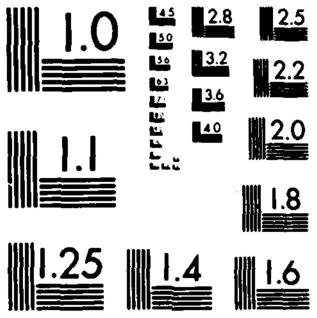
AD-A128 629 EVENT-BASED SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE B CHEN 1982 AFOSR-TR-83-0388 1/3

UNCLASSIFIED F49620-80-C-0001 F/G 9/2 NL


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

3

EVENT-BASED SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS

by

Bo-Shoe Chen

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

Dissertation Submitted to the Faculty of the Graduate School of the University of Maryland in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy 1982

(This work was supported in part by the Air Force Office of Scientific Research Contract [redacted]-F49620-80-C-0001 to the University of Maryland.)

ITEM #20, CONTINUED: view) of distributed systems. The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, including communication systems, transaction-based systems and process control systems are demonstrated.

Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events: the "precedes" relation, representing time order; and the "enables" relation, representing causality. No assumption about the existence of a global clock is made in the specifications.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and structure specification of a system. Important properties of concurrent systems such as "mutual exclusion", "concurrency", "process coordination", and other "safety" and "liveness" properties can be specified and verified.

Moreover, since the specification technique defines the orthogonal properties of a system separately, each of them can be verified independently. Thus, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 83 - 0388	2. GOVT ACCESSION NO. A128 629	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) EVENT-BASED SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bo-Shoe Chen		8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F; 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Mathematical & Information Sciences Directorate Air Force Office of Scientific Research Bolling AFB DC 20332		12. REPORT DATE 1982
		13. NUMBER OF PAGES 180
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)
Apprcved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Computations of distributed systems are extremely difficult to specify and verify using traditional techniques because the systems are inherently concurrent, asynchronous and nondeterministic. Furthermore, computing nodes in a distributed system may be highly independent, and the entire system may lack an accurate global clock.

In this thesis, the author develops an event-based model to specify formally the behavior (the external view) and the structure (the internal (CONTINUED)

DD FORM 1 JAN 73 1473

83 05 23 [illegible]9

ABSTRACT

Title of Dissertation: Event-Based Specification and Verification of Distributed Systems

Bo-Shoe Chen, Doctor of Philosophy, 1982

Dissertation Directed by: Dr. Raymond T. Yeh, Professor, Department of Computer Science

Computations of distributed systems are extremely difficult to specify and verify using traditional techniques because the systems are inherently concurrent, asynchronous and nondeterministic. Furthermore, computing nodes in a distributed system may be highly independent, and the entire system may lack an accurate global clock.

In this thesis, ~~we~~ the author develops an event-based model to specify formally the behavior (the external view) and the structure (the internal view) of distributed systems. The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, including communication systems, transaction-based systems and process control systems are demonstrated.

Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events: the "precedes" relation, representing time order; and the "enables" relation,

representing causality. No assumption about the existence of a global clock is made in the specifications.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and structure specification of a system. Important properties of concurrent systems such as "mutual exclusion", "concurrency", "process coordination", and other "safety" and "liveness" properties can be specified and verified.

Moreover, since the specification technique defines the orthogonal properties of a system separately, each of them can then be verified independently. Thus, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

ACKNOWLEDGEMENT

I wish to express my sincere thanks and appreciation to my advisor, Professor Raymond Yeh, for his guidance, encouragement, and support during my graduate studies. His thirst for simplicity and his wide-ranging interests gave excellent guidance for shaping my half-baked ideas.

I am indebted to my dissertation committee members, Professors Bing Yao, Harlan Mills, John Gannon, Satish Tripathi, and Virgil Gligor, from whom I gained different perspectives on many topics. Dr. Gannon, in particular, made many valuable suggestions for improving the final document's readability.

With pleasure I acknowledge the advice and encouragement I received from Professor Tung Chi-Sung during his visit to UMCP in the summer of 1981. Thanks also go to my colleagues and readers, Bob Arnald, Gary Luckenbaugh, Joy Reed, and Larry Morell, for their critiques.

My special thanks go to my father and mother, whose love from Taiwan has constantly supported me spiritually. Finally, my warmest thanks and gratitude go to my wife, Jane, whose love, tenderness and inspiration has pushed me through my work.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Distributed Systems

Computer systems should reflect the structure and needs of the problems to which they are applied. For many applications, a distributed computer system represents a natural realization. Examples are flight reservation, banking, and ballistic missile defense systems. For both technical and economic reasons, it is likely that for many existing applications, distributed computer systems will replace conventional computer systems built around a large central processor, and that new applications will emerge based on distributed information processing.

The richness of variation on the general theme makes it impossible to define distributed processing rigorously. However, one can characterize the kinds of systems which interest us and demonstrate some general attributes.

A fundamental characteristic of a distributed system is that there is more than one processing unit called a node, in the system. One form of this involves a complex of two or more complete computer systems. Each has its own processor, clock, memory and secondary storage devices. In addition, each processing unit may have a local complement of printers, tapes and other peripheral devices.

A second characteristic is that the system is formed by interconnecting these nodes by a communication network so that information may flow between them. The communication network may be a long-haul network such a[illegible] [illegible]he ARPANET [ROB70], a local network [CLA78], or a sui[illegible]le combination of these two types. The communication [illegible]y may be highly variable and unpredictable. Each node has access to its own memory only; that is, inter-node communication is possible only by exchanging messages, not by sharing memory.

We infer from the discussion above that distributed processing is inherently concurrent, asynchronous, and nondeterministic. Furthermore, the computing nodes in a distributed system may be highly independent of each other, and the entire system may lack an accurate global clock [LAM78b]. The basic problem in distributed systems is to provide coherence (i.e., synchronization and coordination) in communication between the nodes while allowing them to retain their autonomy.

1.2. Advantages of Distributed Systems

The trend toward distributed systems has been supported mainly by the rapidly falling cost of computing hardware and the increasing power and flexibility of mini and micro computers. The steadily decreasing entry cost of acquiring and operating a free-standing, complete computer system encourages lower-level units within a large organization to acquire their own computers dedicated to their own applications. The computers operate somewhat independently and autonomously from one another, while being at least loosely coupled into a cooperating confederacy that serves as the information system of the organization.

The technical advantages distributed systems offer over centralized ones include the following:

(1) Availability. Availability of information can be increased by replicating it at several nodes. This arrangement not only increases the access bandwidth to the information but in case of a failure of one of the nodes or communication links, the information remains accessible.

(2) Protection. This advantage arises from the actual physical separation of independent or loosely coupled computations and information that belong to different users. The physical boundaries of an individual node prevent propagation of errors originating in a particular node to the rest of the system and protect

information stored at individual nodes from unauthorized access or modification by other nodes.

(3) Expandability. As more users join the system or new services are added, it is not necessary to make any physical replacements in a distributed system. Rather, when one or more new nodes need to be added to the system, if the system is designed properly, it may be possible to accomplish this without interrupting the service of the existing system. Thus, distributed systems offer a potential for a more gradual and smoother growth than systems with a large central processor.

Thus, there are many sound reasons why applications should be implemented as distributed systems. However, though it has been successfully demonstrated that it is not very difficult to interconnect remote computers at the electric and bit level, the effective utilization of such a network at a higher application level is still missing.

This dissertation is aimed at providing tools for the development of application software for distributed systems. In particular, the goal of this dissertation is to develop a behavior specification language to support well-structured design verification of distributed systems.

1.3. The Software Specification and Verification

Of serious concern in software construction are techniques that permit us to recognize whether a given program is correct. Although we are beginning to realize that correctness is not the only desirable property of software, it is surely the most fundamental: if a program is not correct, then its other properties (e.g., efficiency, fault tolerance) have no meaning since we cannot depend on them.

Programs that implement distributed systems can exhibit extremely complicated behavior for they must cope with concurrent, asynchronous, non-deterministic computations, and the possibility of failures in nodes or in communication networks. In such a complicated environment, informal techniques such as testing, debugging or program walk-through for establishing the correctness of programs are inadequate. Not only is the investigation of the program properties incomplete and the steps in the reasoning place too much dependence on human ingenuity or intuition, but also a distributed computation is generally non-reproducible, i.e., running the same distributed program on the same data at different times does not always produce the same results.

Thus, we turn to the techniques for verifying the correctness of distributed programs. However, before a formal verification can be done, a formal specification that describes what the program is supposed to do must be provided. The correctness of an implementation is then demonstrated by showing its equivalence to the specification by formal, analytic means.

Formal specifications are of interest even if they are not followed by a formal proof. The importance of software specification and its potential impact on the reliability of software systems has long been recognized and discussed [BOE74, LIS77, YEH80]. Formal specifications make code "public" by serving as a communication medium between different groups of people (users, experts, analysts and designers) [YEH80], and permit consistency and completeness to be judged in a well-defined manner. The result of a successful specification methodology coupled with an advanced design and programming methodology should result in a reduction of the total cost of software development and operation, and the fast release of an operational system [RAM79]. In particular, formal specification together with a hierarchical construction methodology could lead to programs that are correct by construction [LIS77].

1.4. Criteria for the Specification of Distributed Systems

An approach to specification must satisfy a number of requirements if it is to be useful. [BAL79] gives an excellent discussion on the general requirements for specification languages. We discuss in this section the criteria particular for the specification of distributed systems.

It should be possible using the specification method to construct specifications which describe only the interesting properties of a system and nothing more. The properties of interest must be described precisely, unambiguously and in a way which adds as little extraneous information as possible. This criterion is called the "minimality" requirement. One reason for this criterion is that there are usually so many possible designs that it is better to leave as much freedom as possible to the designers. In some applications, the real-time constraints are so strong that early (extraneous, unnecessary) decisions dictated by a specification may make all possible designs infeasible. Another reason for this "minimality" criterion is the desire to minimize correctness proofs by reducing the number of properties to be proved.

It is desirable that a minimal change in a concept result in a similar small change in its specification. This criterion is called the "modifiability" [YEH80] or "extensibility" [LIS77] of specification techniques. An approach to modifiability is to specify independent properties separately and come up with "orthogonal" specifications.

Associated with each specification technique, there is a <u>representational bias</u> [LIS77]: the extent to which the specifications suggest a representation or implementation for the abstractions being defined. The representational bias of a technique determines, in large measure, its range of applicability. Techniques having a representational bias will be limited primarily to those abstractions which are naturally expressed in the representation; within this range, however, specifications will be relatively easy to construct and comprehend. Concepts outside of the technique's range of applicability can only be defined with difficulty, if at all. Petri Nets, for example, have a control-oriented representational bias. It is quite difficult to specify data-related properties with them.

A distributed system, as discussed in Section 1.1., is inherently concurrent, asynchronous, and nondeterministic. Our last criterion for a specification technique for distributed systems is that it should be able to express accurately these key characteristics of distributed systems. Traditional specification techniques for abstract data types or sequential programs are not applicable according to this criterion.

1.5. Our Approach

According to the criteria in Section 1.4., we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal view) of distributed systems. The specification technique has a rather wide range of applications: examples from different classes of distributed systems, including communication systems, transaction-based systems and process control systems are demonstrated.

Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events; the "precedes" relation, representing time order; and the "enables" relation, representing causality. No assumption about the existence of a global clock is made in the specifications. The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and structure specification of a system.

1.6. An Overview of the Thesis

Chapter 2 describes the event model and the behavior specification language, called the "Event-Based Specification Language (EBS)", based on the event model. The behavior of several distributed system examples is then specified in EBS to show its expressive power.

Chapter 3 describes a distributed system from its internal view. Such a description is called a structure specification. Since we use the same notations (i.e., first order logic) in both behavior and structure specifications, the verification of a structure specification with respect to a behavior specification is carried out as proofs of theorems. The structure specifications of a data-transfer protocol [STE76] and a distributed prime number generator [MIS81], together with their verifications, are demonstrated.

In Chapter 4, the event model is extended to handle transaction-oriented system specifications. By a transaction we mean that, when it is executed alone, the sequence of events in it transforms the system from a consistent state into a new consistent state; that is, transactions preserve consistency. A transaction, by this definition, is also a basic unit for crash-recovery. We develop a language called Transaction-Based Specification Language

(TBS), to specify such transaction-oriented systems.

While a transaction represents a <u>sequence</u> of events caused by an enabling event, <u>event coordination</u> represents the cooperation of two (independent) events to enable a third event. We extend the relations in the event model to specify this kind of event coordination, and call the specification language, Coordination-Based Language (CBS). Properties of CBS, together with examples are given in Chapter 5.

Finally, we summerize the advantages of using EBS as a specification language and highlight several further research topics in Chapter 6.

In Appendix A we discuss the formal semantics of EBS. In particular, the semantics of EBS are given in two different ways: using a centralized processor and using multiprocessors.

Comparisons of our specification technique to other approaches are discussed in each chapter. We compare EBS with SPECIAL [ROB77], AFFIRM [TOM80] (for communication systems), and RSL [ALF77] (for real-time systems) in Chapter 2; TBS with Monitors [HOA74], ADA's Rendezvous [ICH79] and Path Expressions [CAM80] in Chapter 4; CBS with Petri Nets [PET77] in Chapter 5; our whole technique with Temporal Logic [LAM80, OWI80], Actor Models [HEW77, BAK78] and the Trace approach [HOA78b, ZHO81, MIS81] in Chapter 6.

CHAPTER 2

EVENT BASED BEHAVIOR SPECIFICATION

2.1. The Conceptual Modeling

A distributed system may be described from two different points of view. From a designer´s viewpoint, it consists of local processes interacting with users and communicating among themselves via a communication medium. Each local process can be described by the operations responding to users´ commands, messages from other processes or internal clocks. The structure is depicted in Figure 2.1.

From a user´s viewpoint, a distributed system is a black box, or a shared server with only the interfaces visible to him, as shown in Figure 2.2. In this case, except for performance issues, there is no essential difference in functionality between a distributed system and a centralized one.

The distinction between a user´s view and a designer´s view is quite crucial. The only things interesting to a user are the kind of messages or events that may happen in the interfaces and the relationships among those messages or events. We call this kind of

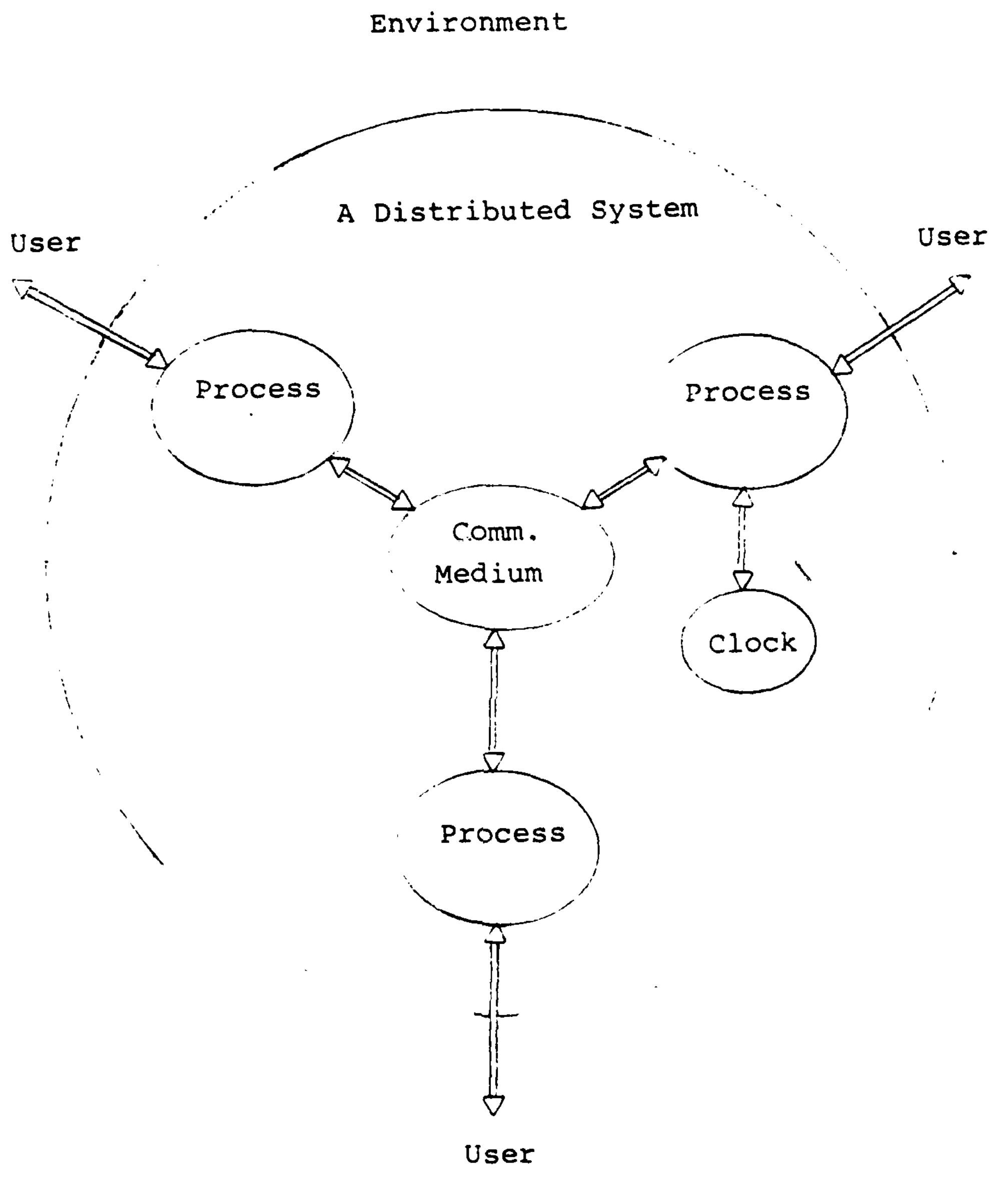


Figure 2.1. A Distributed System:
a Designer's View

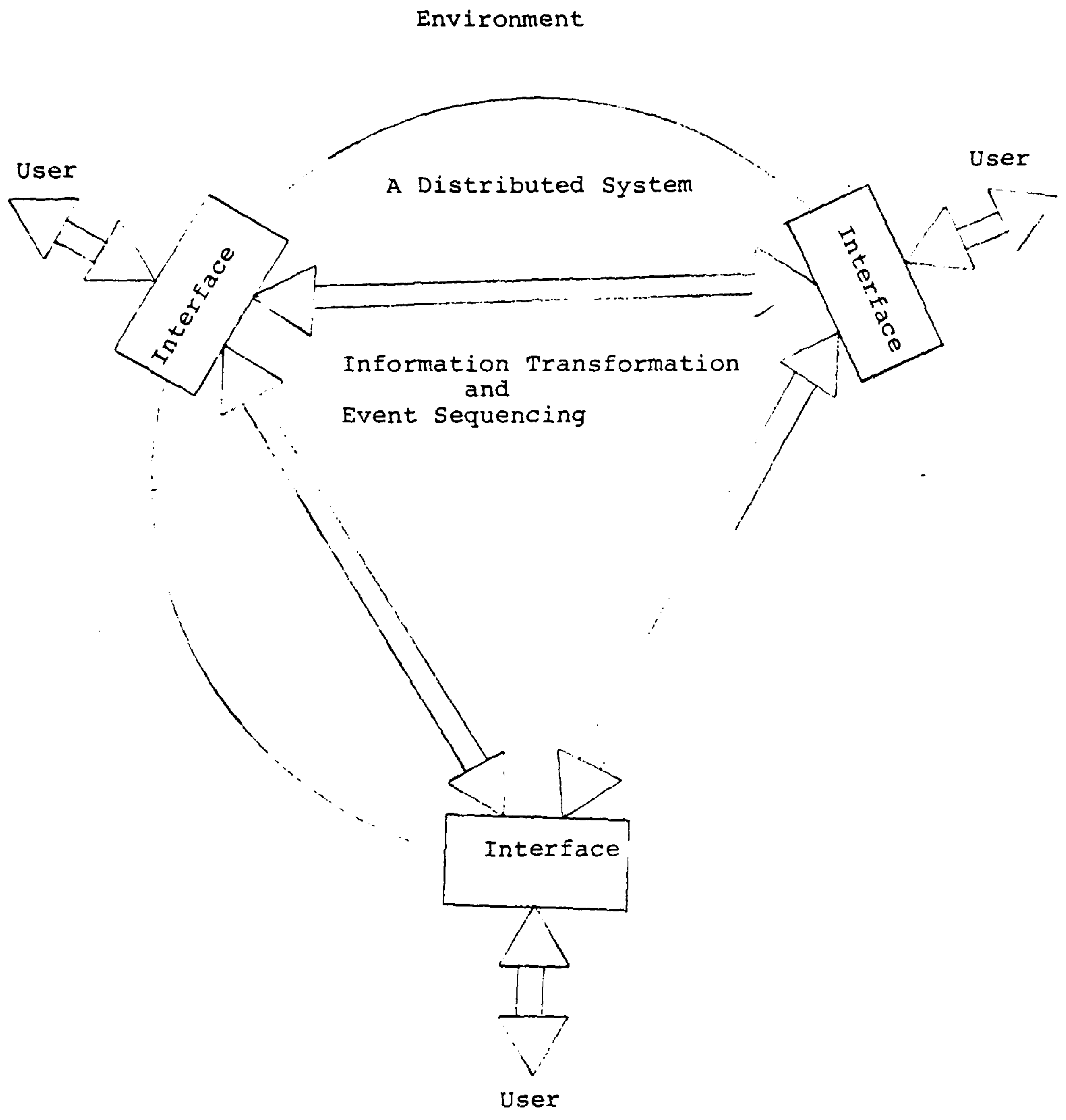


Figure 2.2. A Distributed System: a User's or a System Analysist's View

interface description of a system, its behavior specification.

There are two fundamental issues in the behavior specification of a distributed system, namely, the notions of time and concurrency. The use of synchronization primitives such as monitors, in our opinion, are design decisions which are suitable from a designer's viewpoint, but are at too low a level for behavior specifications. The computation of a system consists of a set of events. By precisely describing the relationships between these events occurring in time, we can characterize the behavior of a system.

2.2. The Event Model

We consider the behavior of a system to be characterized by a set of events. The model upon which our specification is based therefore consists of events and their relationships.

2.2.1. Events

An event is an instantaneous, atomic state transition in the computation of a system. Examples of events are the sending, the receiving, and the processing of messages. By "instantaneous" we mean an event takes zero-time to happen. By "atomic" we mean an event happens completely or not at all. Since time is continuous, the probability that any two or more events happen simultaneously is zero. This assumption allows us to order events totally in a local area computation. Furthermore, we assume that the number of events between any two events is always finite. This criterion rules out infinitely fast machines, which are physically infeasible [HEW77].

2.2.2. Event Relationships

2.2.2.1. The Precedes Relation ->

In describing the time ordering among events, a system-wide reliable clock is usually assumed to order totally the events in a centralized system. Unfortunately, the assumption of a global clock is too strong in describing the computation of a distributed system. Theoretically speaking, it is impossible to order two events totally in some extreme case when they happen in two geographically separated places. Practically speaking, implementing such a global clock is quite expensive and unnecessary in a distributed system having highly autonomous computing nodes. We give up the global clock assumption and adopt a partial ordering relation- the "precedes" relation, denoted by "->", to represent the time concept [GRE77, LAM78].

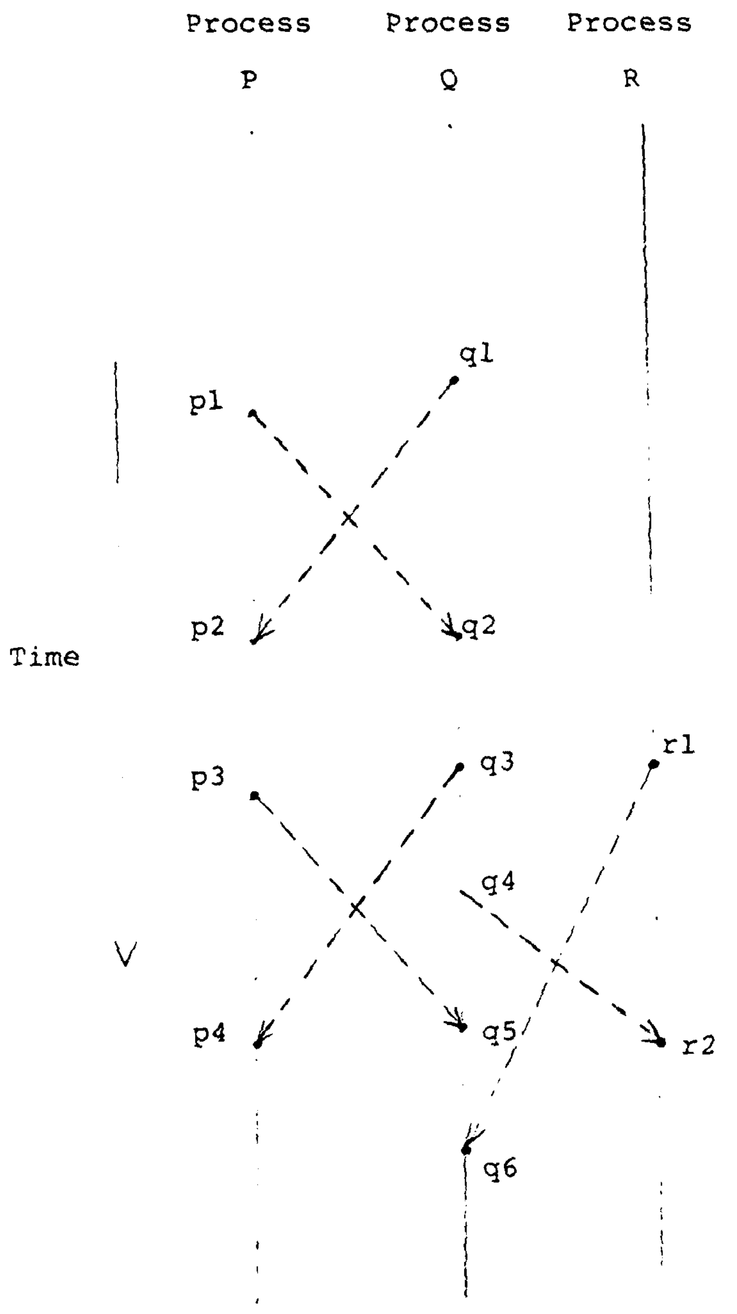


Figure 2.3. Precedes Relation between Events in Distributed Systems

The interpretation of "->" as a time ordering means that, if el and e2 are events in a system and el->e2, then el "precedes" e2 by any measure of time. To understand the meaning of "->", let us look into Figure 2.3. Each vertical line in Figure 2.3. represents the computation history of a (sequential) "process". By a "process" we mean an autonomous computing node having its own "local" clock; different processes may use different time scales. The dots denote events and the dotted lines between events denote messages. The relation "->" has the following properties:

(1) If el and e2 are events in the same process, and el comes before e2 then el->e2 (e.g., pl->p2 in Figure 2.3.);

(2) If el is the event of sending a message by one process and e2 is the event of receiving the message by another process then by the law of "causality", el->e2 (e.g., pl->q2 in Figure 2.3.);

(3) (Transitivity property) If el->e2 and e2->e3 then el->e3 (e.g., pl->r2 in Figure 2.3.);

(4) (Irreflexivity property) For all events, ~(e->e);

(5) (Antisymmetry property) If el->e2 then ~(e2->el)

2.2.2.2. The Concurrency Relation:

Two distinct events, say el and e2, are concurrent, denoted by el//e2, iff ~(el->e2) and ~(e2->el). In Figure 2.3., for example, although pl->q2 and ql->p2, there is no way to tell whether pl or ql comes first; they may be concurrent.

2.2.2.3. The Enables Relation =>

An important class of properties in communication systems is the guaranteed service of message transmission. These properties can be specified by the introduction of the enables relation, denoted by "=>", among events. Two events, say a and b, satisfy the relation a=>b iff the existence of event a will cause the occurrence of event b in the future. The relation => has the following properties:

```
(1) Future-enabling:
    if  a => b then  a -> b

(2) Anti-symmetry:
    if  a => b then  ~(b => a)

(3) Irreflexivity:
       ~(a => a)

(4) Transitivity:
    if a => b and b => c then a => c
```

Properties (2) and (3) can be derived from (1) and the properties of relation "->", while (1) and (4) are essential axioms for the relation "=>".

2.2.2.4. The System, the Environment, and Ports

Since the event space in a computation is usually very large, it is convenient to categorize events into some disjoint domains. We identify three domains: the system, the environment and the interface ports.

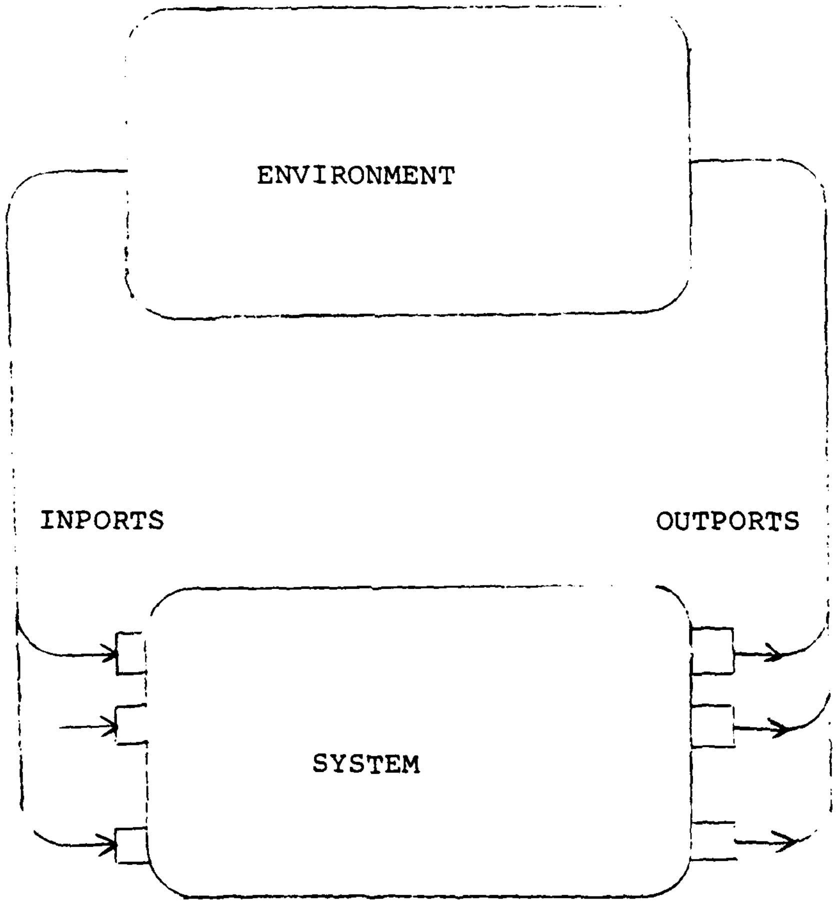


Figure 2.4. System, Environment and and Their Interfaces

A system interacts with its environment by exchanging messages through unidirectional interfaces called ports, as depicted in Figure 2.4. An inport (outport) directs messages from the environment (system) to the system (environment). The sending or receiving of messages are called interface events. We denote each port event set by the name of the port in capital letters and denote the contents of the messages carried by an interface event e by e.msg. In response to the interface events the system or the environment may change its state; such a state change is called a system or an environment event. We denote the system or environment event set by SYS or ENV, respectively. For example, in a banking system, a transfer command (an interface event) moves money from one account to another (a system event); and in an engine-monitoring system, a ring-bell message (an interface event) from the system will turn on a bell (an environment event) in the environment.

Events in the system or in the environment are only partially ordered, i. e., there is no assumption of a global clock either in the system or in the environment. However, we do require the total ordering among events in each port. This assumption is justified since that the probability that two events happen at the same time is zero and there is usually a local clock associated with each port. This assumption allows us to identify uniquely each interface event in a port history by an integer, called the ordinal number. If e is in a history H then its ordinal number can be defined recursively as follows:

```
ord(e, H)=
  IF (e=car(H)) THEN 1
     ELSE ord(e, cdr(H))+1
```

where car(H) represents the first event in H and cdr(H) represents the rest of the events in H. See Appendix A for details about formal definitions.

