronous distributed systems. In K. P. Birman and R. V. Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 263–283. IEEE Computer Society Press, 1993.
- [35] A. P. Sistla. Distributed algorithms for ensuring fair interprocess communication. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 266–277. ACM, 1984.
- [36] T. Soneoka and T. Ibaraki. Logically instantaneous message passing in asynchronous distributed systems. *IEEE Transactions on Computers*, 43(5):513–527, May 1994.
- [37] K. Taylor. The role of inhibition in asynchronous consistent-cut protocols. In J.-C. Bermond and M. Raynal, editors, *Proc. of the 3rd International Workshop on Distributed Algorithms*, pages 280–291. Springer-Verlag, 1989.

- [38] G. R. Wright and W. R. Stevens. *TCP/IP Illustrated, Volume 2*. Addison-Wesley, 1995.

Vita

Venkataesh Murty was born on the 10th day of December 1969 in Hyderabad, India. After completing high school at Hartmann High School, Bareilly, he entered the Indian Institute of Technology, Madras. He started his undergraduate program with the hopes of graduating with a degree in Electrical Engineering. After a year disillusioned with electrical engineering, he dabbled with industrial and systems engineering. Eventually, he graduated in four years with a B.Tech. in Mechanical Engineering. His interests were now focused on control theory, neural networks, and applications of neural networks in control systems. He entered University of Texas at Austin in the Mechanical Engineering Department and graduated with an M.S. in the Spring of 1993. His main focus there was the use of neural networks in the area of system identification. In addition, while searching for a field of interest for his further studies he explored control theory, mathematics, and computer science. Finally, he entered the Electrical and Computer Engineering Department in the Fall of 1993 and earned his Ph.D. in August 1997.

This dissertation was typeset with L^AT_EX 2_ε¹ by the author.

¹L^AT_EX 2_ε is an extension of L^AT_EX. L^AT_EX is a collection of macros for T_EX. T_EX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.