For convenience, we use ord(e) to abbreviate ord(e, H) when H can be understood from the context. A port history is bounded if the length of events in it is finite; otherwise, it is unbounded. Formally, a history H is unbounded iff

$$\forall n \in N \; \exists e \in H \quad [1]$$
$$ord(e) > n$$

where N is the set of natural numbers, or equivalently

$$(\exists e \in H) \wedge$$
$$(\forall e1 \in H \; \exists e2 \in H$$
$$e1 \rightarrow e2)$$

which says that every event in H has a successor event and

[1] See Section 2.3.1. for notation abbreviations.

that there is at least one event in the history H.

The concept of unbounded history is important when we want to specify a non-zero probability of message transmission over an unreliable transmission medium. (See Example 2.3.)

2.3. The Event-Based Specification Language (EBS)

Based on the notions of events, event relationships, together with first order predicate calculus, we develop a specification language called the Event-Based Specification Language (EBS).

2.3.1. First Order Predicate Calculus

The behavior specification language is the first order predicate calculus with equality. See Appendix A for the definitions of expressions, terms, atomic formulas, and well-formed formulas. We give here the precedence rules among symbols and the abbreviation rules.

2.3.1.1. Precedence Rules

The precedence among symbols are as follows:

(1) < (belongs to), ->, =>, = (Equals to)
(2) ∀ (for all), ∃ (there exists), ~ (logical not)
(3) ^ (logical and), v (logical or)
(4) #> (logical implication)
(5) when one connective is used repeatedly, the expression is grouped to the right, for example A#> B #> C is A#> (B#> C).

2.3.1.2. Abbreviation Rules

We use the following notation abbreviation rules:

```
(1) ∀ x< A  S abbreviates ∀x (x< A #> S)
(2) ∀ x, y< A S abbreviates ∀x ∀y (x< A ^ y< A #> S)
(3) ∀ x< A, y< B S abbreviates ∀x ∀y (x< A ^ y< B #> S)
(4) ∃ x< A S abbreviates ∃x (x< A ^ S)
(5) ∃ x, y< A S abbreviates ∃x ∃y (x< A ^ y< A ^ S)
(6) ∃ x< A, y< B S abbreviates ∃x ∃y (x< A ^ y< B ^ S)
(7) a-> b-> c abbreviates a-> b ^ b-> c
(8) a=> b=> c abbreviates a=>b ^ b=> c
(9) x= y abbreviates = x y
(10) Similar rule for other two-place predicates
(11) x<> y abbreviates ~ = x y
(12) Outermost parenthesis may be dropped
```

2.3.2. The Syntax of EBS

The syntax of EBS is defined in extended BNF as follows:

```
<system>::= System <head>
                   <message type definition list>
                   <behavior>
                   <structure>
               End system.
<head>::= <id> ({<parameter>;} <parameter>);
<parameter>::= <id> : <parameter type>
<parameter type>::= inport | outport | function
                  | predicate
<message type definition list>
              ::= Messagetype
                     [message type definition;]
                  End messagetype; | <empty>
<message type definition>::= <id> : <data type>
<data type>::= <simple type> | <structure type>
<simple type>::= integer | character|
                 real | boolean
<structure type>::= record
                       [<id> : <data type>;]
                    end
<behavior>::= Behavior
                 [<wff>;]
              End behavior; | <empty>
<structure>::= Structure
                  [<subsystem>;]
```

```
                     <network>;
                     <interface>;
                  End structure; | <empty>
<subsystem>::= <system>
<network>::= Network
                    [link(<portname>, <portname>)
                                  == <portname>;]
                 End network
<interface>::= Interface
                     [<portname> == <portname>;]
                  End interface
<portname>::= <id>.<id>
<empty>::=
```

A specification begins with the reserved word _System_ followed by the name of the system and the names of interface ports. The message type definition list defines the data types of messages associated with each interface port.

The behavior part consists of a sequence of well-formed formulas (wffs) of first order predicate calculus separated by semicolons. The structure, subsystem, network, and interface parts are used in system structure specification which will be discussed in Chapter 3. To support extensible specifications, the message type definitions, the behavior part and the structure part are not required initially. Any of them can be deferred to later phases of the system development.

2.4. Example Distributed Systems and Their Specifications

In this section, we demonstrate the power of EBS by specifying some typical examples of distributed systems.

2.4.1. Example 2.1: Reliable Transmission Systems

A reliable transmission system (RT) is one through which messages are transmitted without error, loss, duplication or reordering from an inport to an outport (see Figure 2.5). Although most physical communication media are unreliable, almost all designers provide communication protocols (e.g., the Alternate Bit Protocol) to convert them into logically reliable ones. It is important that the service provided by the reliable transmission system be specified formally. We first specify orthogonal properties and then integrate them into a complete specification of the whole system.

The property that there is no loss of messages during the transmission means that every message sent from the inport A will be transmitted to the outport B eventually. This can be specified as follows:

```
(* RT11(A,B) [2] : No loss of messages *)
   ∀ a< A ∃ b< B
     a=> b;
```

which says that for every event a in the history of A

[2] We will use RT11 to name this property afterwards for convenience.

there is an event b in the history of B such that a enables b. Similarly, the property that messages at B are not generated internally or externally but are enabled by messages at A, is specified as follows:

```
(* RT12(A,B): no self-existing messages *)
   ∀ b∈ B ∃ a∈ A
     a=> b;

(* RT13(A,B): no internally or externally
              generated messages
*)
   ∀ b∈ B,  s∈ SYS, e∈ ENV
     (s =>b #> ∃ a∈ A a=>s=>b) ^
     (e =>b #> ∃ a∈ A e=>a=>b)
```

The reserved word SYS (ENV) refers to the set of system (environment) events. RT13 specifies that any internal (external) event that enables an event in B, is enabled by (enables) an event in A; in other words, A is the *only* source that may enable an event in B.

No duplication of messages is specified as follows:

```
(* RT14(A,B): no duplication of messages *)
   ∀ a∈ A, b1,b2∈ B
     a=>b1 ^ a=>b2 #> b1=b2
```

which says that every sending event can only enable a unique receiving event. The property that the order of messages is preserved after the transmission is specified as follows:

```
(* RT15(A,B): no out-of-order messages *)
  ∀ a1,a2∈ A, b1,b2∈ B
    a1=>b1 ^ a2=>b2
    #> (a1-> a2 ^ b1-> b2) v
       (a1= a2 ^ b1= b2) v
       (a2-> a1 ^ b2-> b1)
```

which says that if a1 is sent before (after) a2 then it will also be received before (after) a2. The contents of messages are preserved after the transmission is specified as follows:

```
(* RT21(A,B): preservation of message contents *)
  ∀ a∈ A, b∈ B
    a=>b #> a.msg=b.msg
```

which says that the receiving and the sending events carry the same message contents.

We now specify the reliable transmission system as follows:

```
(* Reliable Transmission Systems *)
System RT ( A: inport;
            B: outport);

   Behavior

      (* RT11(A,B): No loss of messages *)
         ∀ a∈ A ∃ b∈ B
           a=>b;

      (* RT12(A,B): No self-existing messages *)
         ∀ b∈ B ∃ a∈A
           a=> b;

      (* RT13(A,B): No internally or externally
         generated messages
      *)
         ∀ b∈ B, s∈ SYS, e∈ ENV
           (s=> b #> ∃ a∈ A a=> s=> b) ^
```

```
                (e=> b #> ∃ a< A e=> a=> b);

           (* RT14(A,B): No duplication of messages *)
              ∀ a< A, b1,b2< B
                a=> b1 ^ a=> b2 #> b1= b2;

           (* RT15(A,B): No out-of-order messages *)
              ∀ a1, a2< A, b1,b2< B
                a=> b1 ^ a2=> b2
                #> (a1-> a2 ^ b1-> b2) v
                   (a1= a2 ^ b1= b2) v
                   (a2-> a1 ^ b2-> b1)

           (* RT21(A,B): No erroneous messages *)
              ∀ a< A, b< B
                a=> b #> a.msg= b.msg;

        End behavior;

     End system.
```

These are the weakest properties that a reliable transmission system should have. This kind of "orthogonal" specification allows a specification to be easily adapted to different applications. For example, if we want to specify the behavior of a communication system which transmits messages reliably and performs code conversions between computer systems communicating using different codes (e.g., ASCII and EBCDIC), we need only change RT21, the data-related property. It is changed to

```
(* TR21(A,B): message transformer *)
   ∀ a< A, b< B
     a=> b #> b.msg= F(a.msg)
```

where F is the code conversion function, and leave the other specifications unchanged.

2.4.2. Example 2.2: Multiplexors and Decoders

A communication network has to share its transmission capacity among all its users who, in general, do not want to use and pay for a transmission circuit all the time. In packet-switched communication systems, the information exchanged between computers or terminals are sequences of short messages called packets, each handled individually by the network mechanism. Packets can be entered into or removed from the network at a speed suitable for the computers or terminals, so the network acts as a speed changer. The fundamental mechanisms in packet-switched networks to achieve this goal are multiplexing and decoding. Multiplexing consists of interleaving packets from various sources of data in a single communication channel. By recognizing the packets according to the information contained in their messages, a decoder distributes the packets to their respective destinations. This section specifies the behavior of multiplexors and decoders.

A multiplexor can be viewed as a system through which messages from several inports are transmitted reliably and multiplexed into a single outport (see Figure 2.5.). We specify a multiplexor with two inports as follows:

```
        System MX (A : inport;
                   B : inport;
                   C : outport);
```

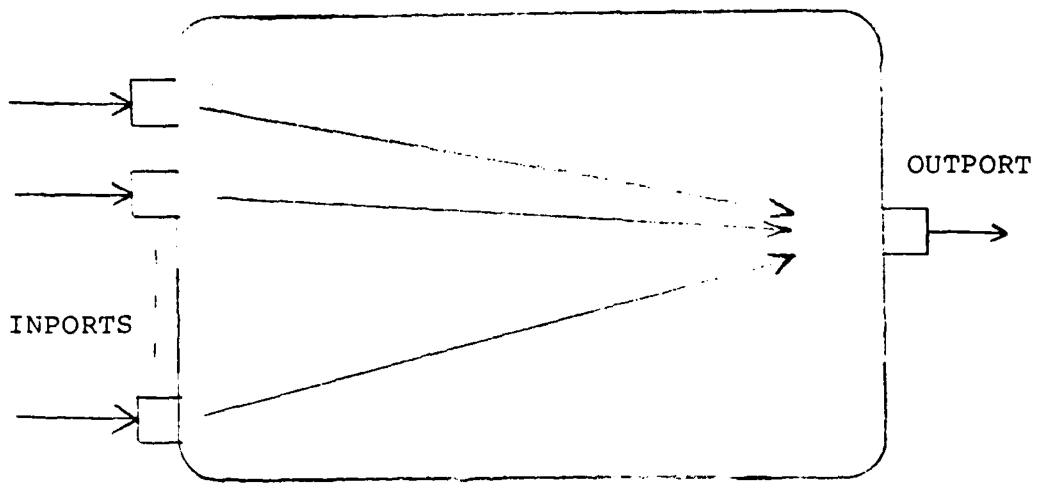


Figure 2.5. A Multiplexor

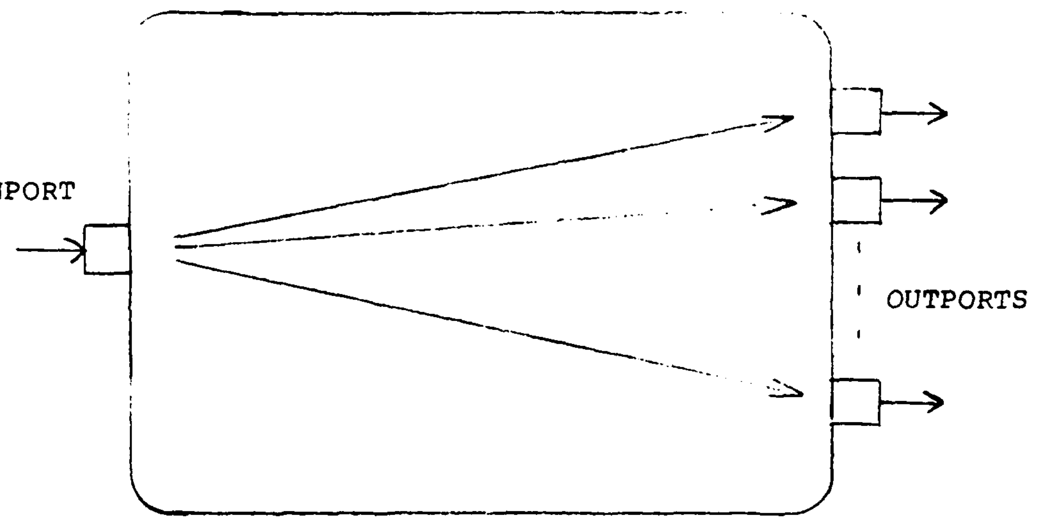


Figure 2.6. A Decoder

```
      Behavior

         (* No loss of messages *)
            RT11(A, C); RT11(B, C);

         (* No self-existing messages *)
            ∀ c< C
               ( ∃ a< A a=> c) v ( ∃ b< B b=> c);

         (* No internally or externally generated
            messages
         *)
            ∀ c< C, s< SYS, e< ENV
               (s=> c #> ( ∃ a< A a=> s=> c) v
                         ( ∃ b< B b=> s=> c)) ^
               (e=> c #> ( ∃ a< A e=> a=> c) v
                         ( ∃ b< B e=> b=> c));

         (* No duplication of messages *)
            RT14(A, C); RT14(B, C);

         (* No out-of-order messages *)
            RT15(A, C); RT15(B, C);

         (* No erroneous messages *)
            RT21(A, C); RT21(B, C);

      End behavior;

   End system.
```

Note that the RT's have been defined in the System RT (Reliable Transmission Systems). We only specify the first-come-first-serve order of events in port A or in port B. The order among events in A and B (i.e., the multiplexing function) is left open to implementation decisions. Note also that it is easy to extend the specification to define the behavior of multiplexors having more than two inports.

A decoder can be viewed as a system that distributes messages reliably from a single inport to several outports according to some predefined distribution criteria (see Figure 2.6.). Before specifying decoders, we define first another important class of network behavior called filters. A filter is a reliable system that transmits messages that satisfy some predefined criteria; other messages are filtered out and have no effects. It can be specified as follows:

```
System FT (A : inport;
           B : outport;
           P : predicate);

   Behavior

      (* Every message at A that satisfies P
         will be sent to B
      *)
         ∀ a∈ A
           P(a) ⇒ ∃ b∈ B a=> b;

      (* Every message received by B should satisfy
         P and have been sent from A
      *)
         ∀ b∈ B ∃ a∈ A
           P(a) ^ a=> b;

      (* No internally or externally generated
         messages
      *)
         RT13(A, B);

      (* No duplication of messages *)
         RT14(A, B);

      (* No out-of-order messages *)
         RT15(A, B);

      (* No erroneous messages *)
         RT21(A, B);
```

```
          End behavior;

       End system.
```

A decoder is, essentially, just a set of such filters. A decoder with N outports B [1..N] and distribution criteria P[1..N] is specified as follows:

```
       System DC (A : inport;
                  B[1..N] : outport; [3]
                  P[1..N] : predicate);

          Behavior
             ∀ i∈ {1..N}
                FT(A, B[i], P[i])
          End behavior;

       End system.
```

2.4.3. Example 2.3: Unreliable Transmission Systems

In an unreliable transmission system, messages may be lost, duplicated or reordered, but there is a non-zero probability of message transmission and no erroneous messages. Most physical communication media belong to this class.

A non-zero probability of message transmission can be specified as follows:

---

[3] B[1..N]: outport represents B[1]: outport; .., B[N]: outport.

```
(* NZ(A,B): a nonzero probability of successful
            message transmission.
*)
   ∀ ai< A
     (∀ aj< A aj.msg= ai.msg
        #> ∃ ak< A aj-> ak ^ ak.msg= ai.msg)
     #> (∃ a< A, b< B
           a=> b ^ a.msg= ai.msg ^ ai-> a)
```

which means that if a group of messages having the same contents are repeatedly sent then at least one of them will reach B.

The unreliable transmission system is specified as follows:

```
System  UT (A : inport;
            B : outport);

Behavior

   (* NZ(A,B): a nonzero probability of successful
               message transmission.
   *)
      ∀ ai< A
        (∀ aj< A aj.msg= ai.msg
           #> ∃ ak< A aj-> ak ^ ak.msg= ai.msg)
        #> (∃ a< A, b< B
              a=> b ^ a.msg= ai.msg ^ ai-> a);

   (* No self-existing messages *)
      RT12(A, B);

   (* No internally or externally generated messages *)
      RT13(A, B);

   (* No erroneous messages *)
      RT21(A, B);

   (* RT11, RT14 and RT15 are discarded, which means *)
   (* that the system may lose, duplicate or reorder *)
   (* messages.                                      *)

  End behavior
```

End system.

2.4.4. Example 2.4: Data Transfer Protocols

The protocol described in [STE76] provides reliable message transmission service over unreliable transmission media. Stenning defined two processes: a SENDER and a RECEIVER. The sender sends messages from a given sequence to the receiver using a communication line. The receiver in turn accepts messages from the line, stores them in an output sequence, and acknowledges their receipt by sending a message to the sender via another communication line. Both communication lines are unreliable.

The protocol uses a conventional positive-acknowledgement, retransmission-on-time-out technique, and the sender and the receiver both maintain windows of messages. The sender's window contains messages sent but not yet acknowledged. Similarly, the receiver buffers messages received out of order (up to some limit), awaiting receipt of the next expected message.

The implementation of this data transfer protocol depicted in Figure 2.7, is non-trivial. However, to the users of this protocol, the only thing interesting is that the protocol provides a reliable transmission service with two restrictions: (1) the length of the messages in tran-

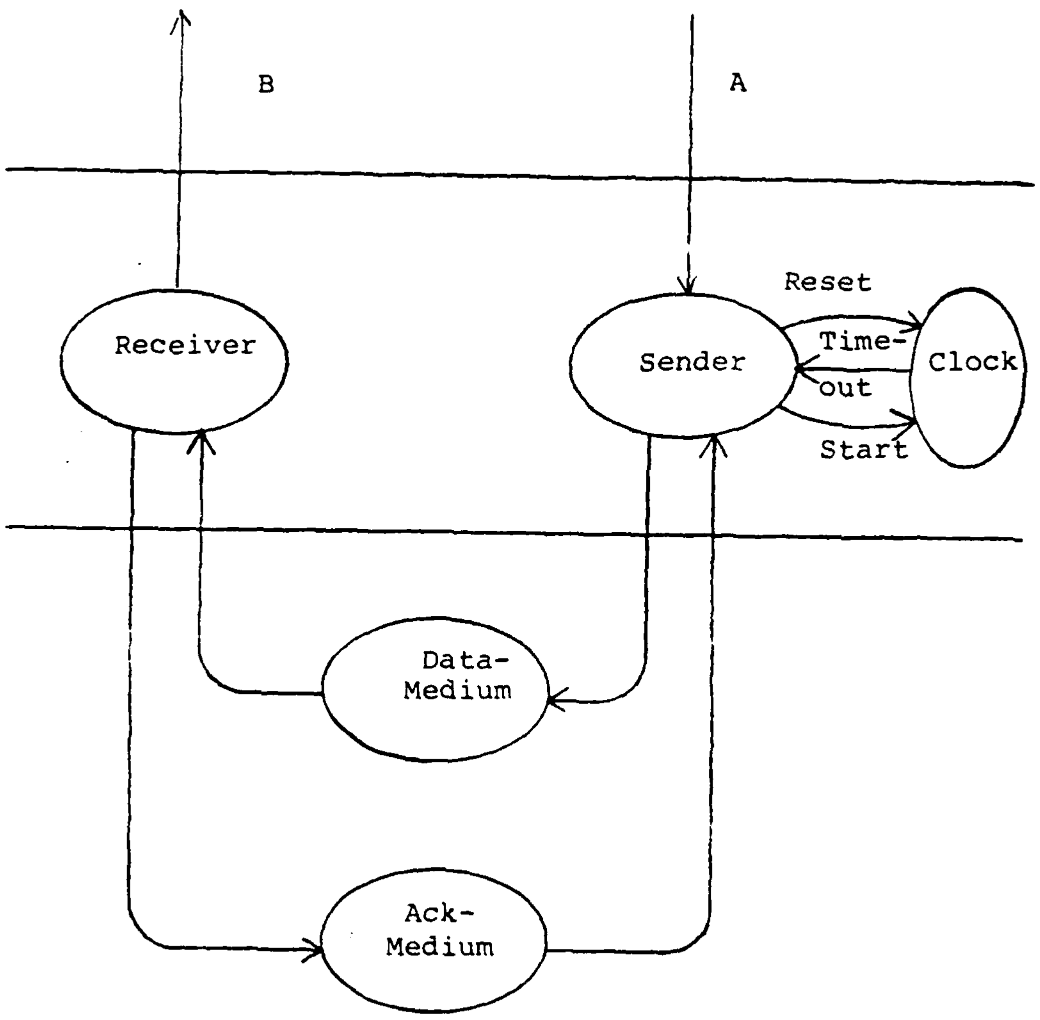


Figure 2.7. An Implementation Structure of the Data-Transfer Protocol

sit cannot be longer than the sender´s window size and (2) received messages can be buffered out of order only if the length of the messages does not exceed the receiver´s window size. The behavior of this protocol can be described as follows:

```
(* Stenning´s Data Transfer Protocol *)
System DTP( A: inport;
            B: outport);

   Behavior
      (* No loss of messages *)
         RT11(A, B);

      (* No self-existing messages *)
         RT12(A, B);

      (* No internally or externally generated *)
      (* messages                               *)
         RT13(A, B);

      (* No duplication of messages *)
         RT14(A, B);

      (* DTP1: sender´s window control: one *)
      (* cannot send more than sw messages  *)
      (* if the first is not acknowledged,  *)
      (* where sw is the sender´s window    *)
      (* size.                              *)
         ∀ a1, a2∢ A, b1∢ B
           a1=> b1 ^ ord(a2)≥ ord(a1)+ sw
           #> b1-> a2;

      (* DTP2: receiver´s window control:   *)
      (* one cannot receive ahead more than *)
      (* rw out-of-order messages, where    *)
      (* rw is the receiver´s window size   *)
         ∀ a1, a2∢ A, b1, b2∢ B
           a1=> b1 ^ a2=> b2 ^ ord(a2)≥ ord(a1)+ rw
           #> b1-> b2;

      (* No erroneous messages *)
         RT21(A, B);

    End behavior;
```

End system.

When both the sender´s and the receiver´s window sizes are reduced to one, there is neither concurrent transmission of several messages from the sender nor acceptance of out-of-order messages by the receiver. In this case, DTP2 (the receiver´s window control) is equivalent to RT15 in the Reliable Transmission System; and DTP1 (the sender´s window control) means that the sender has to wait until all previous messages are received successfully before sending the next message. This is the service provided by the Alternate Bit Protocol [BAR69].

2.4.5. Example 2.5: an Engine Monitoring System

A microprocessor aircraft engine monitor for use on both experimental and in-service aircrafts is described in [ALF77]. The capability of this Engine Monitoring System is as follows:

1. Monitor 1 to 10 engines
2. Monitor
   a. 3 temperatures
   b. 3 pressures
   c. 2 switches
3. Monitor each engine at a specific rate.
4. Output a warning message if any parameter falls outside prescribed limits.
5. Activate an audio alarm if any parameter falls outside prescribed limits.
6. Record the history of each engine.
7. The operator may change the warning or alarm

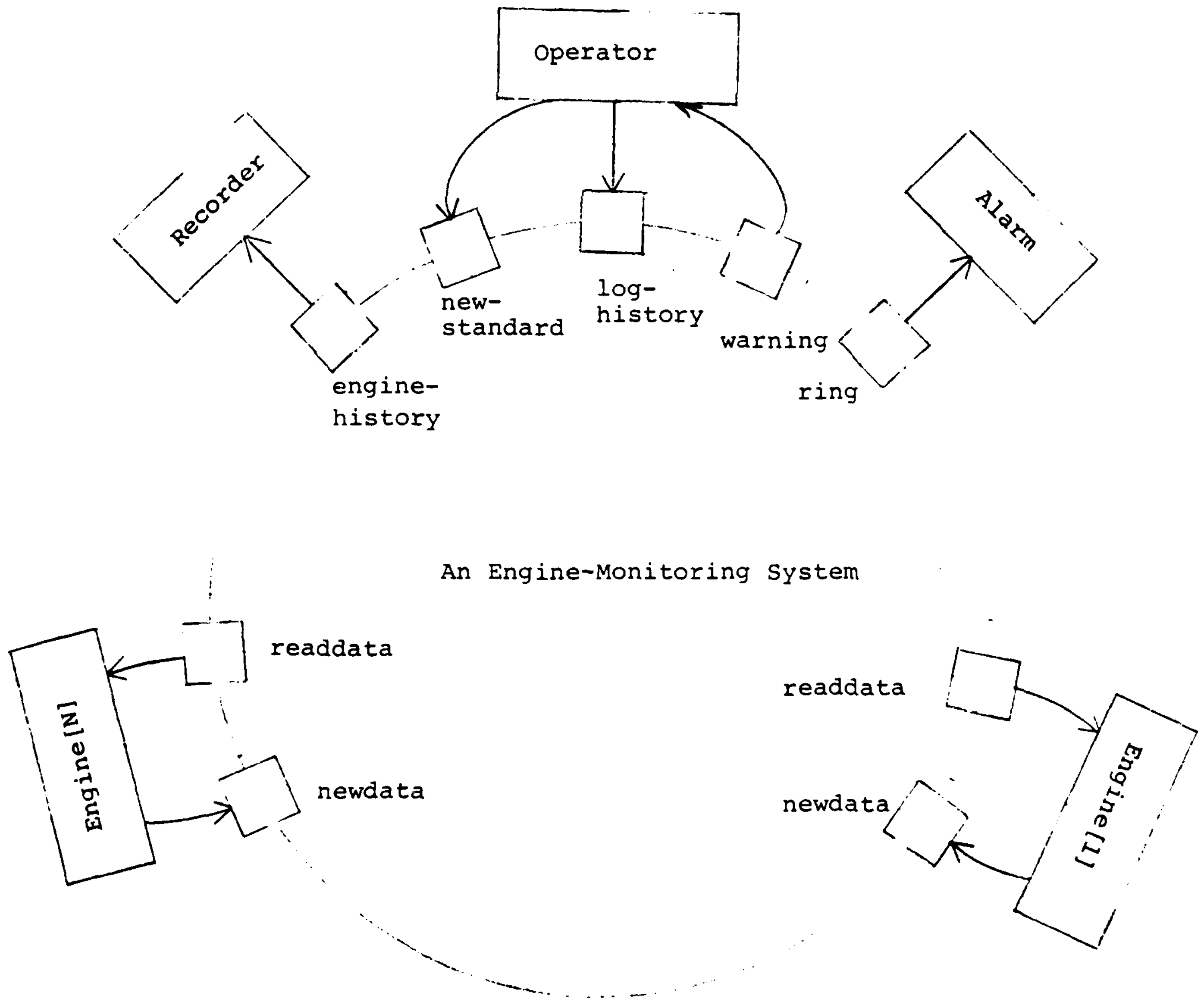


Figure 2.8. An Engine-Monitoring System

limits and may log the history of each machine.

The system interface structure is depicted in Figure 2.8. We specify this system in EBS as follows:

```
(* Engine Monitoring System *)
System EMS (
            engine[i]|newdata: inport;
             (* Comment i:= 1 to 10.
                A port pt of engine[i] is
                represented by engine[i]|pt
             *)
            log-history: inport;
            new-standard: inport;
            engine[i]|readdata: outport;
              (* i:= 1 to 10 *)
            warning: outport;
            ring: outport;
            engine-history: outport
            inwarning: predicate;
            inalarm: predicate;
            realtime: function
           );
   Messagetype
      newdata.msg: record
                     T1, T2, T3,
                     P1, P2, P3: real;
                     S1, S2: boolean;
                     Time: real;
                   end;
      log-history.msg: integer;
      new-standard.msg:
         record
           id: integer;
           (* Comment: id is the engine name
              whose standard is to be changed
           *)
           engine-standard:
             record
               UWT1, (* Upper warning margin for T1 *)
               LWT1, (* Lower warning margin for T1 *)
                ...
               UAT1, (* Upper alarm margin for T1 *)
               LAT1, (* Lower alarm margin for T1 *)
                ...
                   : real;
             end;
         end;
   engine-history.msg:
      record
        id: integer;
        engine-data:
           record
             T1, T2, T3,
             P1, P2, P3: real;
             S1, S2: boolean
             Time: real;
           end;
        end;
     warning.msg: integer;
        (* Comment: The name of the engine that
           is in warning range.
        *)
     ring.msg: boolean;
     engine[i]|readdata.msg: boolean;
       (* i:= 1 to 10 *)
Behavior
   (* Part I: System's response to a newdata *)
   (* when a newdata comes, check its security
      against the most recent engine standard
      set up by new-standard. If it is in warn-
      ing range then send out a warning message;
      if it is in alarm range then ring the
      bell; otherwise, do nothing but record
      the newdata in memory.
   *)

   (* Part I.1: The relationship between inport
      newdata and outport warning: output a
      warning message if and only if a newdata
      is in warning range.
   *)

       (* NW11: If a newdata is in warning range
          then send a warning message.
       *)
          ∀ i∈ {1..10},
            x∈ engine[i]|newdata,
            mrs∈ new-standard
            (* mrs is the event of setting up the
               most recent standard
            *)
            mrs.msg= i ^ mrs-> x ^
            ( ∀ c∈ new-standard
                c.msg= i ^ c-> x #> c= mrs v c-> mrs) ^
            inwarning(x.msg, mrs.msg)
            #> ∃ w∈ warning x=> w;

       (* NW12: Send a warning message only when a
        new data is in warning range.
      *)
        ∀ w< warning
        ∃ i< {1..10}
          x< engine[i]|newdata,
          mrs< new-standard
          mrs.msg= i ^ mrs-> x ^
          ( ∀ c< new-standard
              c.msg= i ^ c-> x #> c= mrs v c-> mrs) ^
          inwarning(x.msg, mrs.msg) ^ x=> w;

    (* NW13: No internally or externally generately
       messages.
    *)
       ∀ w< warning, s< SYS, e< ENV
         (s=> w #> ∃ i< {1..10}, x< engine[i]|newdata
                     x=> s=> w) ^
         (e=> w #> ∃ i< {1..10}, x< engine[i]|newdata
                     e=> x=> w);

    (* NW14: No duplication of messages *)
       ∀ i< {1..10},
         x< engine[i]|newdata,
         w1, w2< warning
         x=> w1 ^ x=> w2 #> w1= w2;

    (* NW15: No out-of-order messages *)
       ∀ i< {1..10},
         x1, x2< engine[i]| newdata,
         w1, w2< warning
         x1=> w1 ^ x2=> w2
         #> (x1= x2 ^ w1= w2) v
            (x1-> x2 ^ w1-> w2) v
            (x2-> x1 ^ w2-> w1);

    (* NW21: Contents of warning messages *)
       ∀ i< {1..10},
         x< engine[i]|newdata, w< warning
         x=> w #> w.msg= i;

(* Part I.2: The relationship between inport
   newdata and outport ring: Output an alarm
   message if and only if a newdata is in
   alarm range.
*)

    (* The behavior of outport ring is essentially
       the same as that of warning and is omitted
       here.
    *)
```

```
(*Part II: System's response to a log-history command *)

    (* LH11: A log-history[i] command will record into
       engine-history all previous engin[i]|newdata
       events.
    *)
       ∀ i∢ {1..10}, x∢ log-history
         x.msg= i #>
         ( ∀ e∢ engine[i]|newdata
             e-> x #> ∤ h∢ engine-history
                        x=> h ^ h.enginedata= e.msg
                              ^ h.id= i);

    (* LH12: Engine-history is enabled by a log-
       history command.
    *)
       ∀ h∢ engine-history
       ∤ x∢ log-history, e∢ engine[h.id]|newdata
         x=> h ^ x.msg= h.id ^ e.msg= h.enginedata;

    (* LH13: No internally or externally generated
       messages.
    *)
       ∀ s∢ SYS, e∢ ENV, h∢ engine-history
         (s=> h #> ∤ x∢ log-history x=> s=> h) ^
         (e=> h #> ∤ x∢ log-history e=> x=> h);

    (* LH14: No duplication of messages *)
       ∀ h1, h2∢ engine-history, x∢ log-history,
         e1, e2∢ engine[x.msg]|newdata
         x=> h1 ^ x=> h2 ^ h1.enginedata= h2.enginedata
         #> h1= h2;

    (* LH15: No out-of-order messages *)
       ∀ x1, x2∢ log-history, h1, h2∢ engine-history
         x1=> h1 ^ x2=> h2
         #> (x1= x2 ^ h1= h2) v
            (x1-> x2 ^ h1-> h2) v
            (x2-> x1 ^ h2-> h1);

(* Part III: The behavior of outport readdata *)

    (* Read engine data periodically and infinitely  *)
    (* T is the period between two successive read's *)
    (* Realtime is a function mapping from two events
       to a real number meaning the time period between
       the two events.
    *)
       ∀ i∢ {1..10}, x, y∢ engine[i]|readdata
         ord(y)= ord(x)+ 1
         #> realtime(x, y)= T;
```

```
          (* Read engine data infinitely *)
             ∀ i< {1..10}, x< engine[i]|readdata
             ∄ y< engine[i]|readdata
               x-> y;

     End behavior;

  End system.
```

This Engine-Monitoring System example shows the capability of EBS in dealing with "side-effects". We are not specifying the effects of a command by changing the values of system "state variables", since no such variables are allowed in the specification. A "pure" side-effect command does not have any visible effect until other commands reference its message. For example, the effect of a "newstandard" command is defined by its relations with "newdata" commands in NW11 and NW12. The specifications show that only the most recent "newstandard" is of concern to a "newdata"; other old "newstandards" do not have any effect. This is a natural and direct way in specifying the system behavior from a user's view point.

2.5. Conclusions and Comparisons to Other Approaches

SPECIAL [ROB77] and AFFIRM [TOM80] are currently being used as the service specification techniques for communication protocols. SPECIAL, a tool developed for the design of large software systems, is based on a methodology using the concept of a hierarchy of modules. A module is specified using Parnas' State-Machine approach. AFFIRM was developed as a general-purposed specification and verification system. It combines the State-Machine Model and Algebraic/Axiomatic specification techniques into an integrated methodology.

Total ordering of events underlies the semantics for both SPECIAL and AFFIRM; there is little support for concurrency expressions. In addition, "liveness" properties of concurrent systems, such as the guaranteed message transmission service, are not specified in either language. In comparison, EBS takes the buffer-history approach [SUN79], specifying a system in terms of input/output relations. The time concept is represented by a partial ordering relation of events. Concurrency is expressed by the lack of order between events. This makes EBS a more accurate model for distributed systems. "Liveness" properties are specified in EBS with the "enables" relation among events.

The requirement language RSL is used in [ALF77] to specify the Engine Monitoring System. A central information processing system is assumed in the specification to request and test each new incoming engine data serially, which is more a design decision than a requirement. Moreover, state variables are used extensibly to specify the effects of commands, which complicates the specification and blurs the distinction between the external behavior and the internal structure of the system.

We conclude our discussion by listing some benefits of Event-Based Behavior Specification: (1) formality: partial ordering relations and first order predicate calculus are mathematically sound; (2) generality: "safety", "liveness", data-related and control-related properties can be specified; (3) accuracy: the inherent concurrent behavior of distributed systems is represented by the lack of ordering among events; (4) orthogonality: properties are specified separately making a specification minimal and extensible.

CHAPTER 3

STRUCTURE SPECIFICATIONS AND VERIFICATIONS

3.1. Introduction

The event-based model developed in Chapter 2 specifies the behavior (the external view) of distributed systems. Both control-related and data-related properties of distributed systems are specified using two fundamental relationships among events: the "precedes" relation, representing time order; and the "enables" relation, representing causality. No assumption about the existence of a global clock is made in the specifications.

In this Chapter, the event-based model is extended to specify the design structure (the internal view) of distributed systems; we call such specifications "structure specifications". The important structure concepts such as "subsystems", which simulate the communicating computing nodes, and "connection links", which simulate the communication networks interconnecting subsystems, are defined formally in terms of events, and then used as basic constructs in the structure specification of distributed systems.

The same nonprocedural specification language, the Event-Based Specification Language (EBS), is used both in the behavior and the structure specifications. An immediate benefit of this homogeneous view of behavior and structure specifications is that design verifications can be carried out as proofs of theorems. It also supports top-down, hierarchical designs, since each subsystem in the structure specification can be further decomposed into sub-subsystems, using the same specification language. Both "safety" and "liveness" properties of concurrent systems [LAM78], can be formally specified and verified.

3.2. System Constructs

A system can be structurally decomposed into a set of _subsystems_, which are the basic building blocks of the system; a set of connection _links_, via which subsystems exchange messages; and a set of _interface definitions_, via which the whole system communicates with its environment.

A subsystem, except for the naming convention, is a system, whose behavior is described in EBS and whose structure may be further decomposed. From the event viewpoint, a subsystem defines the boundary between its event set and its enclosing system event set.

A _link_ connects an outport of a subsystem to an inport of another subsystem. When two ports are linked, they are *merged* into a single port. The event semantics of a link is that ports are *identical*: any event for one is an event for the other.

A link is different from a reliable transmission system (see Chapter 2) in that the latter introduces a finite message delay while the former transmits messages reliably without delay. Furthermore, two ports cannot be linked unless they have identical message types.

As the links supply paths for message exchanges among subsystems, an *interface definition* defines an interface path through which the whole system communicates with its environment.

The properties of a system structure specification can be summarized as follows: if a system S is composed of a set of subsystems S1, ..., Sm, a set of links connect(P1, Q1)==CL1, ..., connect(Pn, Qn)==CLn and a set of interface definitions X1==Y1, ..., Xk==Yk then

(A) Subsystems and connection links are *mutually exclusive*, *collectively exhaustive subsets* of the whole system:

(1) subsets:

```
∀ e∈ SYS(Si) e∈ SYS(S)
        for all subsystems Si; and
∀ e∈ CLi e∈ SYS(S) for all links CLi;
```

(2) mutual exclusion

```
 ∀ e∈ Si ~(e∈ Sj)
        for all Si and Sj, Si<> Sj;
        ∀ e< CLi ~(e< CLj)
              for all Ci and CLj, CLi<> CLj; and
        (∀ e< Si ~(e< CLj)) ^ (∀ e< CLj ~(e< Si))
              for all Si and CLj;

    (3) Collective exhaustion

        ∀ e< SYS(S)
          e< SYS(Si) for some Si or
          e< CLj for some CLj

(B) Connect(Pi, Qi)==CLi means Pi= Qi, i.e.,
    (∀ e< Pi e< Qi) ^ (∀ e< Qi e< Pi);
    and the link is renamed CLi of system S.

(C) Xi==Yi means that Xi of a subsystem is  renamed  Yi
    of system S.
```

3.3. Example 1: a Tandem Network

In a packet-switched network, a packet of messages is reliably transmitted through intermediate nodes rather than sent directly from the source node to the destination node using a dedicated, long-haul transmission line. Thus, the structure of the. communication system can be considered as a set of reliable transmission subsystems connected in series which, as a whole, supply the service of a reliable transmission system. We call a serial connection of two or more subsystems, a tandem network (see Figure 3.1).

3.3.1. The Structure Specification of a Tandem Network

The structure specification of a system SC which is composed of a serial connection of two reliable transmission systems, SA and SB, is specified as follows:

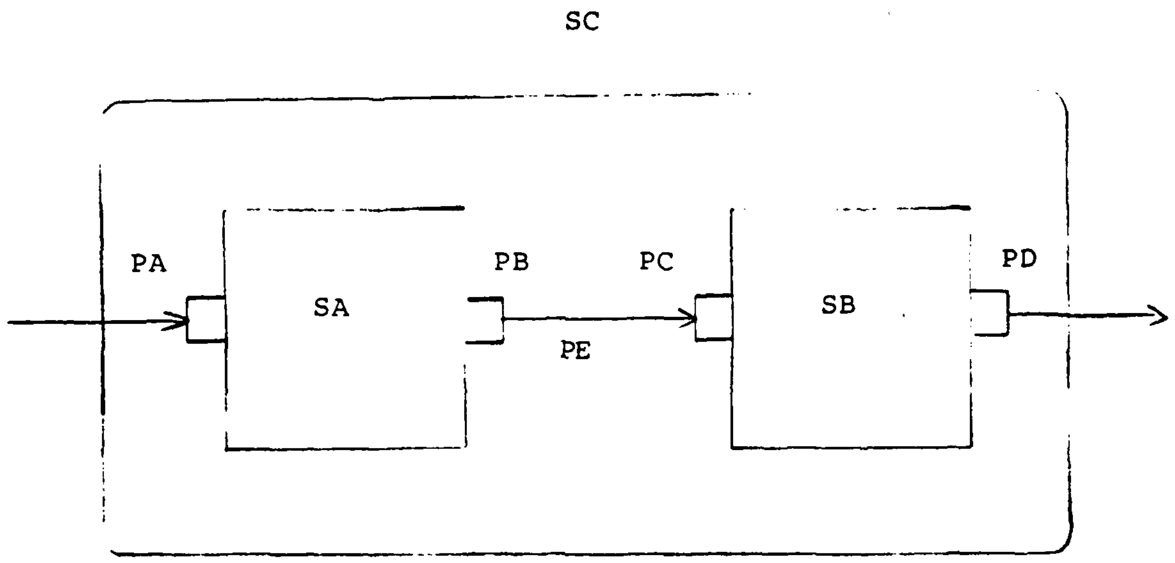


Figure 3.1. A Tandem Network

```
System  SC (PA : inport;
            PD : outport);

  Structure

    Subsystem SA (PA : inport;
                  PB : outport);

      Behavior
         RT (PA, PB);  [1]
      End behavior;

    End subsystem;

    Subsystem SB (PC : inport;
                  PD : outport);

      Behavior
         RT (PC, PD);
      End behavior;

    End system;

    Network
       connect (SA.PB, SB.PC) == SC.PE;
    End network;

    Interface
        SC.PA == SA.PA
        SC.PD == SB.PD
    End interface

  End structure

End system.
```

System SC is composed of two reliable transmission subsystems SA and SB. A system name followed by a port name denotes a port. A link connects outport PB of system SA to inport PC of SB and is renamed PE in system SC. The interface part says that system SC uses system SA's inport

[1] See Section 2.4.1. for the definition of RT.

PA and system SB's outport as interface ports. A tandem network with more than two subsystems can be specified in a similar way. The behavior of the whole system can be specified as follows:

```
System SC (PA : inport;
           PD : outport);

   Behavior
         RT (PA, PD)
   End behavior;

End system.
```

which means that a serial connection of two reliable transmission systems results in a single reliable transmission system.

3.3.2. The Verification of the Tandem Network

Once we have both the behavior and the structure specifications it is important to verify that the latter is a legal implementation of the former, i.e., the former is a logical consequence of the latter. Since the same specification language is used for both, the verification is a proof of a first order predict calculus theorem.

We first prove a theorem about a transitive relation. Since the event relations -> and => are transitive, the theorem holds for them.

Theorem 3.1.

If T is a transitivity relation,

(i.e., ∀ p, q, r T(p,q) ^ T(q,r) #> T(p,r))

then,

(∀ x∈ X ∃ y∈ Y T(x,y)) ^
(∀ y∈ Y ∃ z∈ Z T(y,z))
#> (∀ x∈ X ∃ z∈ Z T(x,z))

<u>Proof</u>

(1) ∀ p, q, r T(p,q) ^ T(q,r)#> T(p,r)
(2) ∀ x∈ X ∃ y∈ Y T(x,y)
(3) ∀ y∈ Y ∃ z∈ Z T(y,z)
(4) x∈ X

there exists y0∈ Y s.t.
(5) T(x,y0) .....(2), (4)

there exists z0∈ Z s.t.
(6) T(y0,z0) .....(3), (5)
(7) T(x,z0) .....(1), (5), (6)
Q. E. D.

In this proof, each predicate is preceded by a number to identify it, and is followed by a sequence of numbers to explain its derivation. A predicate without explanation is either an assumption (e.g., (1)) or a precondition (e.g., (2), (3) and (4)) of the predicate to be proved. Each derived predicate is explained by attaching to it the number of its preconditions. For example, (5) is derived from (2) and (4), in the proof above.

This theorem will be used frequently in the following verifications.

<u>Theorem 3.2.</u>

A tandem connection of two reliable transmission systems behaves as a single reliable transmission system.

Proof

```
(I) RT11(PA, PD):
    (* No loss of messages *)

    Obvious from RT11(PA, PB) of SA, RT11(PC, PD)
    of SB and Theorem 3.1.

(II) RT12(PA,PD):
     (* No self-existing messages *)

     Obvious from RT12(PA, PB) of SA, RT12(PC, PD)
     of SB and Theorem 3.1.

(III) RT13(PA,PD):
      (* No internally generated messages *)
      (* ∀ s< SYS(SC), d< PD                *)
      (*   s=> d ⇒ ∃ a< PA a=> s=> d        *)

      (1) s< SYS(SC), d< PD
      (2) s=> d
      (3) s< SYS(SA) v s< SYS(SB) v s< PE  .....(1)
      case
      (a): (4) s< SYS(SA)
           (5) s< ENV(SB) .....(4)

               there exists c0< PC s.t.
           (6) s=> c0=> d .....(2), (5), RT13(PC, PD)

               there exists a0< PA s.t.
           (7) a0=> s=> c0=> d
                          .....(4), (6), RT13(PA, PB)
      (b): (8) s< SYS(SB)

               there exists c0< PC s.t.
           (9) c0=> s=> d .....(2), (8), RT13(PC, PD)

                there exists a0< A s.t.
           (10) a0=> c0=> s=> d .....(9), RT12(PA, PB)
      (c): (11) s< PE

                there exists a0< PA s.t.
           (12) a0=> s=> d .....(2), RT12(PA, PB)

      Thus there exists a0< PA s.t.
      (13) a0=> s=> d ...(7), (10), (12)
```

```
(10) d1-> d2
          .....(5), (7), (9), RT14(PC, PD)
```

Similar proofs can be made for the cases of a2-> a1 and a1= a2.

```
(VI) RT21(PA, PD):
    (* No erroneous messages*)

    (1) a< PA
    (2) d< PD
    (3) a=> d
    (4) a< ENV(SB) .....(1)

        there exists c0< PC s.t.
    (5) a=> c0=> d .....(2), (3), (4), RT13(PC, PD)
    (6) a.msg= c0.msg .....(1), (5), RT21(PA, PB)
    (7) c0.msg= d.msg .....(2), (5), RT21(PC, PD)
    (8) a.msg= d.msg .....(6), (7)
    Q. E. D.
```

Although the theorems were proved in a semi-formal way, a more formal proof using the Robinson´s Refutation Graph can be found in Appendix B.

3.4. Example 2: an Alternate-Bit Protocol

An Alternate-Bit Protocol provides a reliable message transmission service over an unreliable transmission medium from a fixed sender to a fixed receiver. This protocol provides the service of a reliable transmission system and is specified as follows:

```
System  ABP( IP : inport;
             OP : outport);

   Behavior
         RT(IP, OP);
   End behavior;

End system.
```

```
        (* No externally generated messages *)
        (* ∀ x< ENV(SC), d< PD              *)
        (*   x=> d #> ǂ a< PA x=> a=> d     *)

        (1) x< ENV(SC), d< PD
        (2) x=> d
        (3) x< ENV(SB) .....(1)

            there exists c0< PC s.t.
        (4) x=> c0=> d .....(2), (3), RT13(PC, PD)
        (5) x< ENV(SA) .....(1)

            there exists a0< PA s.t.
        (6) x=> a0=> c0=> d
                           .....(4), (5), RT13(PA, PB)

(IV) RT14(PA, PD):
       (* No duplication of messages *)

       (1) d1, d2< PD
       (2) a< PA
       (3) a< ENV(SB) .....(2)
       (4) a=> d1

           there exists c1< PC s.t.
       (5) a=> c1=> d1 .....(1), (3), (4), RT13(PC, PD)
       (6) a=> d2

           there exists c2< PC s.t.
       (7) a=> c2=> d2  .....(1). (3), (6), RT13(PC, PD)
       (8) c1= c2 .....(5), (7), RT14(PA, PB)
       (9) d1= d2 .....(5), (7), (8), RT14(PC, PD)

(V) RT15(PA, PD):
      (* No out of order messages *)

      (1) d1, d2< PD
      (2) a1, a2< PA
      (3) a1, a2< ENV(SB) .....(2)
      (4) a1=> d1

          there exists c1< PC s.t.
      (5) a1=> c1=> d1 .....(1), (3), (4), RT13(PC, PD)
      (6) a2=> d2

          there exists c2< PC s.t.
      (7) a2=> c2=> d2 .....(1), (3), (6), RT13(PC, PD)
   case
      (i) (8) a1-> a2
          (9) c1-> c2
                  .....(5), (7), (8), RT14(PA, PB)
```

The underlying communication medium is unreliable and may lose, duplicate, or reorder messages; however, there is a non-zero probability of successful message transmission. For details of unreliable transmission systems see Chapter 2.

3.4.1. The Structure Specification of the Alternate-Bit Protocol

To guarantee that a message sent is eventually received, we take advantage of the "non-zero probability of message transmission" property of the unreliable medium. The Sender sends the same message repeatedly until it receives an acknowledgement from the Receiver, which acknowledges all messages received. To avoid duplication of messages, a serial number is attached to each message sent by the Sender and the Receiver accepts messages only if their serial numbers have never appeared before. To avoid reordering messages, a message is not sent until all the previous ones are acknowledged.

These ideas lead naturally to a protocol design (see Figure 3.2.) whose structure specification is depicted as follows:

```
(* Alternate-Bit Protocol *)

System   ABP (IP : inport;
              OP : outport);
```

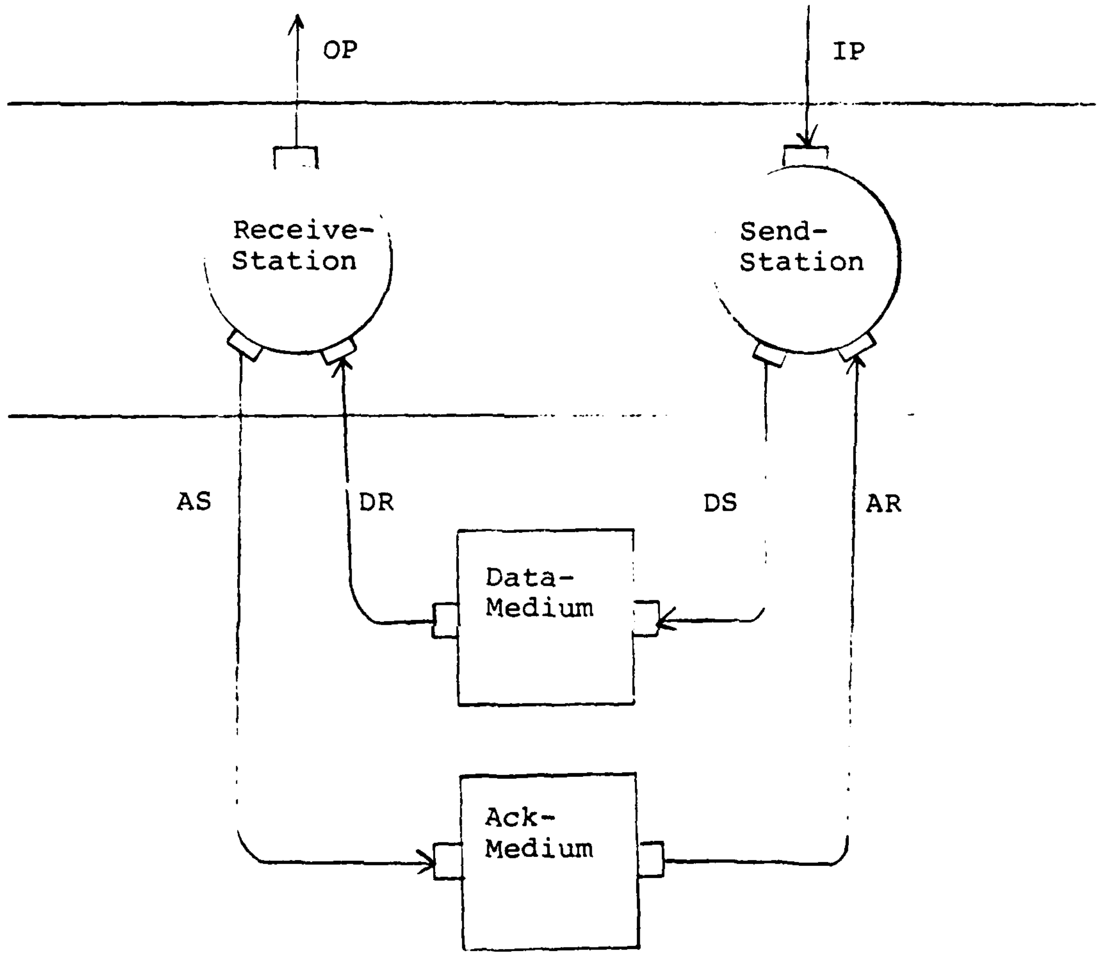


Figure 3.2. An Implementation Structure
of the Alternate-Bit Protocol

```
Messagetype
  IP.msg: elem;
  OP.msg: elem;
End messagetype;

Structure

   Subsystem SS (IP: inport;
                 AR: inport;
                 DS: outport);

      Messagetype
         IP.msg: elem;
         DS.msg: record
                    data: elem;
                    msgno: integer;
                 end;
         AR.msg: integer;
      End messagetype;

      Behavior

         (* SS1: Guaranteed message transmission: keep
            sending  the  same  message  until  ack-
            nowledged.  *)
              ∀ ip ∈ IP
                (∃ ds∈ DS ip=> ds) ^
                ((∃ ar∈ AR ar.msg= ord(ip)) v
                 (∀ d1∈ DS ip=> d1
                    #> ∃ d2∈ DS ip=> d2 ^ d1-> d2));

         (* SS2: No self-existing messages *)
               RT12(IP, DS);

         (* SS3: No internally or externally
            generated messages *)
               RT13(IP, DS);

         (* SS4: Sequence Control: do not accept a new
            input  message until all previous ones are
            acknowledged.  *)
              ∀ ip∈ IP
                ( ∀ k∈ N   [2]
                    k< ord(ip)
                    #> ∃ ar ∈ AR ar.msg = k ^
                                 ar -> ip);
```

[2] N refers to the set of natural numbers.

```
        (* SS5: Contents of messages: send out a mes-
           sage  together  with  a serial number as a
           unique id.  *)
              ∀ ip ⋖ IP, ds ⋖ DS
                ip => ds #> ds.data= ip.msg ^
                            ds.msgno= ord (ip);

End subsystem.


Subsystem  RS (DR : inport;
               AS : outport;
               OP : outport);

   Messagetype
      DR.msg: record
                 data: elem;
                 msgno: integer;
              end;
      AS.msg: integer;
      OP.msg: elem;
   End messagetype;

   Behavior

      (* The relation between DR and AS  is  essen-
         tially a transformer. *)
         (* Send an acknowledgement for every  mes-
            sage received back to the Sender.   *)
                RT11(DR,AS);
                RT12(DR,AS);
                RT13(DR,AS);
                RT14(DR,AS);
                RT15(DR,AS);

         (* RS21: send back the serial number as an
            acknowledgement of receipt.  *)
                ∀ dr ⋖ DR, as ⋖ AS
                  dr => as #> as.msg = dr.msgno;

      (* The relation between DR and OP  is  essen-
         tially  a  a filter which filters out mes-
         sages with the same serial number.  *)
         (* RS22: New messages will be accepted. *)
                ∀ dr ⋖ DR
                  ( ∀ dr´⋖ DR dr´-> dr
                      #> dr.msgno <> dr´.msgno)
                  #> ∃ op⋖ OP dr => op;

         (* RS23: Accept  only new  messages. *)
                ∀ dr ⋖ DR
```

```
                        ( ∃ op∢ OP dr => op)
                       #> ~( ∃ dr´∢ DR
                            dr´-> dr ^
                            dr´.msgno= dr.msgno);

          (* No internally or  externally  generated
             or duplicated or out-of-order messages.
             *)
                 RT13(DR,OP);
                 RT14(DR,OP);
                 RT15(DR,OP);

          (* RS24: accept only the data part of  the
             message. *)
                 ∀ dr ∢ DR, op ∢ OP
                   dr=> op #> op.msg = dr.data;

End subsystem;

(* Data-Transmission-Medium *)

Subsystem  DM (DS: inport;
               DR: outport);

   Messagetype

      DS.msg, DR.msg: record
                         data: elem;
                         msgno: integer;
                      end;

   End messagetype;

   Behavior
               UT(DS, DR);

   End behavior;

End subsystem;

(* Acknowledgement-Transmission-Medium *)

Subsystem  AM (AS : inport;
               AR : outport);

   Messagetype

      AS.msg, AR.msg: integer;

   End messagetype;
```

```
            Behavior

                        UT(AS, AR);

            End behavior;

        End subsystem;

        Network

            connect(SS.DS, DM.DS)== ABP.DS;
            connect(DM.DR, RS.DR)== ABP.DR;
            connect(RS.AS, AM.AS)== ABP.AS;
            connect(AM.AR, SS.AR)== ABP.AR;

        End network;

        Interface

            ABP.IP== SS.IP;
            ABP.OP== RS.OP;

        End interface;

    End system.
```

3.4.2. The Verification of the Alternate-Bit Protocol

We now prove that the structure specification of the Alternate-Bit Protocol satisfies its behavior specification. Since the DM (Data Transmission Medium) is an unreliable one, the SS (Send Station) has to send the message unboundedly to guarantee that at least one message will reach the RS (Receive Station). However, since the AM (Acknowledgement Transmission Medium) is also an unreliable one, it is possible that the acknowledgement may be lost. Fortunately, it can be proved that if the SS sends the same messages unboundedly, though DM is unreliable, an unbounded number of messages will arrive at RS.

Since RS acknowledges all messages received, it is guaranteed that at least one acknowledgement will arrive at SS.

<u>Theorem 3.3</u>.

If the underlying communication medium has a non-zero probability of message transmission, then if an unbounded number of identical messages are sent from A, an unbounded number of these messages will arrive at B. Formally:

```
UT(A, B) ^
(a0∈ A ^ ( ∀ aj∈ A aj.msg= a0.msg
                #> ∃ ak∈ A
                       ak.msg= a0.msg ^ aj-> ak))
#> (∃ al∈ A, bl∈ B
      al.msg= a0.msg ^ al=> bl) ^
   ( ∀ aj∈ A, bj∈ B
       aj.msg= a0.msg ^ aj=> bj
       #> ∃ ak∈ A, bk∈ B
           ak.msg= a0.msg ^ ak=> bk ^ bj-> bk)
```

<u>Proof</u>

Since an unbounded number of events f A happen after a0, one of them will enable an event of B. Similarly, for any event aj of A that enables an event bj of B, there is an event ak of A after aj that enables an event bk of B, since there are unbounded events happen after aj. If bj->bk, the proof is complete. Unfortunately, it is possible that bk->bj although aj->ak. This is due to the fact that the communication medium is unreliable that may reorder messages.

To solve this problem, the latest event x in A that enables an event of B at a time no later than bj, is chosen as the starting point of the unbounded sequence. This choice of event x guarantees that any event of A that occurs after x cannot enable an event of B that happens before bj. Since there are an unbounded number of events after x, one of them will enable an event of B at a time later than bj.

(1) $a0 \in A$

(2) $(\forall\, aj \in A\ aj.msg = a0.msg$
$\quad \Rightarrow \exists\, ak \in A\ ak.msg = a0.msg \wedge aj \rightarrow ak)$

there exist $al \in A$, $bl \in B$ s.t.

(3) $al \Rightarrow bl \wedge al.msg = a0.msg \wedge a0 \rightarrow al$
...(1), (2), NZ(A, B)

(4) $aj \in A$, $bj \in B$

(5) $aj.msg = a0.msg$

(6) $aj \Rightarrow bj$

Let S1 be the set of events in B that happen no later than bj and that are enabled by events in A having the same message contents as a0.msg:

(7) $S1 = \{y \in B \mid \exists\, x \in A\ x \Rightarrow y \wedge$
$\quad x.msg = a0.msg \wedge (y = bj \vee y \rightarrow bj)\}$

Let S2 be the set of events in A such that each of them enables an event in S1:

(8) $S2 = \{x \in A \mid \exists\, y \in S1\ x \Rightarrow y\}$

Select x from S2 such that x is the latest event in S2:

(9) $\forall\, x' \in S2\ x' = x \vee x' \rightarrow x$

there exist $ak \in A$, $bk \in B$ s.t.

(10) $ak.msg = a0.msg \wedge ak \Rightarrow bk \wedge x \rightarrow ak$
...(2), NZ(A,B)

Assume

(11) $bk = bj \vee bk \rightarrow bj$

Then

```
    (12) bk< S1, ak< S2 ...(7), (8), (10), (11)
    (13) ak= x v ak-> x ...(9), (12)
          (13) contradicts (10)
Thus
    (14) bj-> bk
  Q. E. D.
```

The verification that the structure specification of this Alternate-Bit Protocol satisfies its behavior specification is shown by the following sequence of lemmas and theorems.

Lemma 3.4.1.

If an event dr in DR is enabled by an event ip in IP then dr will carry ord(ip) as part of its message contents. Similar statements can be made for AS, AR, and OP. Formally:

```
 ∀ ip < IP, dr < DR, as < AS, ar < AR, op < OP
   (i) ip => dr #> dr.msgno = ord(ip)
   (ii) ip => as #> as.msg = ord(ip)
   (iii) ip => ar #> ar.msg = ord(ip)
   (iv) ip => op #> op.msg = ip.msg
```

Proof

```
(i)  (1) ip< IP
     (2) dr< DR
     (3) ip=> dr
     (4) ip< ENV(DM)   ..... (1)

         there exists ds0< DS s.t.
     (5) ip=> ds0=> dr .....(2), (3), (4), RT13(DS, DR)
     (6) ds0.msgno= ord(ip)   ^
         ds0.data= ip.msg .....(5),SS5
     (7) dr.msg= ds0.msg  .....(5), RT21(DS,DR)
     (8) dr.msgno= ord(ip) ...(6), (7)
```

Proofs of (ii), (iii) and (iv) are similar to (i).

Lemma 3.4.2.

Any event ds in DS that carries ord(ip) as part of its message contents is enabled by the ip in IP; similar statements can be made for DR, AS and AR. Formally:

```
 ∀ ip< IP, ds< DS, dr< DR, ar< AR, op< OP
  (i) ds.msgno= ord(ip) #> ip=> ds
  (ii) dr.msgno= ord(ip) #> ip=> dr
  (iii) as.msg= ord(ip) #> ip=> as
  (iv) ar.msg= ord(ip) #> ip=> ar
```

Proof

```
(i)  (1) ip< IP
     (2) ds< DS
     (3) ds.msgno= ord(ip)

         there exists ip0< IP s.t.
     (4) ip0=> ds  .....RT12(IP, DS)
     (5) ds.msgno= ord(ip0)  .....(4), SS5
     (6) ord(ip)= ord(ip0)  .....(3), (5)
     (7) ip= ip0  .....(6)
     Proofs of (ii), (iii) and (iv) are similar to (i).
```

Lemma 3.4.3.

Every event ip in IP will get back an acknowledgement from RS, carrying ord(ip) as message contents. Formally:

```
 ∀ ip< IP ∃ ar< AR
   ar.msg= ord(ip)
```

Proof

```
Prove by contradiction
   (1) ip< IP
Assume
   (2) ~(∃ ar< AR ar.msg= ord(ip))

       there exists ds0< DS s.t.
   (3) ip=> ds0 ...(1), SS1
   (4) ds0.msgno= ord(ip) ^ ds0.data= ip.msg
                                ....(3), SS5
   (5) ∀ dsi< DS ip=> dsi
```

```
            #> ∃ dsj< DS
               ip=> dsj ^ dsi-> dsj ...(2), SS1
      (6) ∀ dsi< DS dsi.msg= ds0.msg
            #> ∃ dsj< DS
                 dsj.msg= ds0.msg ^ dsi-> dsj
                   ....(5), Lemma 3.4.1., Lemma 3.4.2.
      (7)  ( ∃ dsi< DS, dri< DR
             dsi.msg= ds0.msg ^ dsi=> dri) ^
           ( ∀ dsi< DS, dri< DR
             dsi=> dri ^ dsi.msg= ds0.msg
               #> ∃ dsj< DS, drj< DR dsj.msg= ds0.msg ^
                   dsj=> drj ^ dsi-> dsj)
                                  ... (6), Theorem 3.3.
      (8)  ( ∃ asi< AS asi.msg= ord(ip)) ^
           ( ∀ asi< AS asi.msg= ord(ip)
               #> ∃ asj< AS
                   asj.msg= ord(ip) ^ asi-> asj)
            ...(4), (7), RS21, RT11(DR, AS), RT15(DR, AS)

           there exist ar0< AR, as0< AS s.t.
      (9)  as0=> ar0 ...(8), NZ(AS, AR)
      (10) ar0.msg= as0.msg= ord(ip)  ...(9)
             (10) contradict (2).
  Thus
      (11) ∃ ar< AR ar.msg= ord(ip)
               Q. E. D.
```

Lemma 3.4.4.

Every event dr in DR either causes an output event in OP or some dr´ that happens before dr, enabled by the same ip, has caused an output event in OP. Formally:

```
 ∀ dr< DR, ip< IP
   ip=> dr #> ∃ dr´< DR
                (dr´= dr v dr´-> dr) ^
                (∃ op< OP ip=> dr´=> op)
```

Proof

```
      (1) dr< DR
      (2) ip< IP
      (3) ip=> dr
      (4) dr.msgno= ord(ip)
                 .....(1), (2), (3), Lemma 3.4.1.

          Define D to be the set of events in DR that
          are enabled by ip:
      (5) D={ d< DR| ip=> d}
```

```
(6) D<> Ø .....dr< D, (1), (3)

    Select x from D such that x is the earliest
    event in D:
(7) ∀ d< D x=d v x-> d
(8) ∀ y< DR y-> x #> y.msgno<> x.msgno
                        ....(7), Lemma 3.4.2.
(9) ∃ op< OP x=> op .....RS22, (8)
      Q. E. D.
```

<u>Theorem</u> <u>3</u>.<u>5</u>.<u>1</u>

```
(* RT11: No loss of messages *)
   ∀ ip< IP ∃ op< OP
     ip=> op
```

<u>Proof</u>

```
(1) ip< IP

    there exists  ar0< AR s.t.
(2) ar0.msg= ord(ip) .....(1), Lemma 3.4.3.
(3) ip=> ar0 .....(2), Lemma 3.4.2.

    there exists dr0< DR s.t.
(4) ip=> dr0=> ar0
         .....(3), RT13's

    there exist dr'< DR, op0< OP s.t.
(5) ip=> dr' => op0 .....(4), Lemma 3.4.4.
```

<u>Theorem</u> <u>3</u>.<u>5</u>.<u>2</u>

```
(* RT12: No self-existing messages *)
   ∀ op< OP ∃ ip< IP
     ip=> op
```

<u>Proof</u>

Trivial from RT12(DR, OP), RT12(DS, DR), RT12(IP, DS) and Theorem 3.1.

<u>Theorem</u> <u>3</u>.<u>5</u>.<u>3</u>.

```
 (* RT13: No internally or  externally  generated  mes-
sages *)
     ∀ op< OP, e1< SYS(ABP), e2< ENV(ABP)
       (e1=> op #> ∃ ip< IP ip=> e1=> op) ^
       (e2=> op #> ∃ ip< IP e2=> ip=> op)
```

<u>Proof</u>

```
(1) op< OP
(2) e1< SYS(ABP)
(3) e1=> op
(4) e1< (SYS(SS) v SYS(DM) v SYS(RS) v
         SYS(AM) v DS v DR v AS v AR) .....(2)

    in all cases there exists ip0< IP s.t.
(5) ip0=> e1  .....(1), (3), RT13's
(6) ip0=> e1=> op .....(3), (5)
(7) e2< ENV(ABP)
(8) e2=> op
(9) e2< (ENV(SS) ^ ENV(DM) ^ ENV(RS))  .....(7)

     there exist ip0< IP, ds0< DS, dr0< DR s.t.
(10) e2=> ip0=> ds0=> dr0=> op
                    .....(8), (9), RT13's
```

<u>Theorem</u> <u>3</u>.<u>5</u>.<u>4</u>.

```
(* RT14: No duplicated messages *)
   ∀ ip< IP, op1, op2< OP
     (ip=> op1) ^ (ip=> op2) #> op1= op2
```

<u>Proof</u>

```
(1) ip< Ip
(2) op1, op2< OP
(3) ip=> op1

    there exist dr1< DR, ds1< DS s.t.
(4) ip=> ds1=> dr1=> op1  .....(1), (2), (3), RT13's
(5) dr1.msgno= ord(ip)  .....(4), Lemma 3.4.1.
(6) ip=> op2

    there exist dr2< DR, ds2< DS s.t.
(7) ip=> ds2=> dr2=> op2  .....(6), RT13's
(8) dr2.msgno= ord(ip) .....(7), Lemma 3.4.1.
(9) dr1.msgno= dr2.msgno ...(5), (8)
(10) dr1= dr2 ...(4), (7), (9), RS23
(11) op1= op2  .....(4), (7), (10), RT14(DR, OP)
```

<u>Theorem</u> <u>3</u>.<u>5</u>.<u>5</u>.

```
(* RT15: No out-of-order messages *)
   ∀ ip1, ip2< IP, op1, op2< OP
     (ip1=> op1) ^ (ip2=> op2)
     #> (ip1= ip2 ^ op1= op2) v
        (ip1-> ip2 ^ op1-> op2) v
        (ip2-> ip1 ^ op2-> op1)
```

<u>Proof</u>

```
(1) ip1, ip2< IP
```

```
    (2) op1, op2∈ OP
    (3) ip1=> op1

        there exist dr1∈ DR, ds1∈ DS s.t.
    (4) ip1=> ds1=> dr1=> op1  ...(1), (2), (3), RT13's
    (5) ∀ d∈ DR d<>dr1 ^ ip1=> d #> dr1-> d
                            ....(4), RS23, Lemma 3.4.2.
  case
    (i) (6) ip1-> ip2

            there exist ar1∈ AR, d∈ DR s.t.
        (7) (ip1=> d=> ar1) ^ (ar1-> ip2) .....(6), SS4
        (8) ip2=> op2

            there exists dr2∈ DR s.t.
        (9) ip2=> dr2=> op2 .....(8), RT13's
        (10) dr1-> (or =) d-> ar1-> ip2-> dr2
                            .....(5), (7), (9)
        (11) op1-> op2 .....(4), (9), (10), RT15(DR, OP)
    Similar proofs can be made for the cases of
    (ip1= ip2) and (ip2-> ip1).
```

Theorem 3.5.6.

```
  (* RT21: No erroneous messages *)
     ∀ ip∈ IP, op∈ OP
       ip=> op #> ip.msg= op.msg
```

Proof

This theorem has been proved in Lemma 3.4.1.

3.5. Example 3: A Distributed Prime Number Generator

A Prime Number Generator PNG consists of one input port A from the environment and an output port B to the environment. PNG receives a bounded sequence of integers greater than or equal to two in ascending order; PNG outputs the sequence of primes from the input sequence.

The behavior of the system PNG is, in terms of EBS, nothing more than a filter which filters out the non-prime numbers, and is specified formally as follows:

```
   System PNG (A: inport; B: outport);

      Behavior

      (* Output a number to B if and only if it  is  prime
         *)

         (* PN11: *)
            ∀ a< A
              ~(∃ a'< A a'-> a ^ a'.msg|a.msg)
              #> ∃ b< B a=> b;                    [3]

         (* PN12: *)
            ∀ b< B ∃ a< A
              a=> b ^ ~(∃ a'< A a'-> a ^ a'.msg|a.msg);

      (* PN13: No internally or externally generated  mes-
         sages; PN14: No duplicated messages; PN15: No out
         of order messages; PN21: No erroneous messages *)
            RT13(A, B); RT14(A, B);
            RT15(A, B); RT21(A, B);

      End behavior

   End system.
```

A distributed design [HOA78] to generate prime numbers using the "sieve of Eratosthenes" method, is depicted in Figure 3.3.

PNG consists of two types of processes: Sieves and a Printer. In order to simplify description, we assume there are infinite number of sieve processes, denoted by Sieve[1], Sieve[2], ..., Sieve[i], ... Each Sieve[i] has one inport P[i] by which it receives input from Sieve[i-1] (or the environment, if i= 1). Ports P[i], i=2, 3, ... are

[3] a|b means that a divides b.

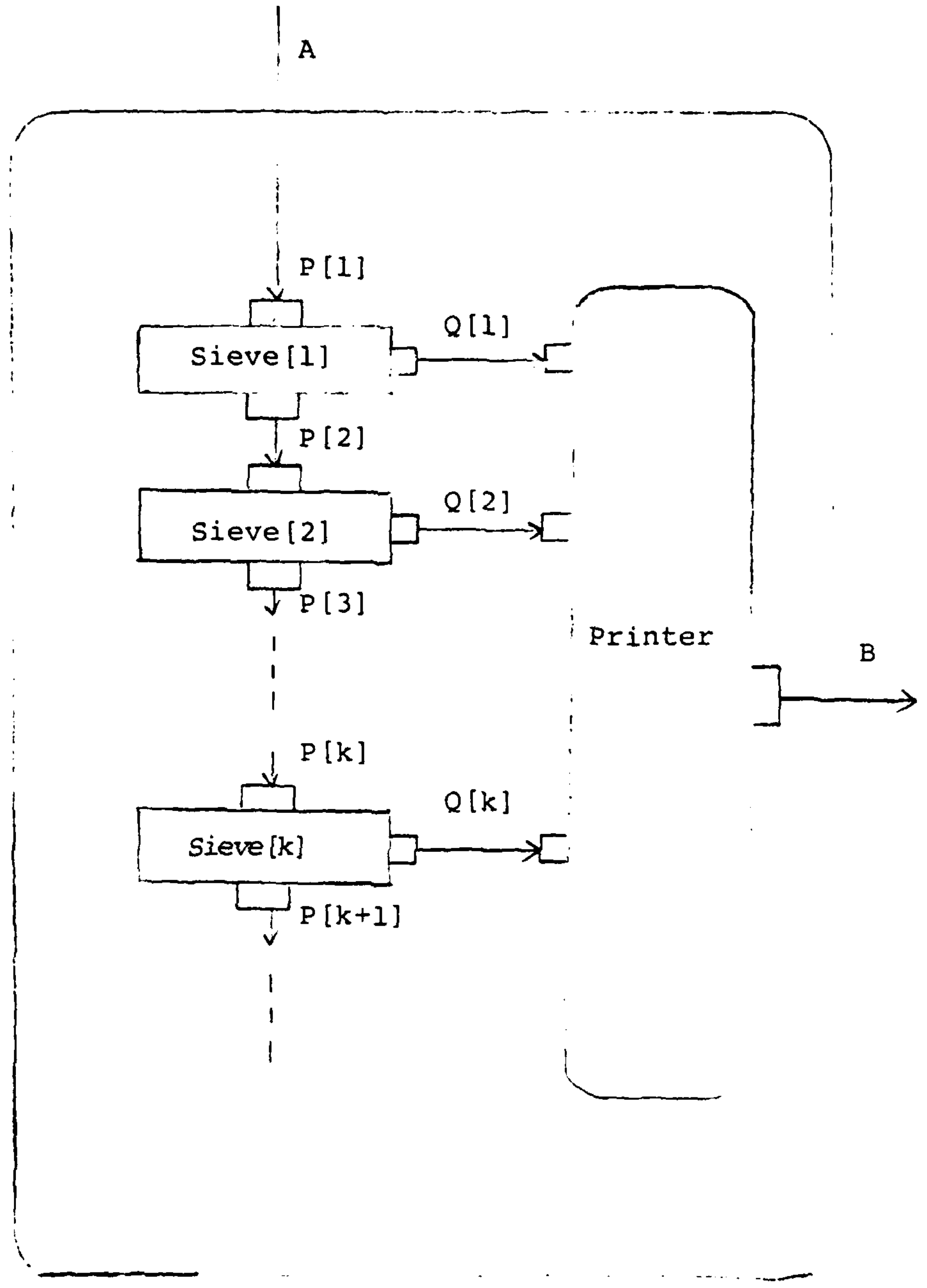


Figure 3.3. A Distributed Prime Number Generator

internal to PNG, but P[1] is an inport directed toward PNG. Sieve[i] has two outports P[i+1] and Q[i]. The latter is directed toward the Printer process. The Printer process has one outport B, which is also the outport of PNG.

3.5.1. An EBS Description of Sieve[i]

The first message p received by Sieve[i] is sent to the Printer process. Every subsequent message x received by Sieve[i] is checked to see if it is a multiple of p; if x is a multiple of p it is discarded; otherwise, it is sent on to Sieve[i+1] through port P[i+1]. An EBS specification of a Sieve follows. (See also Figure 3.4.)

```
System SV (P: inport; Q: outport; R: outport);

   Behavior

   (* Part I. Relationship between P and Q:  A  message
      is  sent to Q if and only if it is the first mes-
      sage in P *)

      (* SQ11: *)
         ∀ p< P
           ord(p)= 1 #> (∃ q< Q p=> q);

      (* SQ12: *)
         ∀ q< Q ∃ p< P
           p=> q ^ ord(p)= 1;

   (* SQ13, SQ14, SQ15, SQ21 *)
      RT13(P, Q); RT14(P, Q); RT15(P, Q); RT21(P, Q);

   (* Part II. Relationship between P and R: A  message
      is  sent to R if and only if it is not a multiple
      of the first message in P *)

      (* SR11: *)
         ∀ p1, p2 < P
           ord(p1)= 1 ^ ord(p2) > 1 ^ ~(p1.msg|p2.msg)
           #> ∃ r< R p2=> r;
```

```
    (* SR12: *)
       ∀ r∢ R ∤ p2∢ P
         p2=> r ^ ord(p2) > 1 ^
         ~(∤ p1∢ P ord(p1)= 1 ^ p1.msg|p2.msg);

  (* SR13, SR14, SR15, SR21 *)
     RT13(P, R); RT14(P, R); RT15(P, R); RT21(P, R);

  (* QR11: Messages sent to Q  and  R  maintain  their
     orders *)
     ∀ p1, p2∢ P, q∢ Q, r∢ R
       p1=> q ^ p2=> r
       #> (p1-> p2 ^ q-> r) v
          (p2-> p1 ^ r-> q);

 End behavior;

End system.
```

3.5.2. An EBS Description of the Printer Process

The Printer process waits to receive input along all input ports. Upon receiving an input message, it sends the received value to the outport. The printing service is on a first-come-first-serve basis.

In terms of EBS, the behavior of the Printer process is nothing more than a multiplexor, and is specified as follows:

```
System PRT ( Q[k]: inport; (* k∢ N *)
             B: outport);

  Behavior

  (* PR11: No loss of messages *)
     ∀ k∢ N, q∢ Q[k] ∤ b∢ B
       q=> b;

  (* PR12: No self-existing messages *)
     ∀ b∢ B ∤ k∢ N, q∢ Q[k]
       q=> b;
```

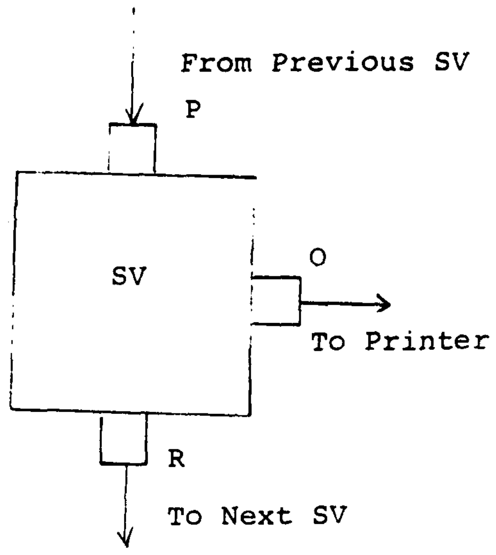


Figure 3.4. A Sieve Process

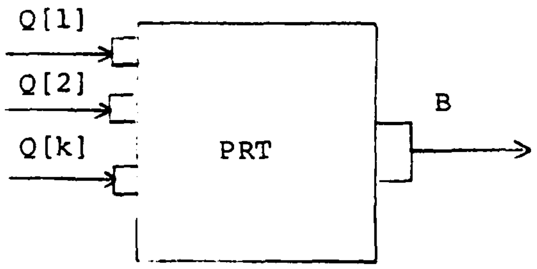


Figure 3.5. A Printer Process

```
(* PR13: No internally or externally generated  mes-
   sages *)
   ∀ b∈ B, s∈ SYS, e∈ ENV
     (s=> b #> ∃ k∈ N, q∈ Q[k] q=> s=> b) ^
     (e=> b #> ∃ k∈ N, q∈ Q[k] e=> q=> b);

(* PR14: No duplicated messages *)
   ∀ k∈ N, q∈ Q[k], b1, b2∈ B
     q=> b1 ^ q=> b2 #> b1= b2;

(* PR15: No out of order messages *)
   ∀ i, j∈ N, q1∈ Q[i], q2∈ Q[j], b1, b2∈ B
     (q1=> b1 ^ q2=> b2 #> (q1= q2 ^ b1= b2) v
                           (q1-> q2 ^ b1-> b2) v
                           (q2-> q1 ^ b2-> b1);

   (* PR21: No erroneous messages *)
   ∀ k∈ N, q∈ Q[k], b∈ B
     q=> b #> b.msg= q.msg;

End behavior;

End system.
```

3.5.3. The Structure Specification of PNG

The whole system structure can be specified as follows:

```
System PNG (A: inport; B: outport);

   Structure

      Subsystem  Sieve[1] ( P[1]: inport;
                            Q[1]: outport;
                            P[2]: outport);

         Behavior

            SV (P[1], Q[1], P[2]);

            (* Input numbers are in ascending order *)
               ∀ p, p´∈ P[1]
                 p-> p´ #> p.msg< p´.msg;

            (* Input is bounded by maxinteger *)
               ~(∃ p∈ P[1] ord(p) > maxinteger);

          End behavior;
          End system;

          Subsystem Sieve[k] ( P[k]: inport;
                               Q[k]: outport;
                               P[k+1]: outport);

             Behavior

                SV(P[k], Q[k], P[k+1]);

             End behavior;

          End system;

          (* k< N, k<>1 *)

          Subsystem Printer (Q[k]: inport (* k< N *);
                             B: outport);

             Behavior

                SV (Q[k]: inport (* k< N *);
                    B: outport);

             End behavior;

          End system;

          Network

             connect (Sieve[k].P[k+1],
                      Sieve[k+1].P[k+1])== PNG.P[k+1]);
             connect (Sieve[k].Q[k],
                      Printer.Q[k])== PNG.Q[k];

             (* k< N *)

          End network;

          Interface

             Sieve[1].P[1]== PNG.A;
             Printer.B== PNG.B;

          End interface;

       End structure;

    End system.
```

3.5.4. The Verification of the Prime Number Generator

We now prove that the distributed prime number generator satisfies the behavior specification. Since a message is sent to the Printer process if and only if it first arrives a Sieve, a critical step in the verification is to prove that a number will first arrive at a Sieve if and only if it is a prime. This is proved by the following lemmas.

Since it is easily shown that y.msg=x.msg if and only if the event x of A enables the event y of P[k], Q[k] or B, the predicates x=>y and y.msg=x.msg are used interchangeably in the proofs.

Lemma 3.6.1.

The message sequence in every port is in ascending order. Formally:

```
∀ i∈ N, p1, p2∈ P[i]
  p1-> p2 #> p1.msg < p2.msg
```

Proof

```
(* By induction on port id numbers *)

  The Induction Hypothesis
  H(i) = p1, p2∈ P[i]
         p1-> p2 #> p1.msg< p2.msg

(I) H(1) is trivial;
(II) Assume H(k) is true, then
     (1) p1, p2∈ P[k+1], p1-> p2

         there exist p1´, p2´∈ P[k] s.t.
     (2) p1´=> p1 ^ p2´=> p2 ...SR12
     (3) p1.msg= p1´.msg ^ p2.msg= p2´.msg
                                 .... SR21
```

```
        (4) pl´-> p2´ ...(1), (2), SR15
        (5) pl´.msg < p2´.msg .... (4), Hypothesis
        (6) pl.msg< p2.msg ...(3), (5)
        Thus H(k+1)
    Q. E. D.
```

<u>Lemma</u> <u>3</u>.<u>6</u>.<u>2</u>.

If a number x appears at port P[i] then no number sent to the Printer process from any previous port divides x. Formally:

```
∀ i◂ N, x◂ P[i]
  ~(∃ j◂ N, a◂ P[j]
      j< i ^ ord(a)= 1 ^ a.msg|x.msg)
```

<u>Proof</u>

```
    (* By induction on port id numbers *)

       The Induction Hypothesis
       H(i) = ∀ x◂ P[i]
                  ~(∃ j◂ N, a◂ P[j]
                      j< i ^ ord(a)= 1 ^ a.msg|x.msg)

    (I) H(1) is trivial;
    (II) Assume H(k) is true, then
         for all y◂ P[k+1]

         there exists x◂ P[k] s.t.
         (1) x=> y ^ x.msg= y.msg ...SR12, SR21
         (2) ~(∃ x´◂ P[k]
                   ord(x´)= 1 ^ x´.msg|x.msg) ...SR12
         (3) ~(∃ j◂ N, a◂ P[j]
                   j< k ^ ord(a)= 1 ^ a.msg|x.msg)
                                         ...Hypothesis
         (4) ~(∃ j◂ N, a◂ P[k]
                   j≤ k ^ ord(a)= 1 ^ a.msg|y.msg)
                                       ... (1), (2), (3)
         Thus H(k+1) is true
     Q. E. D.
```

<u>Lemma</u> <u>3</u>.<u>6</u>.<u>3</u>.

If a number x first arrives at port P[i] then every number that is less than x is divisible by some number sent to the Printer process from a previous port.

Formally:

```
∀ i◂ N, x◂ P[i]
  ord(x)= 1 #> (∀ a◂ P[1]
                a.msg< x.msg
                #> (∃ j◂ N, b◂ P[j]
                      j< i ^ ord(b)= 1 ^
                      b.msg|a.msg))
```

<u>Proof</u>

```
(* By contradiction *)
   (1) i◂ N, x◂ P[i], ord(x)= 1
   (2) a◂ P[1], a.msg< x.msg
   Assume
   (3) ~(∃ j◂ N, b◂ P[j]
            j< i ^ ord(b)= 1 ^ b.msg|a.msg)

       there exists y◂ P[i] s.t.
   (4) a=> y ....(3), SR12
   (5) y.msg= a.msg< x.msg ...(4), SR21
   (6) y-> x .... (5), Lemma 3.6.1.
       (6) contradicts (1), (4)
         Q. E. D.
```

<u>Lemma 3.6.4.</u>

If a number x first arrives at port P[i] then no previous number divides x. Formally:

```
∀ i◂ N, x◂ P[i]
  ord(x)= 1 #> ~(∃ a◂ P[1]
                  a.msg< x.msg ^ a.msg|x.msg)
```

<u>Proof</u>

```
(* By contradiction *)

   (1) x◂ P[i] ^ ord(x)= 1

       Assume there exists a◂ P[1] s.t.
   (2) a.msg< x.msg ^ a.msg|x.msg

       there exist j< i, y◂ P[j] s.t.
   (3) ord(y)= 1 ^ y.msg|a.msg ...(2), Lemma 3.6.3.
   (4) y.msg|x.msg ...(2), (3)
       (1), (4) contradict Lemma 3.6.2.
       Q. E. D.
```

<u>Lemma 3.6.5.</u>

The length of messages strictly decreases as the port id number increases. Formally:

```
∀ i< N, a< P[i],  b< P[i+1]
  a=> b #> ord(a) > ord(b)
```

<u>Proof</u>

```
(* By induction on the ordinal number of b *)

   Define a function f: P[i+1]-> P[i]
   such that f(b)= a iff a=> b
   This function exists because of SR12 and SR15.
   The Induction Hypothesis
     H(n)= b< P[i+1], a< P[i]
           ord(b)= n ^ f(b)= a #> ord(b) < ord(a)
     (I) H(1) is true because of SR12.
     (II) Assume H(k) is true then
          if b'< P[i+1], a'< P[i]
          ord(b')= k+1 ^ f(b')= a'
          (1) a-> a' ...SR15
          (2) ord(a') > ord(a) > ord(b) = ord(b') -1
                                  ...(1), Hypothesis
          (3) ord(a') > ord(b') ...(2)
        Q. E. D.
```

<u>Lemma 3.6.6.</u>

Every message terminates at some port; in particular the message can not go beyond the port whose id number equals to the ordinal number of the message. Formally:

```
∀ a< P[1]
  ~(∃ i< N, x< P[i]
      i> ord(a) ^ a=> x)
```

<u>Proof</u>

```
(* By contradiction *)

   Let a< P[1]

   Assume there exist i< N, x< P[i] s.t.
   (1) i> ord(a) ^ a=> x

       there exist a[k]< P[k]
       for all k from 1 to i s.t.
   (2) (a=) a[1]=> a[2]=> ...=> a[i] (= x)
```

```
                                          .........SR13
        (3) ord(x) < ord(a) - i ....... Lemma 3.6.5.
        (4) ord(x) < 0 ......(1), (3)
              (4) contradicts the fundamental
                  property of ordinal numbers
     Q. E. D.
```

<u>Lemma 3.6.7.</u>

A prime number first arrives at some port. Formally:

```
     ∀ a∈ P[1]
       ~(∃ a'∈ P[1] a'-> a ^ a'.msg|a.msg)
       => ∃ k∈ N, x∈ P[k]
            (a= x v a=> x) ^ ord(x)= 1
```

<u>Proof</u>

```
     For all a∈ P[1],
     from Lemma 3.6.6. and SR21,
     there exists i∈ N  s.t.
                  ~(∃ y∈ P[i] a=y v a=> y)
     Let j be the lower bound of such id numbers,
     (it is obvious that j > 1)
     then there exists x∈ P[j-1] s.t.
          (1) a=x v a=> x
          (2) ~(∃ y∈ P[j] a=y v a=> y)
          (3) ~(∃ a'∈ P[1] a'-> a ^ a'.msg|a.msg)
          (4) ~(∃ x'∈ P[j] x'-> x ^ x'.msg|x.msg)
                    ....(1), (3), SR11, SR15, SR21

          Assume
          (5) ord(x)> 1

              there exists y∈ P[j] s.t.
          (6) x=> y   ...(4), (5), SR12
              (6) contradicts (2)
          (7) ord(x)= 1
              Q. E. D.
```

<u>Theorem 3.7.</u>

The distributed "sieve of Eratosthenes" is a correct prime number generator.

<u>Proof</u>

By the sequence of theorems 3.7.1. to 3.7.6.

<u>Theorem 3.7.1.</u>

```
    ∀ a∈ A
      ~(∃ a´∈ A a´-> a ^ a´.msg|a.msg)
      ⇔ ∃ b∈ B a=> b
```

<u>Proof</u>

```
        (1) a∈ A ^ ~(∃ a´∈ A a´-> a ^ a´.msg|a.msg)

            there exist k∈ N, x∈ P[k] s.t.
        (2) (a=x v a=> x) ^ ord(x)= 1 ....(1), Lemma 3.6.7.

            there exists y∈ Q[k] s.t.
        (3) x=> y ....(2), SQ11

            there exists b∈ B s.t.
        (4) y=> b    .... PR11
        (5) a=> b    ....(2), (3), (4)
```

<u>Theorem 3.7.2.</u>

```
    ∀ b∈ B ∃ a∈ A
      a=> b ^ ~(∃ a´∈ A a´-> a ^ a´.msg|a.msg)
```

<u>Proof</u>

```
        (1) b∈ B

            there exists k∈ N, y∈ Q[k] s.t.
        (2) y=> b ....(1), PR12

            there exists x∈ P[k] s.t.
        (3) x=> y ^ ord(x)= 1 .... SQ12

            there exists a∈ A s.t.
        (4) a= x v a=> x ....SR12
        (5) ~(∃ a´∈ A a´-> a ^ a´.msg|a.msg)
                             .... (3), (4), Lemma 3.6.4.
        (6) a=> b ....(2), (3), (4)
```

<u>Theorem 3.7.3.</u>

```
    ∀ b∈ B, s∈ SYS, e∈ ENV
      (s=> b ⇔ ∃ a∈ A a=> s=> b) ^
      (e=> b ⇔ ∃ a∈ A e=> a=> b)
```

<u>Proof</u>

Trivial

<u>Theorem 3.7.4.</u>

```
    ∀ a< A, bl, b2< B
      a=> bl ^ a=> b2 #> bl= b2
```

<u>Proof</u>

```
        (1) a< A, bl, b2< B, a=> bl, a=> b2

            there exist i, j< N, pl< P[i], p2< P[j]
                                 ql< Q[i], q2< Q[j] s.t.
        (2) (a= pl v a=> pl) ^ (pl=> ql=> b)
        (3) (a= p2 v a=> p2) ^ (p2=> q2=> b)
                              ...(1), SR13, SQ13, PR13

            assume i< j then there exists pl'< P[i] s.t.
        (4) (a= pl' v a=> pl') ^ (pl'=> p2)
                              ... (3), SR13
        (5) pl= pl' ...(2), (4), SR14
            (2), (4), (5) conflict SR11 and SQ11

            Similar proof can show the
            infeasibility of j< i; Thus
        (6) i= j
        (7) pl= p2, ql= q2, bl= b2 ...SR14, SQ14, PR14
```

<u>Theorem 3.7.5.</u>

```
    ∀ al, a2< A, bl, b2< B
      al=> bl ^ a2=> b2 #> (al= a2 ^ bl= b2) v
                           (al-> a2 ^ bl-> b2) v
                           (a2-> al ^ b2-> bl)
```

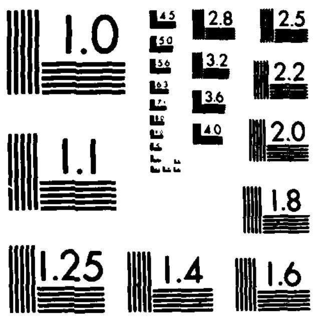
<u>Proof</u>

```
        (1) al, a2< A, bl, b2< B, al=> bl, a2=> b2

            there exist i, j< N, ql< Q[i], q2< Q[j] s.t.
        (2) al=> ql=> bl ^ a2=> q2=> b2
            case (i) i< j

                    there exist p2< P[i+1],
                                pl, pl'< P[i] s.t.
                    (3) (a2= pl' v a2=> pl') ^
                        (pl'=> p2=> q2) ^
                        (al= pl v al=> pl) ^
                        (pl=> ql)       ...SR12, SQ12
                    (4) pl-> pl'  ...(3), SR12, SQ12
                    (5) ql-> p2-> q2 ...(3), QR15
                    (6) al-> a2 ^ bl-> b2
                          ...(3), (4), (5), SR15, SQ15, PR15
                (ii) i= j
                    (7) al= a2 ^ bl= b2
```

AD-A128 629 EVENT-BASED SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE B CHEN 1982 AFOSR-TR-83-0388

2/3

UNCLASSIFIED F49620-80-C-0001 F/G 9/2 NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

(iii) i> j
(8) a2-> a1 ^ b2-> b1
...similar to case (i)

<u>Theorem 3.7.6.</u>

∀ a< A, b< B
a=> b #> a.msg= b.msg

<u>Proof</u>

Trivial

CHAPTER 4

TRANSACTION BASED SYSTEM SPECIFICATIONS AND VERIFICATIONS

4.1. Transactions

In database systems, users access shared data under the assumption that the data satisfies certain consistency assertions. For example in a FIFO (first-in-first-out) message buffer, consistency assertions require that

(1) The length of the output sequence is less than or equals to the length of the input sequence;

(2) If a message x enters the queue before a message y then x should also leave the queue before y; and

(3) The output sequence is a subsequence of the input sequence.

The system state continuously changes due to events caused by user commands. One might think that consistency constraints could be enforced at the event level; but this is not true. One may need to temporarily violate the consistency of the system state while modifying it. For example, in transferring money from one bank account to another there will be an instant during which one account has been debited and the other not yet credited. This violates a constraint that the number of dollars in the system is constant.

For this reason, the events in response to a user command are grouped into sequences called _transactions_, which are units of consistency. In general, consistency assertions cannot be enforced during the time between the beginning and the end of a transaction. In this chapter, it is assumed that each transaction, when executed alone, transforms the system from a consistent state into a new consistent state; that is, transactions preserve consistency.

We are interested in concurrent transactions in which events are interleaved yet each transaction maintains a consistent view of the system state. Not all consistent sequences for a set of transactions give exactly the same state (i.e., consistency is a weaker property than determinacy). For example, in an airlines reservation system if a set of transactions each requests a seat on a particular flight, then each consistent sequence will have the property that no seat is sold twice and no request is denied if there is a free seat, but two distinct consistent sequences may differ in the details of the seat assignments.

In such a transaction-oriented environment, each transaction must employ some protocol to insure that data which is temporarily inconsistent is not accessed. Mechanisms have been built on low level protocols like P, and V Semaphores [DIJ68] or some high level protocols like Monitors [HOA74] or Rendezvous [ICH79]. However, from the user´s viewpoint, the only interesting thing is the _service_ the protocol provides but not _how_ the protocol _mechanism_ manages to provide the service.

From the service specification viewpoint, several transaction synchronization description methods are compared. We show that when compared with others, our Transaction-Based Specification Language (TBS), which is based on the event model, is suitable for the service specification of these transaction-oriented systems.

The description methods surveyed are:

(1) Dijkstra´s P, V Semaphores

(2) Hoare and Brinch Hansen´s Monitors

(3) Mao and Yeh´s Communication Ports or ADA´s Rendezvous

(4) Campbell and Habermann´s Path Expressions

(5) Transaction-Based Specification Language

The methods are compared by examining their solutions to two typical examples: a bounded-buffer and a reader-writer database. The survey is, by no means, complete;

however, it covers most of the major methods.

4.2. Examples: a Bounded-Buffer and a Reader-Writer Database

In this section, we introduce two famous examples, namely: a bounded-buffer and a reader-writer database.

4.2.1. Example 1: a Bounded-Buffer

A bounded-buffer is a data structure containing a finite number of slots for storing information in transit from a producer to a consumer. The producer passes to the buffer monitor a portion of information to be buffered. The buffer monitor stores the information if there is space available; otherwise the producer waits until space becomes available. The consumer process requests information from the buffer, and awaits if this information is not available. The buffer monitor dispatches portions to the consumer, waiting if there is none available.

4.2.2. Example 2: a Reader-Writer Database

Another important scheduling problem is the coordination of several users on a common data object with unspecified internal structure, but whose contents may be inspected or changed. This is the famous reader-writer problem that was first published and solved in [COU71]. There are two kinds of users: "writers" modify the object and must thus have exclusive access, whereas "readers" only inspect the object and therefore may share it with other readers. This problem illustrates two different kinds of synchronization requirements at the same time, namely, mutual exclusion and concurrency control.

4.3. Dijkstra's P and V Semaphores

One of the oldest and best known primitive sets is the boolean semaphore described by Dijkstra [DIJ68]. This consists of two operators P and V acting on a semaphore S which takes two values, busy or free (or equivalently true or false). The behavior of the operation is:

P(S) If S is busy the process is suspended until S becomes free. If S is free then it is set busy and the process proceeds.

V(S) S is set free. If there are processes held on a P(S) operation then one of them is allowed to proceed.

Semaphores can be used to protect data by surrounding code which accesses the data by matched calls.

```
Process A                          Process B
P(S)        ...lock data.....      P(S)
 .                                  .
 .          ...access data...       .
 .                                  .
V(S)        ...unlock data...      V(S)
```

Semaphores can also be used to coordinate processes. One process waits by calling P, the other signals by calling V.

```
Process A                      Process B
P(S)      ...wait for B        V(S)      ...signal A
```

Semaphores can therefore be used both for data protection and process cooperation. They also have the merit of being both simple to describe and easy to understand. Solutions to the bounded-buffer and the reader-writer database using Semaphores can be found in [HAB72] and [COU71] respectly.

However, their disadvantages are also well-known. The most important drawback is that semaphores are visible to user processes which need not access them. Typical problems derived from this drawback are:

* A call of P may be skipped, giving access to unprotected data.
* A call of V may be skipped, leaving the semaphore busy so that the system deadlocks.
* One can forget to use them.

The Monitor concept introduced by Brinch Hansen [BRI74] and Hoare [HOA74] tried to remedy this drawback.

4.4. Hoare and Brinch Hansen's Monitors

A monitor consists of a collection of protected data and associated procedures. The data is protected by the monitor, and can only be accessed within the body of a monitor procedure. The procedures are accessible to all processes, in that any process may at any time attempt to call such a procedure. However, the monitor ensures that at most one process is executing a monitor procedure at a time by blocking any other process that calls an entry procedure. Processes communicate with one another by condition variables (usually implemented as queues) with two associated operations, signal and wait. The solutions to the bounded-buffer and the reader-writer database by using Monitors are depicted in Program 4.1. and Program 4.2. respectively.

Program 4.1. Bounded-Buffer in Monitor

```
Monitor bounded-buffer;

   Var
      buffer: array [0..bufsize- 1] of elem;
      inp, outp: integer;
      nonempty, nonfull: condition;

   Procedure produce (x: elem);
      Begin
        if inp-outp= bufsize
           then nonfull.wait;
        buffer[inp Mod bufsize]:= x;
        inp:= inp+ 1;
        nonempty.signal
      End;

    Procedure consume (Result x: elem);
      Begin
        if inp= outp
           then nonempty.wait;
        x:= buffer[outp Mod bufsize];
        outp:= outp+ 1;
        nonfull.signal
      End;

 Begin inp:= 0; outp:= 0 End.
```

Program 5.2. Reader-Writer Database in Monitor

```
  Monitor reader-writer;

    Var
      readers, writers: integer;
      reading, writing: condition;

    Procedure start-read;
      Begin
        if writers <> 0
           then reading.wait;
        readers:= readers+ 1;
        reading.signal
      End;

    Procedure end-read;
      Begin
        readers:= readers- 1;
        if readers= 0
           then writing.signal
      End;

    Procedure start-write;
      Begin
        writers:= writers+ 1;
        if (readers <> 0) or (writers <> 0)
           then writing.wait
      End;

    Procedure end-write;
      Begin
        writers:= writers- 1;
```

```
            if writers<> 0
               then writing.signal
               else reading.signal
          End;

     Begin readers:= 0; writers:= 0 End.
```

4.4.1. Discussion

Though monitors solve the mutual exclusion problems, the signal and wait operations on condition variables suffer the same structure problem as P and V operations on semaphores. In addition, there is very little support for concurrency expression in Monitors.

(1) Low level synchronization is mixed with high level scheduling in Monitors. Consider the following example:

```
          Monitor M1;

            Var x: integer;
                c1, c2: condition;

            Procedure A1;
               Begin
                     .
                  If x>3 Then c1.wait;
                     .
               End;

             Procedure A2;
               Begin
                     .
                  If x< 1 Then c2.wait;
                     .
               End;

            Procedure A3;
               Begin
               .
         x:= 2;
         (* question point *)
         ? c1.signal or c2.signal ?
               .
        End;

    Begin .... End.
```

Hoare´s definition of monitors requires that a procedure waiting on a condition variable must run immediately when another procedure signals that variable, and the signaling procedure in turn [illegible] as soon as the waiting procedure leaves the m[illegible]or. This requirement may seriously distort the [illegible]ll scheduling policy. A process waiting on a con[illegible]tion pre-empts the process executing a condition signal operation, even though the priority of the former is lower than the later. For example, at the "question point" in monitor M1, all the three procedures: A1, A2, and A3, can be scheduled next without violating the data integrity (consistency) requirement. The introduction of the signal operation at this point unavoidably dictates a specific scheduling policy.

In the reader-writer database, inside the procedure end-write, the statement

```
if writers <> 0
   then writing.signal
   else reading.signal
```

gives priority to writers over readers, which is only a design decision and not a requirement of the problem itself. This problem can be solved by the introduction of nondeterminacy as in ADA's Rendezvous and Mao and Yeh's Communication Ports, or by automatic condition evaluation and signaling as in Brinch Hansen's EDISON [BRI80].

(2) Goto-like behavior of signal and wait primitives makes monitor programs difficult to follow and verify. Dahl's suggestion that signals should always be the last operation of a monitor procedure is, in fact, a very natural simplification [HOA74]. The monitor solutions to the bounded-buffer and the reader-writer database have this property. Moreover, as Lampson indicates in [LAMB80], a signal operation is only an implementaticn hint for ease of scheduling. Both Dahl's and Lampson's suggestions are incorporated in ADA's Rendezvous (or Mao and Yeh's Communication Ports [MAO80]) and in Brinch Hansen's EDISON, which do without the signal primitive in their synchronization mechanisms.

(3) Monitors were mainly designed to specify mutual exclusion on shared objects. There is little support for concurrency expression. To specify concurrency among readers in the reader-writer database, we have to decompose the read operation into three

suboperations: start-read, readop, and end-read. The monitor specifies the synchronization among start-reads, and end-reads, but leaves readop unsynchronized. Instead, it requires the calling sequence of a user process to be start-read, readop and end-read. What if the user does not (or forgets to) follow the convention? The problem is almost the same as P and V primitives as discussed in Section 4.3. This problem can be solved by the right/synchronization controller mechanism (similar to extended capabilities) introduced by Michael Conner [CONM79], or the "procedure" invocation mechanism introduced in ADA´s Rendezvous.

4.5. ADA´s Rendezvous

An ADA task defines a set of "entries" and "procedures", which are visible to and can be invoked by other tasks. An entry is similar to a monitor procedure in that it is a mechanism to achieve mutual exclusion. A task can only "accept" an entry exclusively at one time. On the other hand, a "procedure" body can be executed concurrently. An entry can be called externally (by another task) or internally (by a procedure). A task may allow a certain sequence of calls by simply sequencing a group of "accept" statements. When an accept statement is encountered, the corresponding entry call will be executed.

The "when guard" [1] in ADA is similar to the wait condition concept in Monitors. A when guard is a boolean expression which has to be true for the entry to be accessible. A construct called the "select" statement is introduced to allow nondeterministic choice of several true guards. This capability increases the expressive power of ADA tremendously. A "select" statement is different from a "case" statement (as in PASCAL) in that if more than one branch can be taken (with true guards and outstanding calls) a random choice is made for the "select" statement; there should be one and only one true branch in a "case" statement. Solutions in ADA to the bounded-buffer and the reader-writer database appear in Program 4.3. and Program 4.4., respectively.

Program 4.3 Bounded-Buffer in ADA

```
Task bounded-buffer Is

   Entry produce (m1: IN elem);
   Entry consume (m2: OUT elem);

End;

Task body Bounded-buffer Is

   Var
```

[1]The concept is originally proposed by Dijkstra in [DIJ76].

```
            buf: array 0.. bufsize- 1 of elem;
            inp: integer := 0;
            outp: integer := 0;

      Begin
         loop
           select
             when inp- outp < bufsize =>
                accept produce (ml: IN elem) do
                   buf [inp Mod bufsize] := ml;
                end;
             inp:= inp+ 1;
           or
             when inp > outp =>
                accept consume (m2: OUT elem) do
                   m2:= buf[outp Mod bufsize];
                end;
             outp:= outp+ 1;
           end select;
         end loop;

       End bounded-buffer.
```

Program 4.4. Reader-writer Database in ADA

```
   Task reader-writer Is

     Procedure read(v: OUT elem);
     Procedure write (e: IN elem);

   End;

   Task body reader-writer is

     Var s: elem;
         readers: integer := 0;
     Entry start-read;
     Entry stop-read;
     Entry start-write;
     Entry stop-write;

     Procedure read (v: OUT elem) Is
       Begin
         start-read;
```

```
    v:= s;
    stop-read;
  End;

Procedure write (e: IN elem) Is
  Begin
    start-write;
    s:= e;
    stop-write;
  End;

Begin
  accept start-write;
  accept stop-write;
  loop
    select
      when writers= 0 =>
         accept start-read;
            readers:= readers+ 1;
    or
        accept stop-read;
            readers:= readers- 1;
    or
       when (writers = 0) and (readers = 0) =>
         accept start-write;
            writers:= writers+ 1;
    or
       accept stop-write;
          writers:= writers- 1;
    end select;
  end loop;

End reader-writer.
```

4.5.1. Discussion

(1) By the introduction of non-deterministic select statements, ADA manages to be a high level description language which specifies the problem (e.g., reader-writer database) without making design decisions (e.g., reader or writer priority). The traditional argument against nondeterminacy is its ineffi-

ciency. While this argument is true in uniprocessor systems, it is no longer valid in today´s multiprocessor systems. In Brinch Hansen´s EDISON, Lampson´s MESA [LAMB80] and Atkinson´s Automatic Verification of Serializers [ATK80] similar arguments are discussed in favor of nondeterminacy.

(2) By the introduction of the "procedure" invocation concept, ADA is able to specify process concurrency without suffering data protection problems as in Monitors. While scheduling can be separated from synchronization in ADA, the control-related and data-related properties are still mixed together. Control and data are quite *different* properties [LAV79, BOS79, CAM74]. For instance, in the bounded-buffer, the input and output pointers (inp and outp variables) specify synchronization properties only, but do not affect directly the value of array buf[1..n-1] except "controlling" "when" the data can be retrieved or updated.

Path Expressions [CAM74] are one of the outstanding approaches which separate control from data specification in a monitor.

4.6. Path Expressions

The term "path expression" denotes many notations that are based on the scheme introduced by Campbell and Habermann. These include Regular Path Expressions [CAM74], Open Path Expressions [CAM74], Predicate Path Expressions [AND79], and other versions which are restrictions or combinations of the above. Notations that are similar to paths that model system behavior were developed independently by Shaw (Flow Expressions [SHA78]) and Riddle (Event Expressions [RID76]). A very good survey is [SHA79]. We choose Open Path Expressions (OPE) as a representative of this class since they seem to be the most popular version. Furthermore an implementation (Path PASCAL) exists for them.

4.6.1. Open Path Expressions

Open path expressions allow three distinct kinds of constraints to be specified: sequencing (denoted by ´;´), resource restriction (denoted by ´n:()´) and resource derestriction (denoted by ´[]´). Each of these can be combined with the other forms to provide complex synchronization constraints and several constraints can be contained in a single path expression.

4.6.1.1. Concurrency

A path with no synchronization information consists of a comma-separated list of operation names surrounded by path and end. The path:

```
path name1, name2, name3 end.
```

imposes no restriction on the order of operation invocation and no restriction on the number of concurrent executions of ´name1´, ´name2´, and ´name3´.

4.6.1.2. Sequencing

The sequencing mechanism imposes an order on procedure executions. The order is specified by a semicolon-separated list. In the example below:

```
path first; second; third end.
```

one execution of operation ´first´ must complete before each execution of ´second´ may begin, and one execution of ´second´ must complete before each execution of ´third´ can begin. However, there is no constraint on the number of initiations of ´first´.

4.6.1.3. Restrictions

Limited resources occasionally make it desirable to limit the number of concurrent executions of an operation.

The resource restriction specification allows concurrent execution of operations to proceed until the restriction limit is reached. Restrictions are denoted by surrounding the expression to be restricted by parenthesis and preceding it with the integer restriction limit and a colon. A typical example is the bounded buffer of size n. The path expression below:

```
path n: (1: (produce); 1: (consume)) end.
```

imposes that every "consume" operation has to be preceded by a "produce" operation, and allows only n concurrent "produce; consume" paths to happen at any time. The "1" in front of "produce" ("consume") inhibits the concurrent execution of the "produce" ("consume") operations.

4.6.1.4. Derestrictions

For some applications it is convenient to process all calls to an operation once that operation's execution has begun. Such a situation might occur when a large spooler is brought into memory to process I/O requests. The specifier denoting "derestriction" of a list of operations is shown by surrounding the list in square brackets. The path:

```
path setup; [spooler] end.
```

requires a "setup" to be executed before each sequence of "spoolers" to proceed, but once a "spooler" has begun execution the "spoolers" proceed until all executions have terminated. Afterwards, a "setup" must again complete before any "spooler" can proceed. The Path PASCAL's solutions to the bounded-buffer and the reader-writer database are depicted in figure 4.5 and Program 4.6. respectively.

Program 4.5. Bounded-Buffer in Path Pascal

```
Type bounded-buffer = Object

   Path n: (1 (produce); 1: (consume)) end;

   Var
      buffer : array 0..bufsize- 1 of elem;
      inp, outp: integer;

   Entry Procedure produce (m: elem);
      Begin
        buffer[inp Mod bufsize]:= m;
        inp:= inp+ 1;
      End;

   Entry Procedure consume (Var m: elem);
      Begin
        m:= buffer[outp Mod bufsize];
        outp:= outp+ 1;
      End;

   Init; Begin inp:= 0; outp:= 0 End;

End object.
```

Program 4.6. Reader-Writer Database in Path PASCAL

```
Type reader-writer= Object

   Path write; [read] End;

   Var s: elem;

   Entry Procedure write (e: elem);
     Begin
       s:= e;
     End;

   Entry Function read: elem;
     Begin
       read:= s;
     End;

   Init; Begin End;

End object.
```

4.6.2. Discussion

By introducing path expressions, Path PASCAL separates control-related from data-related properties. There are neither condition variables, nor signal, nor wait (or when) constructs in the language. However, there are still some intrinsic problems with path expressions.

(1) The meaning of a path expression depends too much on a particular interpretation (implementation). For instance, a path expression specifies the reader-writer database integrity property by

path write; [read] end.

The correctness of the specification depends on the meaning of the notation [ ]. Consider a particular situation:

Operation	Event	Time
write1	arrived	t1
read1	arrived	t2
read2	arrived	t3
write2	arrived	t4
read3	arrived	t5
write1	executed	t6
read4	arrived	t7
write1	finished	t8
read5	arrived	t9

At time t8, the specification clearly states that read1 and read2 will be executed next. However, the precedence between write2 and read3, write2 and read4, write2 and read5 is not so clear; it depends on a particular implementation of compiler to interpret the meaning of [ ]. Worse, while it is possible to assign a special interpretation to [ ], another kind of priority policy may make it difficult to "tune" to the new priority policy from the old interpretation of [ ].

(2) Different synchronization properties such as mutual exclusion (to maintain data integrity), scheduling (to allow priority policies) or process communication (to exchange messages between processes), have to be intergrated into a single ad hoc path expression. A consequence is that it is difficult to extend a path expression to meet a new requirement [BLO79]. A new requirement usually results in a reconstruction of an

old path expression.

(3) Though control and data-related properties are specified separately, a non-procedural language (Path Expressions) is used to specify the control-related properties, while a procedural language (PASCAL) is used to describe the data-related properties. This undesirable mixture of procedural and non-procedural languages complicates issues, especially when dealing with design verifications.

The Transaction-Based Specification Language (TBS), a sublanguage of EBS, which extends the work of [GRE77] and [LAV79], solves most of the problems above.

4.7. The Transaction-Based Specification (TBS)

A user calls a system by submitting a command with input/output parameters. We define the sequence of events caused by the invocation of a user command a transaction. A transaction TR is characterized by two external events: TR.begin and TR.end, which are visible to the user; and a sequence of internal events which happen between these two external events. As with an event, a transaction is atomic, in that the whole sequence of contained events happens completely or not at all.

Thus, we may represent the event sequence of a transaction TR by (e(1), e(2), ..., e(n)). The underlying semantics (in terms of the event model) of this notation is that if e(i) happens then e(i+1) will happen eventually, for all i greater than zero and less than n, where e(1) and e(n) are the beginning and the end of the transaction and represented by TR.begin and TR.end, respectively. This is expressed formally in EBS as follows:

$$e(i) \Rightarrow e(i+1) \text{ for all } i,\ 1 \leq i < n.$$

4.7.1. The Transaction Relations

We now extend the event relations to transaction relations. The first considered is the transaction ordering relation represented by +>. Two transactions are ordered by +>, if and only if the end of one transaction precedes the beginning of the other. Formally, two transactions, say TR1 and TR2, satisfy the relation TR1 +> TR2, iff TR1.end -> TR2.begin. It follows from this definition and the properties of ->, that the relation +> has the following properties:

(1) Non-reflexivity: ~(TR +> TR);

(2) Antisymmetry: TR1 +> TR2 #> ~(TR2 +> TR1); and

(3) Transitivity: TR1 +> TR2 ^ TR2 +> TR3 #> TR1 +> TR3.

In other words, the relation +> is also a partial ordering relation.

```
    TR1.begin   TR1.end
-------*-----------*--------*-----------*----------
                       TR2.begin     TR2.end
```

Figure 4.1. Transaction Ordering Relation +>

Based on the transaction ordering relation +>, the transaction mutual exclusion relation can be defined easily. Two transactions, say TR1 and TR2 are mutually exclusive iff

TR1 +> TR2 v TR2 +> TR1.

Similarly, the transaction concurrency relation between them is denoted by TR1 || TR2, and is defined as

~(TR1 +> TR2) ^ ~(TR2 +> TR1).

In a particular environment where events can be totally ordered, this definition is equivalent to

TR1.begin-> TR2.end ^ TR2.begin-> TR1.end

which easily proven.

```
           TR1.begin               TR1.end
--------------*-------*---------------*---------*---------
                    TR2.begin                 TR2.end
```

Figure 4.2. Transaction Concurrency Relation ||

It should be noted that both the concurrency relation // defined among events and || defined among transactions are reflexive and symmetric; however, neither is transitive. Figure 4.3. and Figure 4.4. show the examples where they are not transitive.

4.7.2. The Transaction Type and the Ordinal Function

We specify the behavior of a system in terms of the transactions that are allowed to happen in the computation history of the system. In order to do that, we need to have some naming mechanism to reference transactions (and events) in the history. This can be achieved by introducing of the concepts of the transaction type and the ordinal function. A transaction type is defined as a set of transaction instances with an ordinal function mapping from transaction instances to natural numbers. The ordinal function is referred to as the invocation order of the transaction instances (i.e., the order of the begin event) in the history of a transaction type. By convention, each user command constitutes a transaction type and is represented by the command name in capital letters.

4.7.3. The Language TBS

Based on the concepts above, we develop a language called TBS (Transaction-Based Specification Language), to specify the transaction-oriented systems. Since the

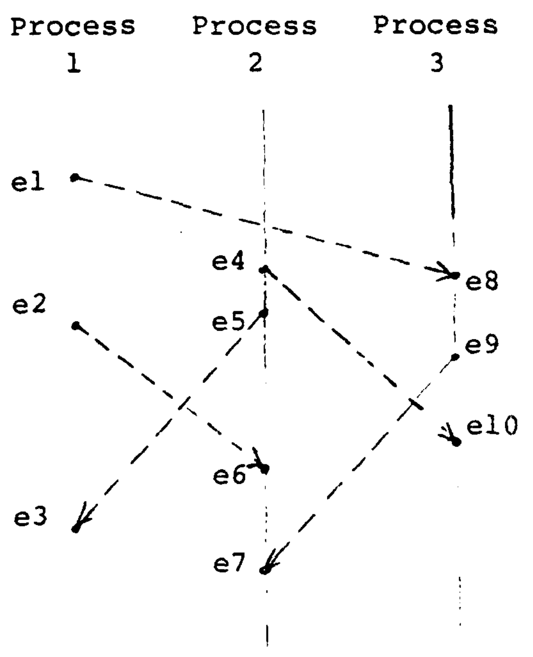


Figure 4.3. Non-transitivity of //:
(e1//e4) and (e4//e9)
but e1→ e9

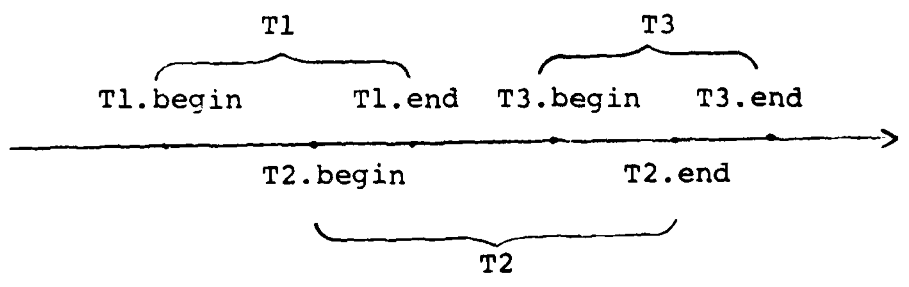


Figure 4.4. Non-transitivity of || :
(T1 || T2) and (T2 || T3)
but (T1+> T3)

syntax of TBS is very similar to EBS, instead of presenting a formal syntax for TBS, the differences between TBS and EBS are indicated:

(1) A system specification in TBS begins with the reserved word Transystem.

(2) The messagetype definition part in EBS is replaced by Transaction definition in TBS: each parameter is preceded by the reserved words IN or OUT, meaning that the parameter is either input to or output from the system, respectively.

(3) Variables in TBS range over the domain of transactions, instead of events as in EBS; and the command names in capital letters represent transaction types.

4.7.4. A TBS's Solution to the Reader-Writer Database

A legal trace of the history of this system could be as follows:

History	Ordinal Functions	
	WRITE	READ
write1.begin	1	-
write1.end	-	-
read1.begin	-	1
read2.begin	-	2
read5.begin	-	3
read1.end	-	-
read5.end	-	-
read2.end	-	-
write2.begin	2	-
write2.end	-	-
write5.begin	3	-

```
        write5.end          -      -
```

The mutual exclusion among write´s can be specified as

```
    ∀ w1, w2∈ WRITE
      w1= w2 v
      w1 +> w2 v w2 +> w1
```

Similarly, the mutual exclusion between read´s and write´s can be specified as

```
    ∀ r∈ READ, w∈ WRITE
      r +> w v w +> r
```

There is no restriction on the order among read´s. They can proceed concurrently. The data-related property is specified as follows:

```
    ∀ r∈ READ, w∈ WRITE
      w +> r ^
      ( ∀ w´∈ WRITE
          w´ +> r #> w´= w v w´ +> w)
      #> r.msg= w.msg
```

which says that a reader (r) will get the information (r.msg) that was put before its arrival by the last writer (w). Note that this specification has the desirable property that when no write transaction precedes a read transaction then the contents of the message obtained by that read command is not constrained by the specification, but is open to implementation interpretations.

The complete specification of the reader-writer database in TBS appears in Program 4.7.

Program 4.7. Reader-Writer Database in TBS

```
Transystem reader-writer-database;

  Transaction
     READ (OUT msg: elem);
     WRITE (IN msg: elem);
  End transaction;

  Behavior

    (* Mutual exclusion among write´s *)
       ∀ w1, w2◂ WRITE
         w1= w2 v
         w1 +> w2 v w2+> w1;

    (* Mutual exclusion between read´s and write´s *)
       ∀ r◂ READ, w◂ WRITE
         r +> w v w +> r;

    (* A read will get the contents of the last write *)
       ∀ r◂ READ, w◂ WRITE
         w +> r ^
         ( ∀ w´◂ WRITE
             w´ +> r #> w´= w v w´ +> w)
         #> r.msg= w.msg

  End behavior;

End system.
```

4.7.5. A TBS´s Solution to the Bounded-Buffer System

The property that a consumer cannot get something from nothing (i.e., has to wait until the buffer is not empty) can be specified as

```
∀ c∈ CONSUME ∃ p∈ PRODUCE
  ord(c)= ord(p)  ^ p +> c;
```

The property that a producer cannot deposit a message when the buffer is full can be specified as

```
∀ p∈ PRODUCE
  ord(p)  > bufsize
  #> ∃ c∈ CONSUME
       (ord(c)= ord(p)- bufsize)  ^  (c +> p)
```

where bufsize is the size of the buffer. Since a bounded buffer is normally a queue (instead of a pool) of message slots, we require that producers not deposit messages concurrently into the buffer, and that consumers not receive messages concurrently from the buffer. These properties can be specified as

```
∀ c1, c2∈ CONSUME, p1, p2∈ PRODUCE
  (c1= c2 v c1 +> c2 v c2 +> c1) ^
  (p1= p2 v p1 +> p2 v p2 +> p1);
```

It can be easily proved that the specification is equivalent to

```
∀ c1, c2∈ CONSUME, p1, p2∈ PRODUCE
  (ord(c2)= ord(c1)+ 1 #> c1 +> c2) ^
  (ord(p2)= ord(p1)+ 1 #> p1 +> p2);
```

This specification only <u>serializes</u> the transactions among "consumes" and the transactions among "produces". If the buffer is neither full nor empty, then it is still possible that the producer deposits a message into the buffer while the consumer obtains a message from the buffer. This additional concurrency cannot be specified either in ADA´s Rendezvous nor in Monitors.

A bounded-buffer also behaves as a first-come-first-serve message queue, i.e., the sequence of messages obtained by the consumers is identical to those deposited by the producers. This is specified as

```
∀ c< CONSUME, p< PRODUCE
  ord(c)= ord(p) #> c.msg= p.msg;
```

The complete specification of the bounded-buffer in TBS appears in Program 4.8.

Program 4.8. Bounded-Buffer in TBS

```
Transystem bounded-buffer;

   Transaction
      PRODUCE (IN msg: elem);
      CONSUME (OUT msg: elem);
   End transaction;

   Behavior

      (* BB1: A consumer has to wait until the
         buffer is not empty.
      *)
         ∀ c< CONSUME ∤ p< PRODUCE
           ord(c)= ord(p) ^ p +> c;

      (* BB2: A producer has to wait until the buffer
         is not full, where the buffer is of size
```

```
            bufsize.
        *)
            ∀ p< PRODUCE
              ord(p) > bufsize
              #> ∄ c< CONSUME
                   (ord(c)= ord(p)- bufsize) ^ (c +> p);

        (* BB3: There are neither  concurrent PRODUCE´s
            nor concurrent CONSUME´s.
        *)
            ∀ c1, c2< CONSUME, p1, p2< PRODUCE
               (c1= c2 v c1 +> c2 v c2 +> c1) ^
               (p1= p2 v p1 +> p2 v p2 +> p1);

        (* BB4: The sequence of messages obtained by the
            consumer is exactly the same as that
            was put by the producer.
        *)
            ∀ c< CONSUME, p< PRODUCE
              ord(c)= ord(p) #> c.msg= p.msg;

      End behavior;

  End system.
```

4.7.6. Discussion

The approaches of TBS and path expressions specify a system by a subset of possible system histories. However, TBS is more abstract than path expressions. In the reader-writer database, we state the data integrity properties independent of priority do that they are properties of any reader-writer database regardless of priority. The important perspective is that we stick strictly to the "separation of concern" principle and specify orthogonal properties separately. In contrast, the path expressions tend to take an integrated approach and come out with complex and difficult-to-understand solutions. A direct consequence is that specifications in TBS are more understandable, minimal and extensible than those in path expressions.

The bounded-buffer solution (Program 4.5.) in ADA shows that when the buffer is neither full (inp- outp < n) nor empty (inp > outp) then it will <u>nondeterministically</u> accept a produce or a consume entry. This nondeterminacy is a big improvement over the Monitor solution (Program 4.1.), which specifies more non-essential scheduling (or priority) policies. However, the nondeterminacy solution in ADA is still not very satisfactory. In reality, a producer and a consumer are allowed to proceed concurrently without violating other data integrity requirements when the buffer is partially full. Nondeterminacy is much stronger than "concurrency" since transactions (represented by entries in ADA) are mutually exclusive to each other in "nondeterminacy" so that there is still total ordering between transactions.

In TBS, data integrity requirements are specified by the transaction ordering relation while (possible) concurrency is specified implicitly by the <u>lack</u> of transaction ordering. This is a natural way to represent the (possible) concurrency between a "produce" and a "consume" transaction. The resulting specification of the bounded-buffer in TBS is minimal in the sense that it describes properties of any bounded-buffer.

4.8. The Semantics of TBS

The semantics of a transaction has been defined in section 4.7. We now elaborate the semantics of the transaction types and the transaction relations.

4.8.1. The Semantics of Transaction Types

In TBS, each user command constitutes a transaction type. A transaction type is defined as a set of transaction instances with an ordinal function mapping from transactions into natural numbers. A transaction definition in TBS defines two ports: an inport, corresponding to the "BEGIN" event set of the transaction type, having the "IN" parameters as its message types; and an outport, corresponding to the "END" event set of the transaction type, having the "OUT" parameters as its message types.

There is a strong relationship between these two ports. From the "BEGIN" port to the "END" port there is a reliable transmission system (except for the properties RT15, and RT21), i.e., "BEGIN" and "END" ports satisfy the properties: RT11(BEGIN, END), RT14(BEGIN, END) : one "begin" event enables one and only one "end" event; and RT12(BEGIN, END), RT13(BEGIN, END): there is no internally or externally generated "end" event except enabled by a "begin" event.

The ordinal function is, by definition, referred to as the <u>invocation</u> order (i.e., the order of the "begin" event) of the transaction instances in the history of transaction type. Thus, in the expression, ord(x, TR), the ordinal number of transaction instance x in the history of transaction type TR, can be translated directly as ord(x.begin, TR.BEGIN).

4.8.2. The Precedes Relation +> and the Concurrency Relation ||

The expression TR1 +> TR2 in TBS can be translated into EBS as TR1.end -> TR2.begin. The expression TR1 || TR2 can be translated into EBS as ~(TR1 +> TR2) ^ ~(TR2 +> TR1), which can in turn be translated into ~(TR1.end -> TR2.begin) ^ ~(TR2.end -> TR1.begin) in a multiprocessor interpretation or (TR1.begin -> TR2.end) ^ (TR2.begin -> TR1.end) in a centralized processor interpretation.

4.9. The Structure Specification and Verification in TBS

The structure specification in TBS is almost the same as that in EBS. The major difference are the semantics of <u>links</u> and more structure linkage of transactions.

Two transactions A and B, can be connected only if:

(1) A is a transaction with OUT parameters only;

(2) B is a transaction with IN parameters only;

(3) There is a one-to-one correspondence mapping from each OUT parameter of A to an IN parameter of B having the same message type.

When two transactions are connected by a <u>connect</u> statement, they are merged into a single transaction. Once transactions are merged, it is impossible to distinguish between them; one transaction begins when the other begins, and ends when the other ends.

The semantics of transaction connections as defined above is very similar to the communication protocol (synchronized send-receive) adopted in CSP [HOA78] and the exchange function in PAISLEY [ZAV81].

<u>4.9.1. Example 2: A Double-Buffer</u>

A double-buffer (see Figure 4.5.) that consists of a series connection of two bounded-buffers BX of size bx and BY of size by can be specified as follows:

```
Transystem DBF;

   Transaction
      PRODUCE (IN msg: elem);
      CONSUME (OUT msg: elem);
   End transaction;
```

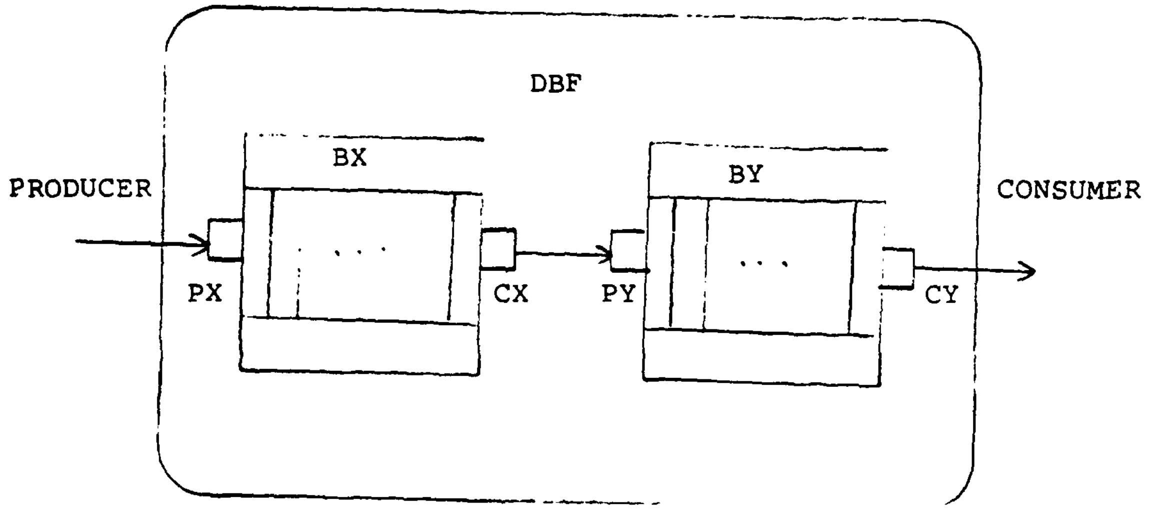


Figure 4.5. A Double Buffer

```
  Structure

     Subsystem BX;

        Transaction
           PX(IN msg: elem);
           CX(OUT msg: elem);
        End transaction;

        Behavior
           BUF(PX, CX, bx);   [2]
        End behavior;

     End subsystem;

     Subsystem BY;

        Transaction
           PY(IN msg: elem);
           CY(OUT msg: elem);
        End transaction;

        Behavior
           BUF(PY, CY, by);
        End behavior;

     End system;

     Network
        connect(BX.CX, BY.PY)== DBF.CP;
     End network;

     Interface
        BX.PX== DBF.PRODUCE;
        BY.CY== DBF.CONSUME;
     End interface;
  End structure;

End system.
```

This double-buffer provides the service of a single bounded-buffer with a buffer size equal to the addition of

[2] A shorthand notation for a buffer of size bx,with PX and CX as its produce and consume respectively.

bx and by. This behavior is specified as follows:

```
Transystem DBF;

   Transaction
      PRODUCE (IN msg: elem);
      CONSUME (OUT msg: elem);
   End transaction;

   Behavior
      BUF (PRODUCE, CONSUME, bx+ by);
   End behavior;

End transystem.
```

The following theorem proves that the structure specification satisfies its behavior specification.

4.9.2. The Verification of the Double-Buffer

Theorem 4.1.

The behavior of a series connection of a buffer BX of size bx and a buffer BY of size by is the same as a buffer of size bx+by, i.e., the behavior specification of transystem DBF is satisfied by its structure specification.

Proof

```
   (* BB1: ∀ c< CONSUME ∤ p< PRODUCE
              ord(p)= ord(c) ^ p+> c *)

       (1) c< CONSUME ...precondition

           there exists cp< CP s. t.
       (2) ord(cp)= ord(c) ^ cp +> c ...(1), BB1(BY)
       there exists p< PRODUCE s. t.
   (3) ord(p)= ord(cp) ^ p +> cp ...(2), BB1(BX)

   (4) ord(p)= ord(c) ^ p +> c ...(2), (3)


(* BB2: ∀ p< PRODUCE
          ord(p) > bx+ by
          #> ∃ c< CONSUME
                ord(c)= ord(p)- (bx+ by) ^ c +> p *)

   (1) p< PRODUCE ...precondition

   (2) ord(p) > bx+ by ...precondition

   (3) ord(p) > bx  ...(2)

       there exists cp< CP s. t.
   (4) ord(cp)= ord(p)- bx ^ cp +> p
                           ...(2), BB2(BX)

   (5) ord(cp) > by  ...(2), (4)

       there exists c< CONSUME s. t.
   (6) ord(c)= ord(cp)- by ^ c +> cp
                           ...(5), BB2(BY)

   (7) ord(c)= ord(p)- (bx+ by) ^ c +> p
                                  ...(4), (6)


(* BB3: ∀ ci, cj< CONSUME, pi, pj< PRODUCE
          (ci= cj v ci +> cj v cj +> ci) ^
          (pi= pj v pi +> pj v pj +> pi)   *)

     Trivial


(* BB4: ∀ c< CONSUME, p< PRODUCE
          ord(c)= ord(p) #> c.msg= p.msg  *)

   (1) c< CONSUME, p< PRODUCE ord(c)= ord(p)
                                   ...precondition

       there exists cp< CP s. t.
   (2) ord(cp)= ord(c) ^ cp +> c ...BB1(BY)

   (3) cp.msg= c.msg ...(2), BB4(BY)

       there exists p0< PRODUCE s. t.
   (4) ord(p0)= ord(cp) ^ p0 +> cp ...BB1(BX)
```

(5) p0.msg= cp.msg ...(4), BB4(BX)

(6) p0.msg= c.msg ...(3), (5)

(7) ord(p0)= ord(c)= ord(p) ...(1), (2), (4)

(8) p= cp ...(7)

(9) p.msg= c.msg ...(6), (8)

Q. E. D.

CHAPTER 5

EVENT COORDINATIONS

5.1. Coordinations

By event coordination we mean the coordination of two or more events to enable a third event. No example in the previous chapters comprises such event relation. We begin our discussion with an adder circuit, whose behavior is simple conceptually but complicated formally.

5.2. An Adder Circuit

An adder having two input arguments A and B, and an output argument C is depicted in Figure 5.1. It receives messages from A and B, adds their contents and sends the result to outport C. We first look at two possible solutions by traditional approaches: Petri Nets and ADA´s Rendezvous.

5.3. Petri Nets

A solution to the adder circuit by Petri Nets is depicted in Figure 5.2.

The graph contains two types of nodes: circles (called places) and bars (called transitions). These nodes, places and transitions, are connected by directed

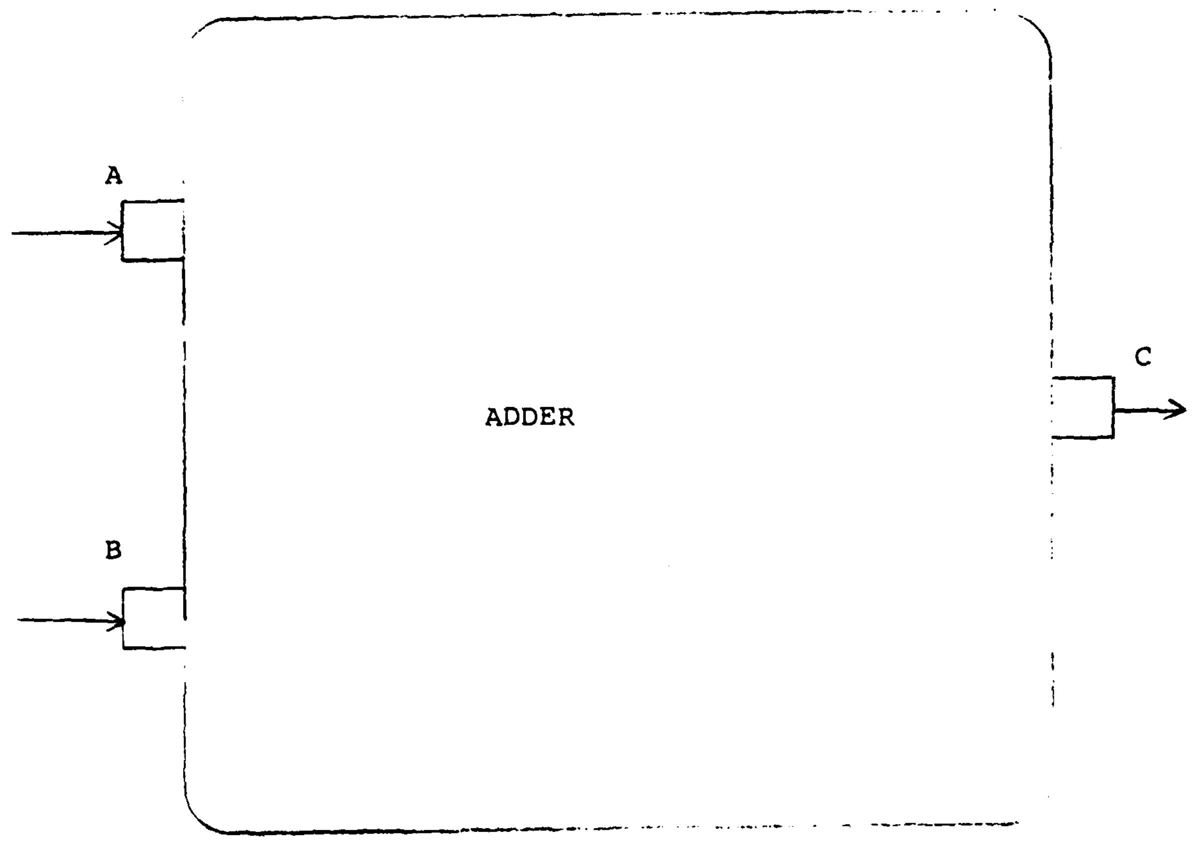


Figure 5.1. An Adder Circuit

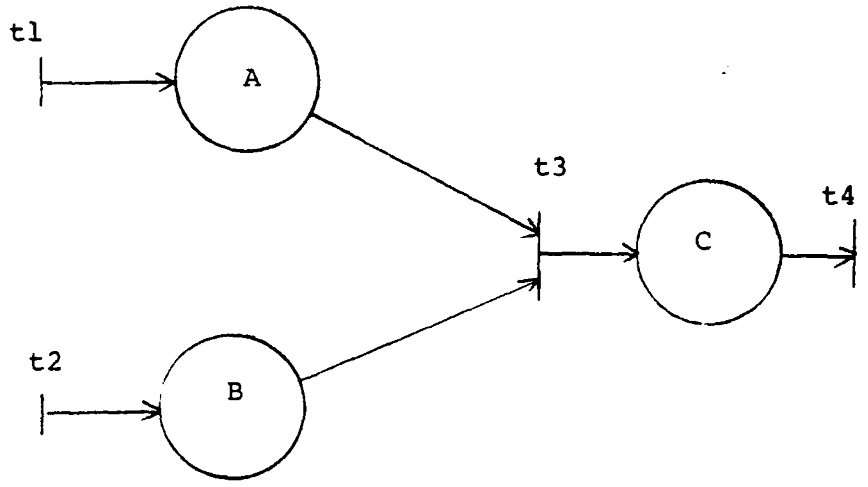


Figure 5.2. A Petri Net Specification of the Adder Circuit

arcs from places to transitions and from transitions to places. If an arc is directed from node i to node j then i is an input to j, and j is an output of i. In Figure 5.2., for example, places A and B are inputs to transition t3; place C is an output of tıansition T6. A transition without an input arc or an output arc is called _a source_ or _a sink_ respectively. In Figure 5.2., t1 and t2 are sources, while t4 is a sink.

In addition to the static properties represented by the graph, a Petri net has dynamic properties that result from its execution. The execution of a Petri net is controlled by the position and the movement of markers (called tokens) in the net. Tokens, indicated by black dots, reside in the circles representing the places of the net.

Tokens are moved by the _firing_ of the transitions of the net. A transition must be _enabled_ in order to fire. A transition is enabled when all of its input places have a token in them. The transition fires by moving the enabling tokens from their input places and generating new tokens which are deposited in the output places of the transition. In Figure 5.2., for example, the transition t3 is enabled if both the input places A and B have a token in them. It then can be fired, removing the two tokens from A and B and placing one token into C. On the other hand, if either place A or place B is empty then a transition waits for tokens to come at the empty place. In this way, places can be synchronized.

The example above illustrates several points about Petri nets. One is inherent concurrency or parallelism. There are two kinds of independent entities in the systems: one from source t1, the other from source t2. There is no need to synchronize the generation of these entities. Thus tokens may enter the place A or B at any time. However, when synchronization is necessary, (for example the generation of C from both A and B), the situation is also easily modeled.

Another major feature of Petri nets is their asynchronous nature. There is no inherent measure of time or the flow of time in a Petri net. As in our event model, this reflects a philosophy of time defined by a partial ordering of the occurrence of events.

A third feature is the nondeterminacy in Petri net execution. If at any time more than one transition is enabled, then any of them may fire. The choice as to which transition fires is made in a nondeterministic manner, i.e., randomly or by forces that are not modeled. Thus a Petri net would seem to be ideal for modeling systems of distributed control with multiple concurrent precesses.

Though Petri Nets have been claimed to be a major model of concurrent systems [BAE73], there are several inherent problems in using them for behavior modeling. First of all, Petri nets model only control-related properties and not data-related properties. For example, in the adder circuit, the property that

" The contents of a messages in C is equal to the sum of those from A and B"

is not modeled in Figure 5.2.

Another problem with Petri nets is that the tokens in a place of a Petri net are treated as indistinguishable; therefore, process priorities cannot be easily modeled.

5.4. A Solution in ADA´s Rendezvous

A solution by using ADA or CP-like language is as follows:

```
Adder Solution in ADA

Task adder is

  Entry A(x: IN elem);
  Entry B(y: IN elem);

End task;

Task body adder is

Var
   buf: elem;
   acome: boolean;
```

```
Begin
   loop
     select
       accept A(x: IN elem) do
          buf:= x;
          acome:= true;
       end;
     or
       accept B(y: IN elem) do
         buf:= y;
         acome:= false;
       end;
     end select;
     if acome then
        accept B(y: IN elem) do
          buf:= buf+ y;
        end else
        accept A (x: IN elem) do
          buf:= buf+ x;
        end;
     end if;
     send (buf);
   end loop;

End.
```

The taskbody maintains two local variables: one is the boolean variable acome, which is true if a message occurs first from A and false otherwise; the other is the variable buf, which stores the contents of the first-come message.

The control begins with a non-deterministic acceptance of message either from A or from B. If a message from A (B) comes first then the control will wait for a message from B (A), add the contents of the two messages and send the result out. The control then returns to the loop.

This scheme handles the data-related properties, but, it is obvious that this solution is more implementation-oriented rather than behavior-oriented when compared with the solution using Petri nets.

Moreover, many possible concurrent operations have been ruled out by this solution. Though the acceptance of messages from A and B are two independent events, they are serialized in the solution. And though it is possible to accept further messages from A and B while the system is preparing the result of previous coordinate messages, a serialization on these events is also dictated.

Thus, we turn to our event based behavior specification technique.

5.5. The Coordination-Based Specification Language (CBS)

The enables relation => has been defined between two events. This relation is not good enough to specify event coordinations, which involves more than two events. Instead of introducing a completely new concept, we extend the enables relation => to specify such kind of event coordinations.

We define a relation => of n+1 arguments, {e1, e2, ..., en} => e, to mean that a group of events e1, e2, ..., en cooperate to enable the event e. Since events cooperate on a fair basis, the order of the enabling events (i.e., e1, ..., en) on the left hand side of the relation is unimportant. To emphasize this fact, we use set notation {} to group the enabling events instead of using paratheses (), which is usually used to denote ordered sequences.

Definition 5.1. (Coordination Relation)

{e1, e2, ..., en} => e iff the existence of e1, e2, ..., en in the computation history guarantees the existence of e in the future; or formally, iff

```
    (e1< CE ^ e2< CE ^ ... ^ en< CE) #> e< CE [1]
    and  e1-> e ^ e2-> e ^ ... ^ en-> e
```

Definition 5.2.

```
{a1, a2, ..., an} => {b1, b2, ..., bm} iff
       {a1, ..., an} => b1 ^
       {a1, ..., an} => b2 ^
        ...
       {a1, ..., an} => bm
```

We also extend the event "precedes" relation "->" to event set "precedes" relation.

Definition 5.3.

```
{e1, ..., en} -> e iff
       e1-> e ^ e2-> e ^ ... ^ en-> e
```

Definition 5.4.

---

[1] CE represents the computation event set including all the events from ENV, SYS and Ports.

```
{a1, ..., en} -> {b1, ...,em} iff
      {a1, ..., an} -> b1 ^
      ...
      {a1, ..., an} -> bm
```

<u>Theorem</u> <u>5</u>.<u>1</u>. (Partial Ordering of Coordination Relation)

The relation => is transitive, antisymmetric and irreflexive, i.e., it is a partial ordering relation. Formally,

```
(1) S1 => S2 ^ S2 => S3 #> S1 => S3
(2) ~ (S => S)
(3) (S1 => S2) #> ~ (S2 => S1)
```

<u>Proof</u>

The antisymmetry and irreflexivity properties of => follows directly from the same properties of ->.

The proof of transitivity property of => also follows from the transitivity property of -> in addition to the transitivity property of logical implication #>.

<u>Theorem</u> <u>5</u>.<u>2</u>. (Extension Theorem)

```
If S1 => S2 and S3 -> S2
   then S1 U S3 => S2        [2]
```

<u>Proof</u>

Trivial.

<u>Theorem</u> <u>5</u>.<u>3</u>. (<u>Contraction</u> <u>Theorem</u>)

```
If S1 U {a} => S2 ^ S1 => a
   then S1 => S2
```

<u>Proof</u>

```
S1 in US [3] implies a in US (since S1 => a)
             implies S2 in US (since S1 U {a} => S2)
S1 -> S2 (since S1 U {a} => S2)
Thus S1 => S2
```

---

[2] The set union operator is represented by "U".
[3] S1 in S2 means that S1 is a subset of S2, which is usually represented by $S1 \subset S2$.

Theorem 5.4. (Fork-Join Theorem)

If A=> S1, A=> S2 and S1 U S2 => B
then A=> B.

Proof

```
A in US implies S1 and S2 in US
        implies B in US
A-> S1, S1-> B
        implies A-> B
Thus A => B.
```

Because of the extension theorem, the relation => is not very expressive since we can always add some unrelated events into the enabling event group. For this reason, we define another kind of enables relation of events called "collectively enables", to represent the relation that every event in the enabling group contributes to the enabling condition, i.e., there are no redundant events in the enabling group.

Definition 5.5. (Collectively Enables Relation *>)

A set S of events collectively enables an event a, denoted by S *> a, iff S enables a but no proper subset of S enables a, or formally, iff

```
   S => a  and
   ~ ( S´ => a) for all S´ in S, S´<> S
```

Definition 5.6.

```
S1 *> S2 iff
   S1 => S2 and
   ~( S1´ => S2) for all S1´ in S1, S1´ <> S1
```

Theorem 5.5.

The collectively enables relation "*>" is antisymmetric, and irreflexive.

<u>Proof</u>

Trivial.

Unfortunately, the relation "*>" is not transitive. This fact is illustrated in Figure 5.3.

Extending the event enables relation to event set enables relation forces the specification language to use the second-order logic. In doing that, although the expressive power increases to a very large extent, the decisive power decreases tremendously. In particular, there is no automatic theorem prover for second order logic to date.

Instead of introducing the second-order logic to our specification language, we try to compromise the expressive power with the decisive power. We choose to restrict the number of elements in a set (called the <u>degree</u> of a set). Thus, any set notation appearing in an n-degree specification, does not represent an arbitrary set but represents a set which can have at most n elements. This restriction keeps our specification language within the first-order logic so automatic theorem proving is possible.

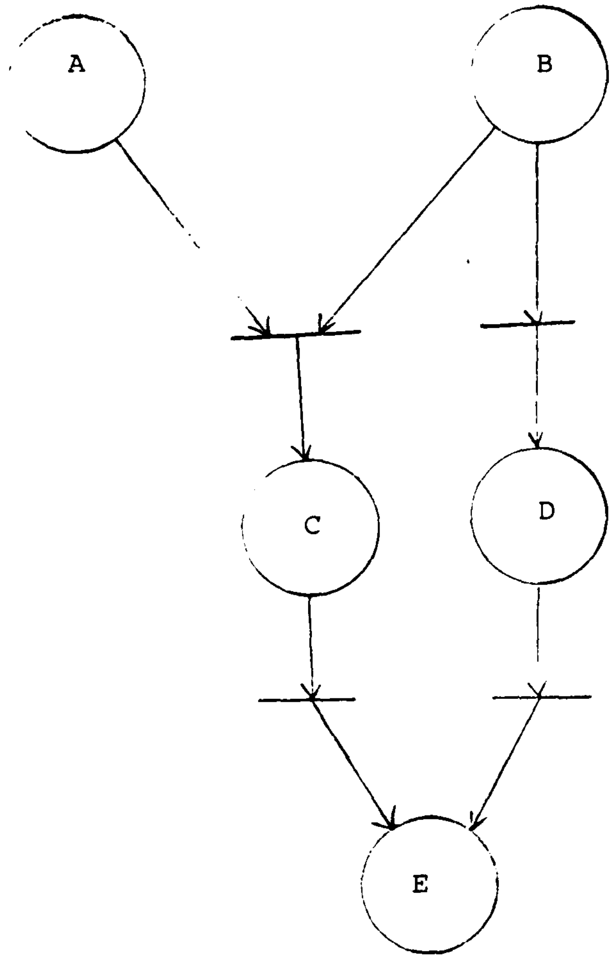


Figure 5.3. Non-transitivity of $*>$: $\{A,B\} *> C$ and $C *> E$ but not $\{A,B\} *> E$

5.5.1. An Adder Circuit Specification in CBS

Using this CBS, we are able to specify the behavior of the adder circuit, which appears as follows:

```
System adder (A: inport;
              B: inport;
              C: outport);

   Degree is 2;
   (* This statement restricts the set variables  in
   the specification to have at most two elements *)

   Behavior

   (* AD10: a single event from either A or B is not
      good enough to enable an event of C *)
      ∀ a∈ A, b∈ B , c∈ C
        ~ (a=> c v b=> c);

   (* AD11: when events from both sides of A  and  B
      come together, an event of C is enabled *)
      ∀ a∈ A, b∈ B
        ord(a)= ord(b) #> ∃ c∈ C {a, b} => c;

   (* AD12: any event of C has to be  enabled  by  a
      cooperation of A and B *)
      ∀ c∈ C ∃ a∈ A, b∈ B
        {a, b} => c;

   (* AD13: any event of C is not  generated  inter-
      nally or externally *)
      ∀ c∈ C
        (∀ S in SYS S*> c
           #> ∃ a∈ A, b∈ B {a,b} *> S) ^
        (∀ E in ENV E*> c
           #> ∃ a∈ A, b∈ B
                E*> {a, b} => c) ^
        (∀ a1, a2∈ A, b1, b2∈ B
           {a1, b1} => c ^ {a2, b2} => c
           #> a1= a2 ^ b1= b2) ^
        (∀ a∈ A, s∈ SYS
           {a, s} *> c
           #> ∃ b∈ B {a, b} => c ^ {a, b} => s) ^
        (∀ b∈ B, s∈ SYS
           {b, s} *> c
           #> ∃ a∈ A {a, b} => c ^ {a, b} => s) ^
        (∀ a∈ A, e∈ ENV
            {a, e} *> c
            #> ∃ b∈ B e => b ^ {a, b} => c) ^
         (∀ b∈ B, e∈ ENV
            {b, e} *> c
            #> ∃ a∈ A e => a ^ {a, b} => c) ^
         (∀ e∈ ENV, s∈ SYS
            {e, s} *> c
            #> (∃ a∈ A e=> a ^ {a, s} *> c) v
               (∃ b∈ B e=> b ^ {b, s} *> c));

   (* AD14: only one event of  C  is  enabled  by  a
      cooperation of A and B *)
      ∀ a∈ A, b∈ B, c1, c2∈ C
         {a, b} => c1 ^ {a, b} => c2 #> c1= c2;

   (* AD15: cooperation is  done  on  a  first-come-
      first-serve basis *)
      ∀ a1, a2∈ A, b1, b2∈ B, c1, c2∈ C
         {a1, b1} => c1 ^ {a2, b2} => c2
         #> (a1= a2 ^ b1= b2 ^ c1= c2) v
            (a1-> a2 ^ b1-> b2 ^ c1-> c2) v
            (a2-> a1 ^ b2-> b1 ^ c2-> c1);

   (* AD21: the contents in event of C is the  addi-
      tion of those from A and B *)
      ∀ a∈ A, b∈ B, c∈ C
         {a, b} => c #> c.msg= a.msg+ b.msg;

   End behavior;

End system.
```

Because of the property AD10, {a, b} => c implies {a, b} *> c. Thus, there is no need to distinguish between the notations => and *> in the specification above when the only events involved are from A, B and C. We group together the control parts (i.e., AD10, AD11, AD12, AD13, AD14, and AD15) of this adder circuit, and call the new system behavior coop(A, B, C). This behavior will be used as a building block in the specification of a connection protocol in section 5.5.3.

5.5.2. The Reliable Transmission System Revisited

The behavior specification of reliable transmission systems was given in Chapter 2; however, the specification was given in terms of first-degree CBS, i.,e., we only considered events being enabled by a single event. The second-degree CBS specification of the reliable transmission system is almost the same as that of a first-degree specification except in specifying that there is no internally or externally generated messages. We need to consider the cases of an event being enabled by two events.

Thus, the behavior RT13 is revised as follows:

```
(* RT13: there is no internally or  externally  gen-
   erated messages *)
   ∀ b∈ B
     (∀ S in SYS S *> b #> ∃ a∈ A a=> S) ^
     (∀ E in ENV E *> b #> ∃ a∈ A E*> a=> b) ^
     ~ (∃ a∈ A, s∈ SYS, e∈ ENV
          {a, e} *> b v {a, s} *> b v {e, s} *> b) ^
     (∀ a1, a2∈ A (a1=> b ^ a2=> b #> a1= a2) ^
                   ({a1, a2} *> b #> a1= a2))
```

This is the only thing changed; other specifications (i.e., RT11, RT12, RT14, RT15, and RT21) remain the same.

5.5.3. A Connection Protocol (CTP)

A connection protocol allows many users in the system, each identified by a port or address, to communicate with one another (see Figure 5.4.). A pair of users must first request a mutual connection before exchanging messages. Once connected, they may simultaneously transfer messages in both directions. When they are finished communicating, they ask to be disconnected.

Connection Protocols appear in many levels in ISO (International Organization for Standardization) model of architecture for Open System Interconnection. Examples are session-connection, transport-connection and network-connection protocols. A connection protocol is different from a data transfer protocol in two important features. First, there are multiple users in a connection protocol. This requires inclusion of explicit addresses in all operations to indicate which users are involved. Second, the connection is established only when both users request to exchange messages; the connection is abolished only when both sides of users agree to terminate the message exchange. "Mutual agreement" implies a need for event coordinations.

5.5.3.1. A Connection Protocol Service Specification in CBS

For simplicity, assume there are fixed k users in the system and let each user be identified by an integer ranging from one to k. There are three kinds of commands from each user i: connect(j), requesting a connection with user j; disconnect(j), requesting a disconnection with user j;

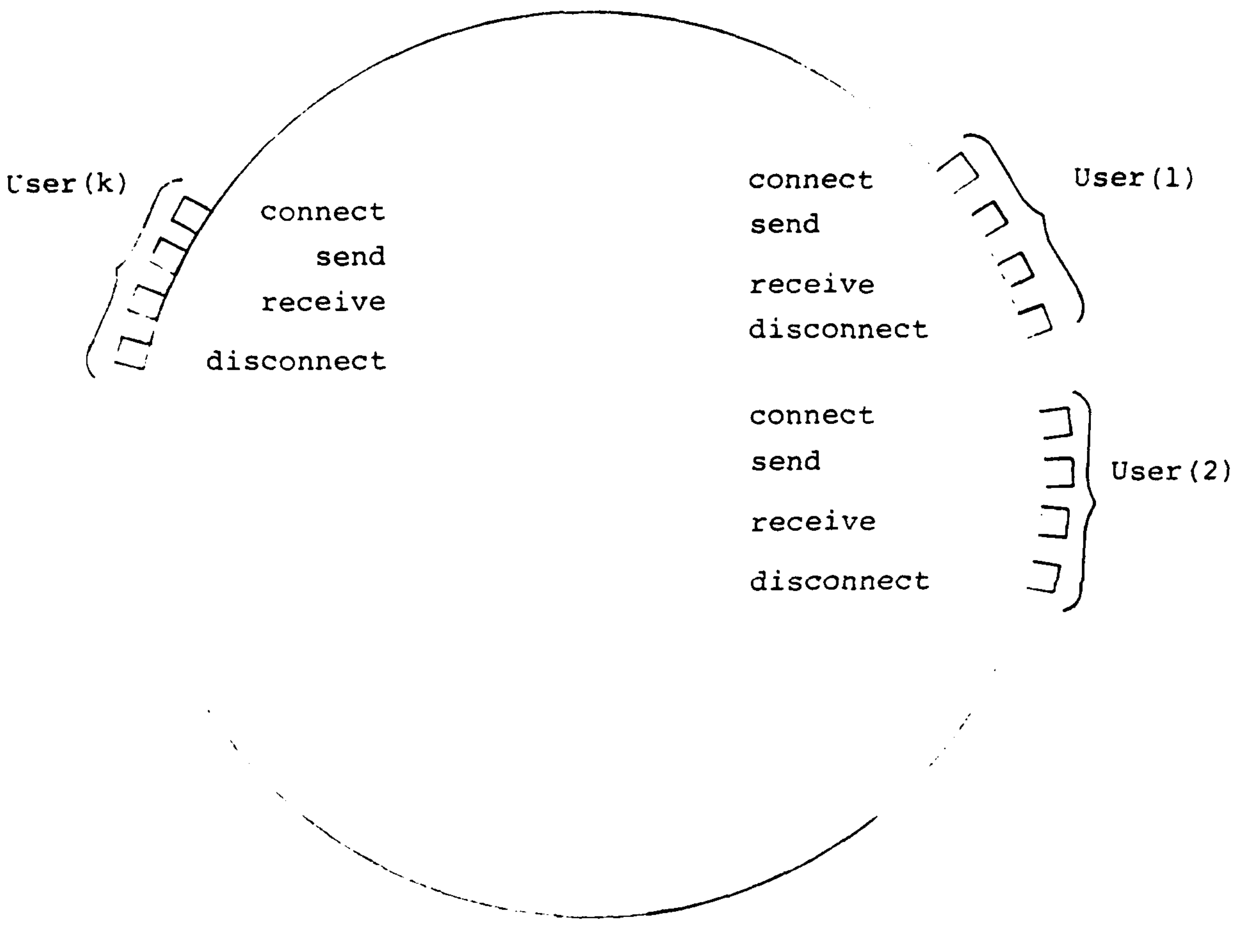


Figure 5.4. A Connection Protocol

and send(j, m), requesting a message m sent to user j. There is one kind of messages to each user i: receive (j, m), meaning that user i receives a message m from user j. We use the notation user(i)|connect to denote the inport connect of user(i). Similar rules are for the commands of disconnect, send and receive.

For ease of expression, we allow the user to specify auxiliary ports. For example, in the specification below, the pconnect(i,j) defines the set of connect(j) events issued by the user i. A port is stronger than a set in the sense that a port also defines the ordering among events. Also it is possible to do without this auxiliary construct in the specification; it is just for ease of expression.

```
System connection protocol
        (user(i)|connect: inport;
         user(i)|disconnect: inport;
         user(i)|send: inport;
         user(i)|receive: outport);

   (*  1 ≤ i ≤ n *)

    Degree is 2;

    Messagetype

      user(i)|connect.msg: integer;
      user(i)|disconnect.msg: integer;
      user(i)|send.msg: record
                          to: integer;
                          data: elem;
                      end;
      user(i)|receive.msg: record
                            from: integer;
                            data: elem;
```

```
                              end;
   (* l ≤ i ≤ n *)

End messagetype;

Define

   (* User-defined ports *)
    pconnect(i,j), pdisconnect(i,j),
    s(i,j), r(i,j) : port;
    (* l ≤ i ≤ n, l ≤ j ≤ n, i <>j *)
    connected (i+j), disconnected(i+j) : port;
    (* l ≤ i < j ≤ n *)

End define;

Behavior

(* pconnect(i,j) is the set of connect(j)  events
   issued by user i. *)
   ∀ x← user(i)|connect
     x.msg=j #> x← pconnect(i,j);
   ∀ x← pconnect(i,j)
     x← user(i)|connect ^ x.msg= j;

   (* l ≤ i ≤ n, l ≤ j ≤ n, i <> j *)

(* pdisconnect(i,j) is the set  of  disconnect(j)
   events issued by user i. *)
   ∀ x← user(i)|disconnect
     x.msg= j #> x← pdisconnect(i,j);
   ∀ x← pdisconnect(i,j)
     x← user(i)|disconnect ^ x.msg= j;

   (* l ≤ i ≤ n, l ≤j ≤ n, i<>j *)

(* connected(i+j) is the coordinate event enabled
   by pconnect(i,j) and pconnect(j,i). *)
   coop(pconnect(i,j), pdisconnect(i,j),
        connected(i+j));

   (* l ≤ i < j ≤ n *)
(*  disconnected(i+j)  is  the  coordinate  event
   enabled        by       disconnect(i,j)       and
   pdisconnect(j,i). *)
   coop (pdisconnect(i,j), pdisconnect(i,j),
         disconnected(i+j);

   (* l ≤ i < j ≤ n *)

(* connected and disconnected run alternately *)
```

```
   ∀ x< connected(i+j), y< disconnected(i+j)
        (ord(x)= ord(y) #> x-> y) ^
        (ord(x)= ord(y)+ 1 #> y-> x);

      (* 1 ≤ i < j ≤ n *)

(* One can only send or receive messages when the
   link is connected. *)
   ∀ s< user(i)|send, r< user(j)|receive
     (s.to= j ^ r.from= i
      #> ∃ x< connected(i+j)
           x-> s ^ x-> r ^
           ~(∃ y< disconnect(i+j) ^
                        (x-> y-> s)  v  (x->  y->
r))

     (* 1 ≤ i < j ≤ n *)
(* Once the link is connected, it is  a  reliable
   transmission system. *)

(* s(i,j) is the set of send events  to   user  j
   issued by user i *)
   ∀ x< user(i)|send
     x.to= j #> x< s(i,j);
   ∀ x< s(i,j)
     x< user(i)|send ^ s.to= j;

(* r(i,j) is the set of receive events by user  j
   from  user i. *)
   ∀ x< user(i)|receive
     x.from= i #> x< r(i,j);
   ∀ x< r(i,j)
     x< user(i)|receive ^ x.from= i;

(* A reliable transmission system from s(i,j)  to
   r(i,j). *)
   RT11(s(i,j), r(i,j));
   RT12(s(i,j), r(i,j));
   RT13(s(i,j), r(i,j));
   RT14(s(i,j), r(i,j));
   RT15(s(i,j), r(i,j));
   ∀ x< s(i,j), y< r(i,j)
     x=> y #> x.data= y.data;

End behavior

End system.
```

5.6. The Expressive Power of CBS

In this section, we show that CBS is at least as powerful as Petri nets by translating each basic construct in Petri nets into CBS. The following are considered to be the basic constructs in Petri nets: Direct nets (Figure 5.5.), Concurrent enable nets (Figure 5.6.), Free-choice nets (Figure 5.7.) Coordinate nets (Figure 5.8.) and Multiplexor nets (Figure 5.9.).

Coordinate nets were discussed in the previous section. Direct nets are just reliable transmission systems and multiplexor nets are just multiplexor systems; both were discussed in chapter 2. We specify free-choice nets and concurrent enable nets in this section.

5.6.1. Free-choice Nets

A _Free-choice net_ has a single inport A and two outports B and C. Depending on some factors which are unknown or not modeled in the system, a message input to A is directed to B or C freely. It is specified in CBS as follows:

```
System free-choice-net (A: inport;
                        B: outport;
                        C: outport);

   Degree is 1;

   Behavior
```

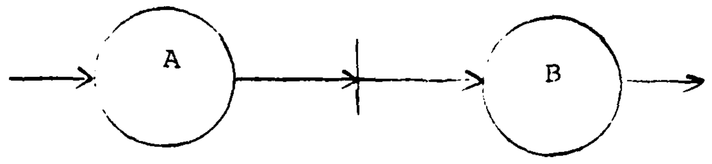


Figure 5.5. A Direct Net

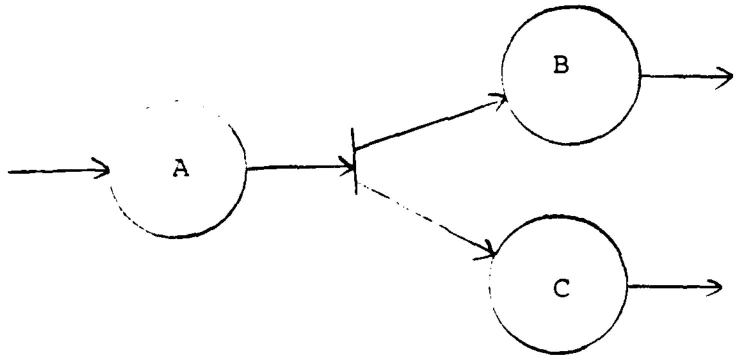


Figure 5.6. A Concurrent Enabling Net

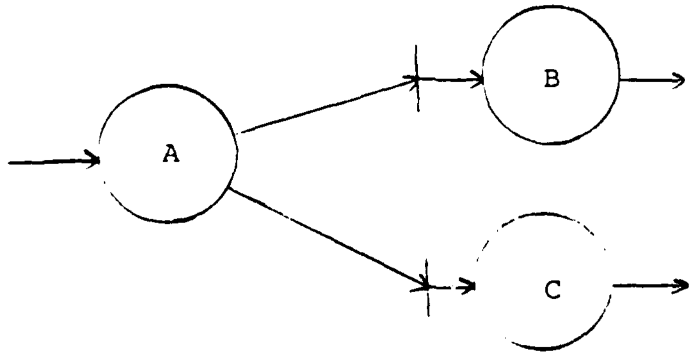


Figure 5.7. A Free Choice Net

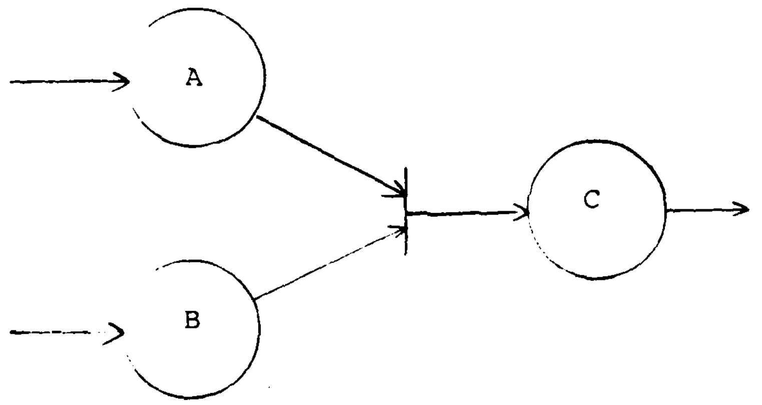


Figure 5.8. A Coordination Net

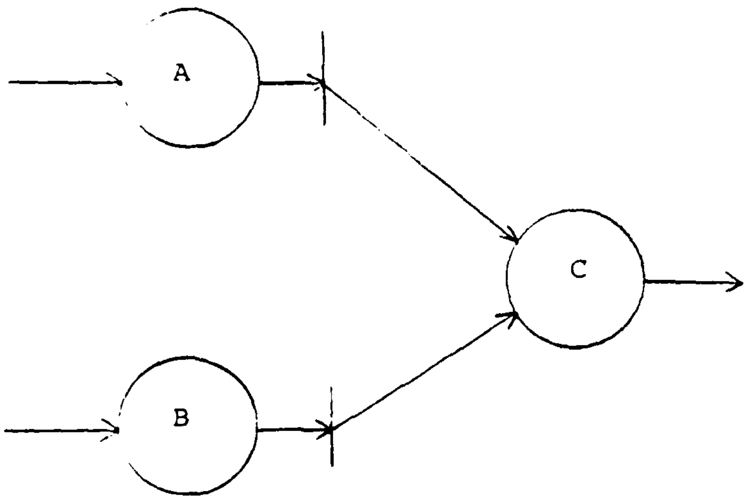


Figure 5.9. A Multiplexor Net

```
(* RT11 *)
  ∀ a< A
    ( ∃ b< B a=> b) v ( ∃ c< C a=> c);

(* RT12 *)
  (∀ b< B ∃ a< A a=> b) ^
  (∀ c< C ∃ a< A a=> c);

(* RT13 *)
  ∀ b< B, c< C, s< SYS, e< ENV
    (s=> b v s=> c #> ∃ a< A a=> s) ^
    (e=> b #> ∃ a< A e=> a=> b) ^
    (e=> c #> ∃ a< A e=> a=> c);

(* RT14 *)
  ∀ a< A, b1, b2< B, c1, c2< C
    (a=> b1 ^ a=> b2 #> b1= b2) ^
    (a=> c1 ^ a=> c2 #> c1= c2) ^
    ~(a=> b1 ^ a=> c1);

(* RT15 *)
  ∀ a1, a2< A, b1, b2 < B, c1, c2< C
    (a1=> b1 ^ a2=> b2
     #> (a1= a2 ^ b1= b2) v
        (a1-> a2 ^ b1-> b2) v
        (a2-> a1 ^ b2-> b1)) ^
    (a1=> c1 ^ a2=> c2
     #> (a1= a2 ^ c1= c2) v
        (a1-> a2 ^ b1-> b2) v
        (a2-> a1 ^ b2-> b1);

End behavior;

End system.
```

Note that the "Selector Systems" specified in Chapter 2 using EBS is more general than free-choice net specified using Petri nets because there is no way to specify the "selection condition" using Petri nets.

5.6.2. Concurrent Enable Nets

A concurrent enable net has a single inport A and two outports B and C. A message in A is directed both to B and

C concurrently. Here the word "concurrent" has a much weaker meaning than before. It means simply that two events happen "altogether" and "consecutively". By "altogether" we mean if one event happens then the other will happen. By "consecutive" we mean that no other event can happen in between these two events. The "altogether" property is specified as follows:

```
(* B and C happen altogether *)
   ∀ a< A
     (∃ b< B a=> b) ^ (∃ c< C a=> c);
```

The "consecutive" property is specified as follows:

```
(* B and C happen consecutively *)
   ∀ a< A, b< B, c< C
     (a=> b ^ a=> c)
     #> ~(∃ d b-> d-> c v c-> d-> b);
```

The other behaviors are almost the same as if A sends messages reliably both to B and to C.

Thus, we conclude that CBS is at least as expressive as Petri nets and is able to specify the properties such as data-related properties and event priorities which are not easily specified using Petri nets.

5.7. The Semantics of CBS

An event coordination specification can be translated into EBS in a rather straightforward way.

5.7.1. The Set Enables Relation =>

In terms of EBS, the semantics of the expression S1 => S2 of two event set S1 and S2 can be specified as follows:

$$((\forall\, e \in S1\ e \in CE)\ \#\!>\ (\forall\, e \in S2\ e \in CE)\ \wedge$$
$$(\forall\, e1 \in S1,\ e2 \in S2\ e1 \rightarrow e2)$$

The first part and the second part of these semantics are derived from the definitions "if every event in S1 happens then every event in S2 should happen eventually" and "every event in S1 happens before any event in S2", respectively.

5.7.2. The Collectly Enables Relation *>

For a specification of degree n, the semantics of the expression S1 *> S2, in terms of EBS, is

$$\begin{aligned}
&(S1 \Rightarrow S2)\ \wedge \\
&(\forall\, e \in S1\ \sim(e \Rightarrow S2))\ \wedge \\
&(\forall\, e1,\ e2 \in S1\ \sim(\{e1,\ e2\} \Rightarrow S2))\ \wedge \\
&\ \ldots \\
&(\forall\, e1,\ \ldots,\ e(n-1) \in S1\ \sim(\{e1,\ \ldots,\ e(n-1)\} \Rightarrow S2)
\end{aligned}$$

The first part specifies that S1 should enable S2, and the second part specifies that any proper subset of S1 is not sufficient to enable S2.

5.8. The Structure Specification and Verification in CBS

The syntax of structure specification in event coordination is the same as that in EBS.

5.8.1. A Comparator Circuit

A comparator (see Figure 5.10.) is a device that has two input arguments, say A and B, and three output arguments, GT, EQ and LT. The device compares the messages from A and B; if the message from A is greater than, equal to, or less than that from B then a message will be sent to GT, EQ or LT respectively. The behavior of this comparator device can be specified as follows:

```
System CMP (A: inport;
            B: inport;
            GT: outport;
            EQ: outport;
            LT: outport);

   Degree is 2;

   Behavior

      (* CP10: A single event from either A or  B  is  not
         good enough to enable GT, EQ or LT. *)
         AD10 (A, B, GT);
         AD10 (A, B, EQ);
         AD10 (A, B, LT);

      (* CP11: If a is greater than, equal to or less than
         b  then  one message will be sent to GT, EQ or LT
         respectively. *)
         ∀ a← A, b← B
           ord(a)= ord(b) #>
           (a.msg > b.msg #> ∃ g← GT {a, b} => g) ^
           (a.msg = b.msg #> ∃ e← EQ {a, b} => e) ^
           (a.msg < b.msg #> ∃ l← LT {a, b} => l);
```

```
(* CP12: GT, EQ or LT receives one message only when
   a is greater than, equal to or less than b
   respectively. *)
   (∀ g< GT ∤ a< A, b< B
      a.msg > b.msg ^ {a, b} => g) ^
   (∀ e< EQ ∤ a< A, b< B
      a.msg = b.msg ^ {a, b} => e) ^
   (∀ l< LT ∤ a< A, b< B
      a.msg < b.msg ^ {a, b} => l);

(* CP13: No internally or externally generated mes-
   sages. *)
   AD13 (A, B, GT);
   AD13 (A, B, EQ);
   AD13 (A, B, LT);

(* CP14: No duplicated messages. *)
   AD14 (A, B, GT);
   AD14 (A, B, EQ);
   AD14 (A, B, LT);

(* CP15: No out of order messages *)
   AD15 (A, B, GT);
   AD15 (A, B, EQ);
   AD15 (A, B, LT);

End behavior

End system.
```

An intuitive implementation of this comparator circuit would be to subtract b from a; if the result is positive, zero, or negative then enable GT, EQ or LT respectively. This idea leads to a design that is composed of a <u>full subtractor</u> followed by a <u>selector</u> (see Figure 5.11.), and is specified as follows:

```
System CMP (A: inport;
            B: inport;
            GT: outport;
            EQ: outport;
            LT: outport);

   Degree is 2;
```

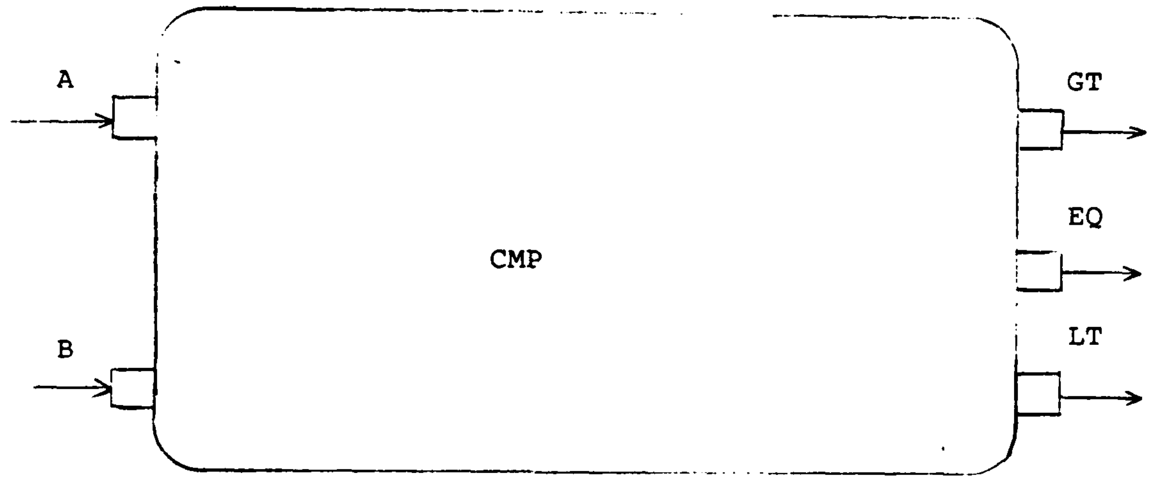


Figure 5.10. A Comparator Circuit

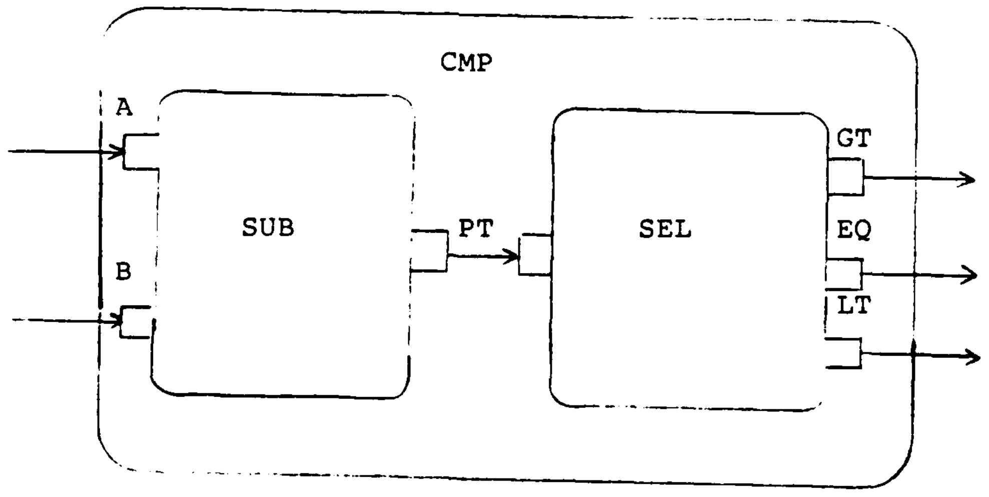


Figure 5.11. An Implementation Structure of the Comparator: a Substractor Followed by a Selector

```
Structure

   Subsystem SUB (A: inport;
                  B: inport;
                  TP: outport);

      Degree is 1;

      Behavior

         (* Control Parts *)
            coop (A, B, TP);

         (* DP: Data parts *)
            ∀ a< A, b< B, t< TP
              {a, b} => t #> t.msg = a.msg - b.msg;

        End behavior;

     End subsystem;

     Subsystem SEL (TP: inport;
                    GT: outport;
                    EQ: outport;
                    LT: outport);

        Degree is 2;

        Behavior

           (* RT11: If t is greater than, equal  to  or
              less  than zero then a message is sent to
              GT, EQ or LT respectively. *)
              ∀ t< TP
                (t.msg > 0 #> ∃ g< GT t=> g) ^
                (t.msg = 0 #> ∃ e< EQ t=> e) ^
                (t.msg < 0 #> ∃ l< LT t=> l);

           (* RT12: GT, EQ or LT  receive  the  message
              only  when t is greater than, equal to or
              less than zero respectively. *)
              (∀ g< GT ∃ t< TP
                 t.msg > 0 ^ t=> g) ^
              (∀ e< EQ ∃ t< TP
                 t.msg = 0 ^ t=> e) ^
              (∀ l< LT ∃ t< TP
                 t.msg < 0 ^ t=> l);

           (* RT13, RT14 and  RT15:  No  internally  or
              externally  generated messages; No out of
              order or duplicated messages. *)
```

```
                RT13 (TP, GT);
                RT13 (TP, EQ); RT13 (TP, LT);
                RT14 (TP, GT);
                RT14 (TP, EQ); RT14 (TP, LT);
                RT15 (TP, GT);
                RT15 (TP, EQ); RT15 (TP, LT);

            End behavior;

         End subsystem;

         Network

            connect (SUB.TP, SEL.TP) == CMP.TP;

         End network;

         Interface

            CMP.A == SUB.A;
            CMP.B == SUB.B;
            CMP.GT == SEL.GT;
            CMP.EQ == SEL.EQ;
            CMP.LT == SEL.LT;

         End interface;

      End interface.
```

5.8.2. The Verification of the Comparator

The verification that the structure specification of the comparator satisfies its behavior specification is given in the following theorem.

Theorem 5.7.

The structure specification of CMP is correct according to its behavior specification.

Proof

We prove only the correctness of the behavior of the outport GT; those of outports EQ and LT can be carried out in much the same way.

```
(* CP10 *)
        (1) g< GT
    Assume there exists a0< A such that
        (2) a0=> g
    Then there exists t0< TP
        (3) a0=> t0=> g
        (4) (3) contradicts to AD10 (A,B,TP)
    Thus
        (5) ∀ a< A ~(a=> g)
     Similar proof can be given for b< B.

    (* CP11 *)
        (1) a< A, b< B
        (2) ord(a) = ord(b)
        (3) a.msg > b.msg
            there exists t0< TP
        (4) {a, b} => t0 ^ t0.msg = a.msg- b.msg
        (5) to.msg > 0  ...(3), (4)
            there exists g0< GT
        (6) t0=> g0

    (* CP12 *)
        (1) g< GT
            there exists t0< TP
        (2) t0=> g ^ t0.msg > 0
            there exist a0< A, b0< B
        (3) {a0, b0} => t0 ^ t0.msg= a0.msg- b0.msg
        (4) {a0, b0} => g ^ a0.msg > b0.msg ...(2), (3)

    (* CP13 *)
    (* We prove only the property that
       ∀ g< GT, S in SYS
         S *> g #> ǂ a< A, b< B
                     {a, b} *> S
       other properties can be proved in much  the  same
       way. *)
        (1) g< GT
        (2) S in SYS(CMP), S *> g
        (3) S in SYS(SUB) v S in SYS(SEL)
    case
        (a) S in SYS(SUB)
            there exists t0< TP
            (4) S *> t0=> g
                there exist a0< A, b0< B
            (5) {a0, b0} *> S
        (b) S in SYS(SEL)
        there exists t0< TP
        (6) t0=> S
            there exist a0< A, b0< B
        (7) {a0, b0} *> t0
        (8) {a0, b0} *> S

(* CP14 *)
   (1) a< A, b< B, g1, g2< GT
   (2) {a, b} => g1 ^ {a, b} => g2
       there exist t1, t2< TP
   (3) {a, b} => t1=> g1 ^ {a, b} => t2=> g2
   (4) t1= t2  ...(3)
   (5) g1= g2  ...(3), (4)

(* CP15 *)
   (1) a1, a2< A, b1, b2< B, g1, g2< GT
   (2) {a1, b1} => g1 ^ {a2, b2} => g2
       there exist t1, t2< TP
   (3) {a1, b1} => t1=> g1 ^ {a2, b2} => t2=> g2
   (4) a1= a2 v a1-> a2 v a2-> a1
 case
   (a) a1= a2
        (5) b1= b2 ^ t1= t2
        (6) g1= g2
   (b) a1-> a2
        (7) b1-> b2 ^ t1-> t2
        (8) g1-> g2
   (c) a2-> a1
        (9) b2-> b1 ^ t2-> t1
        (10) g2-> g1
     Q. E. D.
```

CHAPTER 6

CONCLUSIONS AND FURTHER RESEARCH

6.1. Comparisons with Other Approaches

6.1.1. Finite-State Machine Models

Finite-State Machines, such as Petri Nets, AFFIRM, SPECIAL and Right-Synchronization Controllers [CON79], have been widely used to specify and verify concurrent systems. Each processing unit in the system is represented as a state diagram, with a machine state corresponding to each discernible state of the unit. State transitions in the model reflect transitions in the actual system. Reachability analyses are used to detect possible anomalies in the system behavior. Since the theory behind the finite state-machine model is well-known, tools to simulate the system operations or to analyze the system behavior automatically can be built without much difficulty. Exploiting the AFFIRM term-rewriting system, it has been quite successful at automating analysis and proof of protocols [SCH81].

There are several drawbacks to the state-machine approach however. As the number of possible states increases, analyzing all possible interaction becomes infeasible. Furthermore, rigorous analysis of possible behavior, when practical, guarantees the safety of the system but does not guarantee the liveness of the system. Liveness properties, such as that a message will eventually be received or that each request will eventually be served, are requirements that certain transitions (events) eventually take place. Liveness requirements are difficult or impossible to state and prove using state-machine specifications.

6.1.2. Operational Models

A more general problem using an operational model such as PAISLEY [ZAV81], or GYPSY [GOO79], as a definition tool is the difficulty of separating requirements from expedient. While one would like to state requirements that the system must meet and leave the method of achievement to the implementor, an operation model specifies the requirements by giving an abstract implementation. There is no indication of what aspects of the model are to be rigorously followed and what aspects merely illustrate functionality. In the case of an operation model for a sequential program, this causes no particular problem. Any implementation with identical (isomorphic) input/output behavior is acceptable. For a concurrent system, it becomes more difficult to identify those aspects of data and control that can be modified without affecting system behavior. A specification for a distributed system should be even more explicit as to what is necessary for correct system operation.

6.1.3. Algebraic/Axiomatic Approaches

For sequential programs, algebraic/axiomatic specification techniques provide the abstraction necessary to state properties of a program without stating the function itself. Unfortunately, standard algebraic/axiomatic techniques for defining properties of sequential programs are not suitable for concurrent programs. Temporal properties, such as concurrency, mutual exclusion, and process priority are difficult or impossible to specify in algebraic/axiomatic specifications. Temporal logic, an extension of classical logic, tries to provide the capability of specifying these temporal properties.

6.1.4. Temporal Logic Approaches

Temporal logic was first introduced by Pnuli in 1975 as an adaptation of a classical logic suitable for defining the semantics of computer programs. When applied to programs, the meaning of a computation is taken to be the sequence of states resulting from program execution. A distributed system is modeled by multi-tasking on a single processor. Using temporal operations, one reasons about the execution state sequence resulting from interleaved execution of each process. A set of temporal logic axioms specify properties that must be true for all state sequences resulting from system execution.

The fundamental temporal operator is the unary operator $\square$, pronounced "henceforth". Taking an explicit execution-state sequence model of the system, with i being an index to the current state in the sequence, $\square P$ is defined as:

$$\square P = \forall t \geq i \ P(t)$$

Loosely speaking, $\square P$ asserts that P is true in the present state and will be true for the remainder of the computation.

The dual of $\square$ is $\diamond$, pronounced "eventually", with the interpretation

$$\diamond P = \sim\square\sim P$$

This asserts that either now or at some future point in the computation P will be true. In terms of explicit execution state sequence, $\diamond P$ is $\exists t \geq i \ P(t)$

With these temporal operators, many properties of systems can be stated. $\square I$ states that I is an invariant throughout the system execution. To state that a property P always causes a property Q to occur subsequently, one writes $\square (P \Rightarrow \diamond Q)$. To assert that a property P is satisfied infinitely often, one writes $\square\diamond P$. This says that, from every point in the computation, there is a future point at which P will be true.

Unfortunately, there are several problems in using temporal logic to specify and verify distributed systems. First, the processes in a distributed system are autonomous and run independently. Modeling the computation of a distributed system by the interleaved execution of each process makes the verification unnecessarily complicated. Global invariants have to be proved true for all possible sequences from interleaved execution of each process, even though the computation of a particular process may be independent of the invariants.

Second, global invariants that should be true throughout the computation, rather than merely input/output relations, are stated as the behavior specification of a distributed system. Though invariants facilitate implementation verification, from the user's viewpoint, they are difficult to specify, understand and are less intuitive than input/output relations.

Third, the time concept in temporal logic is too implicit for users to specify the time-ordering relation among events. The time order relation among states in a computation is implicitly expressed by the temporal operators $\Box$ and $\Diamond$; there is no explicit time variable in the specification. Even the time "now" (an index to the current state in the computation sequence) necessary to the interpretation of temporal operators is implicitly defined from the context of the operators in the specification. For example, the time "now' of $\Box P$ when it comes by itself, refers to the system initialization; while the time "now" of $\Diamond Q$ in $\Box (P \# > \Diamond Q)$ refers to the time of a state when P is true, after the system initialization. Such a context sensitive interpretation of the temporal operators makes a specification difficult to follow, especially when many temporal operators are used in the specification.

6.1.5. The Actor Model and Other Event-Based Models

Our event-based model is based on several pioneering works in the past few years. With some modifications, the basic properties of events in our model were adopted from the actor theory invented by Carl Hewitt [HEW77] and his collaborators [BAK78]. Actors, messages, and events are fundamental in the model. Actors interact with one another by exchanging messages. The messages themselves are actors. The receiving and the processing of a message by the target actor are events, and these events are the basic steps in the actor model of computation. Each event is atomic and instantaneous. Several programming language [ATK80, REU80] are built on this model.

Our model differs from the actor model in two major aspects. First, the sending, receiving and processing of a message are considered as individual events in our model. Since the time duration between the sending, the receiving and the processing of a message is non-zero in a distributed system computation, the event in our model is more accurately represents a distributed system computation and is more intuitive from a user's viewpoint. Second, we have aimed at a behavior specification technique rather than a programming tool. Thus, the implementation-oriented constructs such as process, while-loop, and assignment-statement appearing in actor languages are not part of our language.

The partial ordering relation, the "precedes" relation, used as the time ordering to specify the behavior of distributed systems was first introduced by Irene Greif [GRE77]. The basic ideas in our Transaction-Based Specification Language were trigged by the work of Mark Laventhal [LAV78].

There are two major differences between our work and theirs. First, we are able to specify both control-related and data-related properties in EBS; only control-related properties can be specified in the other languages. Second, the introduction of the causality relation, the "enables" relation, in EBS, facilitates the specification of liveness properties in a way more intuitive than those using the time ordering relation directly.

6.1.6. Trace Approaches

The notion of traces is used in the specifications and verifications of networks of processes by Misra & Chandy [MIS81], and Zhoa & Hoare [ZHO81]. The theory of traces of communicating sequential processes is due to Hoare [HOA78b]. A trace of the behavior of a process is defined as "the recorded sequence of communications in which the process engages up to some moment in time" [ZOA81]. In terms of EBS, a trace is simply a sequence of interfacial events.

The specifications of system computations are expressed in traces exclusively; thus, the entire proof technique deals only with propositions on traces. The notations for sequences such as "concatenation of sequences", "prefix (initial) of a sequence", "prefix closure of a sequence" and "the length of a sequence", are basic to the trace specification language.

There are several deficiencies in the trace approach. First, describing the behavior of a distributed system by a trace dictates a total ordering of events and the existence of an implicit global clock, which is not generally a requirement in distributed systems. Second, since notations for sequences are used exclusively, trace specifications are awkward in expressing properties whose data structures are not well-defined sequences. Typical examples are those properties of unreliable transmission systems that may lose, duplicate and reorder messages. To overcome such shortage, informal specifications such as "monotone increasing sequence" and "relative primes" [MIS81] in describing the behavior of a sieve, were adopted. Third, a rather serious deficiency is that the liveness properties are not usually specified and verified using the trace notion directly.

In comparison, events in EBS are only partially ordered; no assumption of the existence of a global clock is made. The concept of events is more elementary than that of traces (sequences of events); consequently, some properties that can be specified in EBS easily can only be expressed in traces with difficulty. The "liveness" properties can be specified directly by the enables relation => in EBS.

6.2. Further Research

We have proposed an event-based model and demonstrated its application to the specification and verification of several classes of distributed systems. Based on our work, there are several research topics that are deserving of further effort.

Since we are dealing with first-order logic in EBS, it is possible that the design verification be processed mechanically if some automatic theorem-prover is available. Since the design of EBS has been aimed at human understandability and human verifiability rather than machine executability, we expect that some "syntactic sugar" should be added to make EBS machine executable.

In a top-down hierarchical design, a distributed system can be described by the behavior specification; then the specification can be decomposed into a set of subsystems communicating via the connection links, described by the structure specification. Correctness of a design can be proved by checking the consistency between the behavior specification and structure specification of a system. Each subsystem can be again decomposed into sub-subsystems. Again, the correctness of the detailed design can be checked according to its subsystem specification. The design and verification processes go so on so forth. In this way, at the end of the whole design phase, we can make sure that the design is correct even before the system is actually implemented.

When the implementation phase is begun, a particular distributed programming language should be adopted to code the design and then the implementation should be proved correct with respect to the design specification. Unfortunately, we have not dealt with implementation verification in this thesis.

Two major steps in verification are (1) specifying the event semantics for the distributed programming language, and (2) devising a set of inference rules for the language in terms of events. A step, before the verification and even more important than the verification, is the selection of a syntactically and semantically sound distributed programming language. All of the three steps are not trivial and are deserving of major efforts.

We have applied the Transaction-Based Specification Language to specify the properties such as mutual exclusion, concurrency, in transaction oriented system. However, it is still not obvious how to specify some important properties such as crash-recovery, or to verify the famous protocols such as 2PC (two phase commit protocol) and 2PL (two phase locking protocol) in distributed database systems. Allowing user-definable events inside a transaction in addition to the system-defined events (i.e., the BEGIN and END events of a transaction) seems to be a necessary step in extending TBS to specify, and verify such properties. Research in this area is therefore desirable.

6.3. Conclusions

In summary, both the behavior and structure specifications based on event model are (1) formal: using partial ordering relations and first order predicate calculus; (2) minimal: orthogonal properties are specified separately; (3) extensible: new requirements can be added without changing the original specification; and (4) complete: interesting properties in distributed systems can be specified.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and structure specification of a system. Both "safety" and "liveness" properties can be specified and verified.

Moreover, since the specification technique defines the orthogonal properties of a system separately, each of them can be verified independently. Thus, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

In addition to having most of the desirable features of a specification technique, EBS represents the concept of time by a partial ordering relation of events and represents concurrency by the lacking of ordering between events. This makes EBS a more accurate model for distributed systems, which are inherently concurrent, asynchronous, and nondeterministic.

7. References

[ALF77] Alford, M. W. et al "Requirement Development using SREM Technology" Vol. 1, Technical Report CDRL COOH, Oct. 1977

[AND78] Andler, S. "Synchronization Primitives and the Verification of Concurrent Programs" Proc. 2nd international Symp. on Operating Systems, IRIA Le Chesnay, France Oct. 1978

[AND79] Andler, S. "Predicate Path Expressions" Proc. 6th Annual ACM Symp. on Principles of Programming Languages: 226- 236, San Antonio, Texas, Jan. 1979

[APT80] Apt, K. R., Frances, N. and Roever, W. P. "A Proof System for Communicating Sequential Processes" ACM TOPLAS 2(3): 359-385, July 1980

[ATK80] Atkinson, R. R. "Automatic Verification of Serializers" Ph. D. Dissertation, MIT, Mar. 1980

[BAK78] Baker, H. J. "Actor Systems for Real-Time Computation" MIT/LCS/TR-197, Ph. D. Dissertation Mar. 1978

[BAL79] Balzer, R. and Goldman, N. "Principles of Good Software Specification and Their Implications for Specification Languages" Proc. of IEEE Symp. on Specification of Reliable Software, 1979

[BAR69] Bartlett, K. A. et al. "Note on Reliable Full Duplex Transmission on Half Duplex Links", CACM

12(5): 260-261, May 1969

[BER80] Bernstein, A. J. "Concurrency Control in a System for Distributed Databases (SDD-1)" ACM TODS 5(1): 18-51, Mar. 1980

[BOC78] Bochmann, G. V. "Finite-State Description of Communication Protocols" Computer Networks 2: 361-372, 1978

[BOS80] Bos, J. V. D. "Comments on ADA Process Communication" ACM SIGPLAN NOTICE 15(6): 77-81, June 1980

[BRE79] Bremer, J. "A New Approach to Protocol Design and Validation" IBM Technical Report RC8018, Dec. 1979

[BRI74] Brinch Hansen, P. "Structured Multiprogramming" CACM 15(7): 574-578, July 1974

[BRI78] Brinch Hansen, P. "Distributed Processes: a Concurrent Programming Concept" CACM 21(11): 934-941, Nov. 1978

[BRI81] Brinch Hansen, P. "The Design of EDISON" Software-Practice and Experience 11: 363-396, 1981

[BRY78] Bryant, R. E. and Dennis, J. B. "Concurrent Programming" MIT/LCS/TR-115, 1978

[CAM74] Campbell, R. H. and Habermann, A. N. "The Specification of Process Synchronization by Path Expressions" Lecture Notes in Computer Science 16: 89-102, Springer Verlag, Heidelberg, 1974

[CAM80] Campbell, R. H. and Rolstad. R. B. "An Overview of Path Pascal´s Design" ACM SIGPLAN NOTICE 15(9):

13-24, Sept. 1980

[CHE81a] Chen, B. and Yeh, T. Y. "Event-Based Behavior Specification of Distributed Systems", Proc. of IEEE Symp. on Reliability in Distributed Software and Database Systems, July, 1981, Pittsburgh, Pennsylvania.

[CHE81b] Chen, B. and Yeh, T. Y. "Behavior Specifications of Distributed Systems", submitted to IEEE Transactions on Software Engineering

[CHE81c] Chen, B. "Formal Specification and Verification of Distributed Systems", Submitted to the 3rd International Conference on Distributed Systems, Florida, 1982

[CON79] Conner, M. H. "Process Synchronization by Behavior Controllers" Ph. D. Dissertation, University of Texas at Austin, Aug. 1979

[COU71] Courtois, P. J., Heymans, F. and Parnas, D. L. "Concurrent Control with ´Readers´ and ´Writers´" CACM 14(10):667-668, Oct. 1971

[DAN80] Danthine, A. A. S. "Protocol Representation with Finite-State Models" IEEE Transactions on Communications COM-28(4): 632-642, April 1980

[DIJ68] Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control" CACM 9(9): 569, Sept. 1968

[DIJ72] Dijkstra, E. W. "Co-operating Sequential

Processes" Academic Press 1972

[END72] Enderton, H. B. "A Mathematical Introduction to Logic" Academic Press, 1972, Chapter 2

[GOO79] Good, D. I. et al. "Principles of Proving Concurrent Programs in GYPSY" University of Texas at Austin, Technical Report ICSCA-CMP-15, Jan. 1979

[GRA78] Gray, J. "Notes on Database Operating Systems" Lecture Notes in Computer Science 60: 393-484, Spring Verlag, New York, 1978

[GRA79] Gray, J. "A Discussion of Distributed Systems" IBM Technical Report RH2699, Sept. 1979

[GRE77] Greif, I. "A Language for Formal Problem Specification", CACM 20(12) : 931-935, Dec. 1977

[HAB72] Habermann, A. N. "Synchronization of Communicating Processes" CACM 15(3): 161-176, Mar. 1972

[HAI80] Hailpern, B. and Owicki, S. "Verifying Network Protocols Using Temporal Logic" In Trends and Applications 1980: Computer Network Protocols, IEEE Computer Society, May 1980

[HEW77a] Hewitt, C. and Baker, H. J. "Laws For Communicating Parallel Processes", IFIP 987-992, 1977

[HEW77b] Hewitt, C. and Baker, H. J. "Actors and Continuation Functionals" MIT/LCS/TR-194, 1977

[HOA74] Hoare, C. A. R. "Monitors: an Operating System Structuring Concept" CACM 17(10): 539-557, Oct. 1974

[HOA78a] Hoare, C. A. R. "Communicating Sequential Processes" CACM 21(7): 123-134, July 1978

[HOA78b] Hoare, C. A. R. "A Model for Communicating Sequential Processes" Computer Lab., Oxford University, Dec. 1978.

[HOW76] Howard, J. H. "Proving Monitors" CACM 19(5): 273-279, May 1976

[ICH79] Ichbiah, J. D. et al. "Rationale for the Design of the ADA Programming Language" ACM SIGPLAN NOTICE 14(6): Chapter 11, June 1979

[KEL76] Keller, R. M. "Formal Verification of Parallel Programs" CACM 19(7): 371-384, July 1976

[LAMB80] Lampson, B. W. and Redell, D. D. "Experience with Process and Monitors in MESA" CACM 23(2): 105-117, Feb. 1980

[LAM77] Lamport, L. "Proving the correctness of Multiproces Programs" IEEE TOSE SE-3(2): 125-134, Mar. 1977

[LAM78a] Lamport, L. "The Implementation of Reliable Distributed Multiprocess Systems" Computer Networks 2(2): 95-114, May 1978

[LAM78b] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", CACM 21(7) : 558-565, July 1978.

[LAM80a] Lamport, L. "´Sometimes´ is Sometimes ´Not Never´" Proc. of the ACM Symp. on Principles of

Programming Languages Jan. 1980

[LAM80b] Lamport, L. "The ´Hoare Logic´ of Concurrent Programs" Acta Informatica 14: 21-37, 1980

[LAU79] Lauer, H. C. "On the Duality of Operating System Structures" Operating Systems Review 13(2): 3-19, Apr. 1979

[LAV78] Laventhal, M. S. "Synthesis of Synchronization Code For Data Abstractions" MIT/LCS/TR-203 Ph.D. Dissertation June 1978.

[LAV79] Laventhal, M. S. "Synchronization Specification for Data Abstractions" Proc. of IEEE Symp. on Specification of Reliable Software: 119-125, 1979

[LIS76] Liskov, B. H. "An Appraisal of Program Specification" MIT Technical Report, July 1976

[LIS77] Liskov, B. H. and Zills, S. "An Introduction to Formal Specifications and Data Abstractions" In Current Trends in Programming Methodology Vol(1): 1-33, Yeh, R. T. Editor, Prentice Hall 1977

[MIS81] Misra, J. and Chandy, K. M. "Proofs of Networks of Processes" IEEE Transactions on Software Engineering SE 7(4): 417- 426, July 1981.

[NIL80] Nilsson, N. J. "Principles of Artificial Intelligence" McGraw-Hill Co., 1980

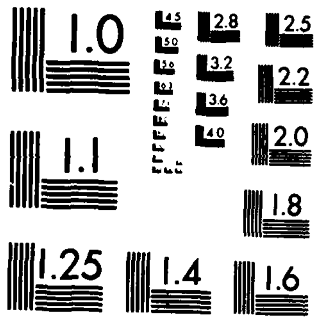
[OWI76] Owicki, S. and Gries, D. "Verifying Properties of Parallel Programs: An Axiomatic Approach" CACM 19(5): 270-285, May 1976

AD-A128 629 EVENT-BASED SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEMS(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE B CHEN 1982 AFOSR-TR-83-0388

3/3

UNCLASSIFIED F49620-80-C-0001 F/G 9/2 NL

END
DATE
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0

[OWI80] Owicki, S. and Lamport, L. "Proving Liveness Properties of Concurrent Programs" Stanford University, Working Draft, Oct. 1980

[PET77] Peterson, J. L. "Petri Nets" ACM Computing Survey, 9(3): 223-253, Sept. 1977

[RAM79] Ramamoorthy and So "Software Requirements and Specifications: Status and Perspective" IEEE Tutorial, Distributed System Design, 1979

[RAO80] Rao, R. "Design and Evaluation of Distributed Communication Primitives" University of Washington, Seattle, TR80-04-01, April 1980

[REU80] Reuveni, A. "The Event-Based Language and Its Multiprocessor Implementations" Ph. D. Dissertation, MIT, Jan. 1980

[RID74] Riddle, W. "The Modeling and Analysis of Supervising Systems" Ph. D. Dissertation, Stanford University, 1974

[RID79] Riddle, W. E. et al. "Behavior Modeling During Software Design" IEEE TOSE 1978

[ROB77] Robinson, L. and Roubine, D. "SPECIAL: A Specification Language and Assertion Language" Technical Report CSL-46, Stanford Research Institute, 1977

[SCH81] Schwartz, R. L. and Melliar-Smith, P. M. "Temporal Logic Specification of Distributed Systems" Proc. of IEEE 2nd Internal Conf. on Distributed Computing Systems: 446-454, Paris, France, April

1981

[SHA78] Shaw, A. C. "Software Descriptions with Flow Expressions" IEEE TOSE SE-4(3): 242-254, May 1978

[SHA79] Shaw, A. C. "Software Specification Languages Based on Regular Expressions" University of Washington, TR-35, June 1979

[STE76] Stenning, N. V. "A Data Transfer Protocol" Computer Networks 1(2) : 99-110, Sept.1976.

[SUN79] Sunshine, C. "Formal Methods for Communication Protocol Specification and Verification", WD-335-ARPA/NBS, Working Draft, Rand Corporation, Sept. 1979.

[SVO79] Svobodova, L., Liskov, B. and Clark, D. "Distributed Computing Systems: Structures and Semantics" MIT/LCS/TR-215, Mar. 1979

[TOM80] Tompson, D. H. et al."Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models." Information Science Institute, 1980.

[YEH80] Yeh, R. T. and Zave, P. "Specifying Software Requirements", Proc. of IEEE, Oct. 1980.

[YON77] Yonezawa, Q. "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics" MIT/LCS/TR-191, Ph. D. Dissertation, Dec. 1977

[ZAV81] Zave, P. and Yeh, R.T. "Executable Requirements for Embedded System" 5th ICSE in San Diego,1981.

[ZHO81] Zhoa, C. C. and Hoare, C. A. R. "Partial Correctness of Communicating Sequential Processes" Proc. of IEEE, 2nd International Conference on Distributed Computing Systems, Paris, France, April 1981.

APPENDIX A: A SEMANTICS INTERPRETER

In this appendix, we will give a formal treatment of our Event-Based Specification Language. The semantics of EBS is interpreted in two different ways: a centralized processor interpretation and a multi-processor interpretation.

1.1. The First Order Predicate Language

The symbols in EBS can be categorized as follows:

I. Logical Symbols

(1) Parenthesis: (,)

(2) Sentential connective symbols: ^ (logical and), v (logical or), ~ (negation), #> (implication)

(3) Variables: x, y, z, ...

(4) Equality symbols: =

II. Parameters

(1) Quantifier symbols: ∀ (for every), ∃ (there exists)

(2) Predicate symbols: -> (precedes), => (enables), ∈ (belongs to)

(3) Function symbols: ord (ordinal function), ENV (environment event set), SYS (system event set), INT

(interface event set), MSG (message contents set)

(4) Constant symbols:

III. All the symbols from natural numbers.

A <u>Specification</u> is defined as follows:

<u>1</u>. <u>Expressions</u>

An expression is a finite sequence of symbols.

<u>2</u>.<u>Terms</u>

(1) Constant symbols and variables are terms.

(2) if f is a n-place function symbols and t1, t2, ..., tn are terms then f(t1, t2, ..., tn) is a term.

(3) An expression is a term only if it can be shown to be a term on the basis of clauses (1) and (2).

<u>3</u>. <u>Atomic</u> <u>Formulas</u>

If P is a n-place predicate symbols and t1, t2, ..., tn are terms then P(t1, t2, ..., tn) is an atomic formula.

<u>4</u>. <u>Well</u>-<u>Formed</u> <u>Formulas</u> (<u>wffs</u>)

(1) Every atomic formula is a wff.

(2) If w1 and w2 are wffs, and x is a variable then (~w1), (w1#> w2), (w1 ^ w2), (w1 v w2), (∀ x w1), and

($\exists$ x wl) are wffs.

5. Specifications

A specification is a wff.

1.2. A Centralized Processor Interpretation

By a centralized processor interpretation we mean the mapping of the partial ordering of events into a totally ordered implementation. That is, we assume that there is a global clock to order totally the universal event space of the whole computation. Since an event, by definition, is instantaneous, and time is continuous, the probability that two or more events happen exactly at the same time is zero. Thus, the totally ordered implementation is actually a totally ordered relation.

The basic data structure in the interpreter is a sequence, i.e., a totally ordered set of objects.

Definition A.1. Sequence

A sequence S is defined recursively as a finite set of objects called nodes. S is either empty (denoted by nil) or

(a) there is a specially designated node denoted by car(S); and

(2) the remaining nodes, form a sequence, denoted by cdr(S).

Definition A.2. Membership

A node e being a member in a sequence S is defined recursively as follows:

```
in(e, S) =
  IF (S= nil) THEN false
  ELSEIF (car(S)= e) THEN true
  ELSE  in(e, cdr(S))
```

Definition A.3. Sequence Number

The sequence number sn of a node in a sequence is defined recursively as follows:

```
sn(e, S)=
  IF (S= nil) THEN  0
  ELSEIF (car(S)= e) THEN  1
  ELSE (sn(e, cdr(S))+ 1)
```

When a node is not in a sequence, its sequence number is meaningless. However, some arbitrary number ("0", in the definition above) should be assigned to it, since only total functions are allowed within the first-order theory. Similar arguments are applied to some definitions below.

Definition A.4. History

A (computation) history is a sequence of events.

Definition A.5. Centralized Computation

A centralized computation is a 6-tuple C= <U, H, Q, A, F, P>, where

U is the universal set of elements (including event space, integers; data types etc.);

H is a history;

Q is a partition of H into sets SYS, ENV, IP1, ..., IPn, and OP1, ..., OPm;

A is a set of constant symbols;

P is a set of predicate symbols; and

F is a set of function symbols.

<u>Definition</u> <u>A</u>.<u>6</u>. <u>Interpreter</u>

An interpreter I is a function

I: V -> U,

mapping the set V of all variables into the universal set U of computation C.

We want to define the correctness of a computation C according to a specification S under an interpreter I, represented by the predicate correct(C, S, I).

The formal definition of correctness proceeds as follows:

First of all, we define an extended interpreter

J: T -> U

a function mapping from the set T of all terms into the universe U of the computation C. The idea is that J(t) should be the member of the universe U that is named by the term t. J is defined recursively as follows:

1. For each variable x, J(x)= I(x)

2. For each constant symbol a, J(a)= a

3. If t1, ..., tn are terms and f is an n-place function symbol, then

   J(f(t1, ..., tn))= f(J(t1), ..., J(tn))

Second, we define the substitution function as follows:

```
I(x|d)(y)=  I(y)   if y <> x
             d      if y = x
```

I(x|d) is the function which is exactly like I except at the variable x it assumes the value d.

<u>Definition</u> <u>A.7</u>. <u>Behavior</u> <u>Specification</u>

A (behavior) specification is a well-formed formula in the specification language.

Finally, the correctness of a centralized computation C according to a specification S under the interpretation I can be defined recursively as follows:

```
correct(C, S, I) =
  IF (S= nil) THEN  true
  ELSEIF (S= (S1 ^ S2))
         THEN  (correct(C, S1, I) ^
                correct(C, S2, I))
  ELSEIF (S= (S1 v S2))
          THEN (correct(C, S1, I) v
                correct(C, S2, I))
  ELSEIF (S= ~S1) THEN  ~correct(C, S1, I)
  ELSEIF (S= S1 => S2))
         THEN  (~correct(C, S1, I) v
                 correct(C, S2, I))
  ELSEIF (S= (t1= t2)) THEN  (J(t1) = J(t2))
  ELSEIF (S= (∀ x S1)) THEN (∀ x< U correct(C, S1, I))
  ELSEIF (S= (∃ x S1)) THEN (∃ x< U correct(C, S1, I))
  ELSEIF (S= (t< SYS)) THEN (J(t) < SYS)
```

```
    ELSEIF (S= (t◂ ENV)) THEN (J(t) ◂ ENV)
    ELSEIF (S= (t◂ IPi)) THEN (J(t) ◂ IPi)
    ELSEIF (S= (t◂ OPi)) THEN (J(t) ◂ OPi)
    ELSEIF (S= (t◂ CE))
           THEN ((J(t) ◂ SYS) v
                 (J(t) ◂ ENV) v
                 (ǂ 1≤ i≤ n J(t) ◂ IPi) v
                 (ǂ 1≤ i≤ m J(t) ◂ OPi))  [1]
    ELSEIF (S= (t1-> t2)) THEN precede(J(t1), J(t2), H)
    ELSEIF (S= (t1=> t2))
           THEN ((~in(J(t1), H) v in(J(t2), H)) ^
                correct(C, (t1-> t2), I))
    ELSEIF (S= (ord(t, IPi)= n))
           THEN eq(J(t), IPi, H, J(n))
    ELSEIF (S= (ord(t, OPi)= n))
           THEN eq(J(t), OPi, H, J(n))
    ELSEIF (S= (P(t1, ..., tn))
           THEN P(J(t1), ..., J(tn))
    ELSE error

 where

 precede(x, y, H) =
    IF (H= nil) THEN true
    ELSEIF (car(H)= y) THEN false
    ELSEIF (car(H)= x) THEN true
    ELSE precede(x, y, cdr(H))

 eq(t, IP, H, n) =
    IF ((H= nil) v (n≤ 0)) THEN false
    ELSEIF ((car(H)= t) ^ (n= 1)) THEN true
    ELSEIF (car(H) ◂ IP) THEN eq(t, IP, cdr(H), n-1)
    ELSE eq(t, IP, cdr(H), n)
```

The interpreter above defines only the meaning of the notations that are unique to EBS (e.g. ->, =>); other user-defined predicates or functions are interpreted in a loose way. The interpreter shows that the relation -> can be used to define the relation =>.

[1] CE represents the computation event set, including SYS, ENV and the interface Ports.

Definition A.8. Centralized Processor Implementation

A centralized processor implementation is a set of centralized computations that satisfy the behavior specification.

1.3. A Multiprocessor Interpretation

By a multiprocessor interpretation we mean the mapping of the partial ordering of events into a finite multi-linear order implementation. That is, assume that there are a finite number of local clocks to order totally the events in the local area computations and that there are explicit synchronization messages to order events in different areas.

Definition A.9. Process

A (local computation) process is a history.

Definition A.10. Synchronization Information

The synchronization information carried by a message m is an ordered pair <ei, ej> of events, where ei is the sending event by a process, denoted by m.send and ej is the receiving event by another process, denoted by m.receive.

Definition A.11. Synchronization

A synchronization relation M is a set of synchronization information.

<u>Definition A.12. Multiprocessor Computation</u>

A K-processor computation is a 6-tuple

MC = <U, X, Q, A, F, P>

where U, A, F, P are the same as in the centralized computation;

X is a set PP of processes (P1, ..., PK) and a synchronization relation M of messages;

Q is a partition of the whole event space (the union of all process events) into sets ENV, SYS, IP1, ... IPn and OP1, ..., OPn such that all events in IPi (or OPj) belong to single processes for all $1 \le i \le n$ (or $1 \le j \le m$); we denote the process which contains IPi (or OPj) by P(IPi) (or P(OPj)).

The interpreter I and the extended interpreter J for multiprocessor computations can be defined in a way similar to those for centralized processor computations.

The correctness of a multiprocessor computation MC according to a specification S under the interpretation I can be defined in much the same way as that of a centralized computation. Only the "precedes" relation "->" and the membership relation "in" need to be redefined, since the relation "=>" can be defined by "->" and "in".

<u>Definition A.13. Membership</u>

An event e being a member <u>im</u> of the K-processor computation is defined as follows:

$im(e, X) = \exists\ 1 \leq i \leq K\ in(e, Pi)$

Definition A.14. Correctness of a Computation

The correctness of a K-processor computation MC according to a specification S under the interpretation I is defined as follows:

```
mcorrect(MC, S, I) =
    ...
    (* same as in centralized processor computation *)
    ...
    ELSEIF (S= (t1-> t2)) THEN prec(J(t1), J(t2), PP, M)
    ELSEIF (S= (t1=> t2))
           THEN ((~im(J(t1), PP) v im(J(t2), PP)) ^
                 mcorrect(MC, (t1-> t2), I))
    ELSEIF (S= (ord(t, IPi)= n))
           THEN eq(J(t), IPi, P(IPi), J(n))
    ELSEIF (S= (ord(t, OPi)= n))
           THEN eq(J(t), OPi, P(OPi), J(n))
    ELSEIF (S= P(t1, ..., tn))
           THEN P(J(t1), ..., J(tn))
    ...
```

where the "precedes" relation prec in the multiprocessor implementation can be defined as follows (see also Figure A.1.) :

```
prec(x, y, PP, M) =
   IF (∃ 1≤ n≤ K in (y, Pn)) THEN
          IF (~(∃ 1≤ m≤ K in(x, Pm)) THEN
              IF (~(m=n)) THEN precede(x, y, Pm)
              ELSE (∃ 1< k≤ K
                     e11 (= x), e12 ∈ Pm1 (= Pm)
                     e21, e22 ∈ Pm2
                     ...
                     ek1, ek2∈ Pmk (= Pn)
                    (∀ 1≤ i≤ k
                       (~precede(ei1, ei2, Pmi) v
                                          (ei1= ei2)) ^
                    (∀ 1≤ i≤ k
```

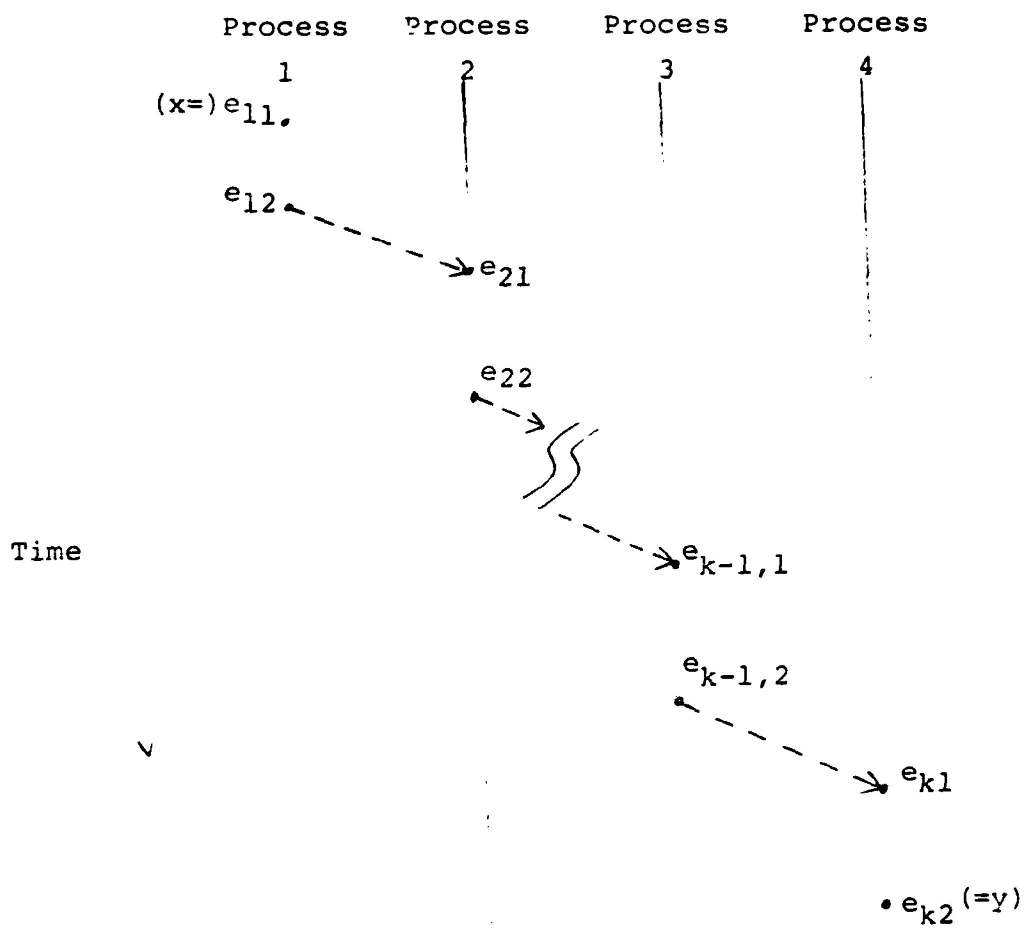


Figure A.1. Interpretation of $x \rightarrow y$ in a Multi-Processor Implementation

```
                    (ei2, e(i+1)1) < M))
        ELSE false
    ELSE true
```

Definition A.15. K-processor Implementation

A K-processor implementation is a set of K-processor computations that satisfy the behavior specification.

APPENDIX B.

A FORMAL PROOF OF THEOREM 3.1.

Theorem 3.1.

```
  If T is a transitivity relation,
     (i.e., ∀ p, q, r T(p,q) ^ T(q,r) #> T(p,r))
     then,
         ( ∀ x∈ X ∃ y∈ Y T(x,y)) ^
         ( ∀ y∈ Y ∃ z∈ Z T(y,z))
         #> ( ∀ x∈ X ∃ z∈ Z T(x,z))
```

Proof

Convert all predicts to well-formed formula.

```
The transitive law:
     ∀ p, q, r
       T(p,q) ^ T(q,r) #> T(p,r)
  == ∀ p, q, r
       ~T(p,q) v ~T(q,r) v T(p,r)

Condition
     ∀ x∈ X ∃ y∈ Y  T(x,y)
  == ∀ x  X(x) #> ∃ y(Y(y) ^ T(x,y))
  == ∀ x ~X(x) v ∃ y(Y(y) ^ T(x,y))
  == ∀ x ~X(x) v (Y(f(x)) ^ T(x, f(x)))
  == ∀ x (~X(x) v Y(f(x))) ^ (~X(x) v T(x, f(x)))

Similarly
     ∀ y∈ Y ∃ z∈ Z T(y,z)
  == ∀ y (~Y(y) v Z(g(y))) ^ (~Y(y) v T(y, g(y)))

Negation of the conclusion to be proved
     ~( ∀ x∈ X ∃ z∈ Z  A(x,z))
  == ~( ∀ x ~X(x) v ∃ z (Z(z) ^ A(x,z)))
  == ∃ x X(x) ^ ∀ z (~Z(z) v ~A(x,z))
  == ∀ z X(a) ^ (~Z(z) v ~A(a, z))

Convert all the wffs to clauses:
    (1) ~T(p,q) v ~T(q,r) v T(p,r)
    (2) ~X(x) v Y(f(x))
    (3) ~X(x) v T(x,f(x))
    (4) ~Y(y) v Z(g(y))
    (5) ~Y(y) v T(y,g(y))
    (6)  X(a)
```

(7) ~Z(z) v ~A(a,z)

The Refutation Graph of these clauses appears as follows:

$$\begin{tikzcd}
\boxed{Y(y)\ v\ T(y,g(y))} \arrow[dr] & & \boxed{X(x)vY(f(x))} \arrow[dl] \arrow[drr] & & \boxed{Y(y)vZ(g(y))} \arrow[d] \\
& \boxed{X(x)vT(f(x),g(f(x)))} & & & \boxed{X(x)vZ(g(f(x)))} \arrow[dd] \\
\boxed{X(x)vT(x,f(x))} \arrow[dr] & & \boxed{X(a)} \arrow[dl] \arrow[d] \arrow[drr] & & \\
& \boxed{T(a,f(x))} \arrow[dd] & \boxed{T(f(a),g(f(a)))} \arrow[ddd] & & \boxed{Z(g(f(a)))} \arrow[dd] \\
\boxed{T(p,q)v\ T(q,r)vT(p,r)} \arrow[dr] & & & \boxed{Z(z)v\ T(a,z)} \arrow[dr] & \\
& \boxed{T(f(a),r)vT(a,r)} \arrow[dr] & & & \boxed{T(a,g(f,a)))} \arrow[dl] \\
& & \boxed{T(a,g(f(a)))} \arrow[dr] & & \\
& & & \boxed{nil} &
\end{tikzcd}$$

A Refutation Graph of Theorem 3.1.