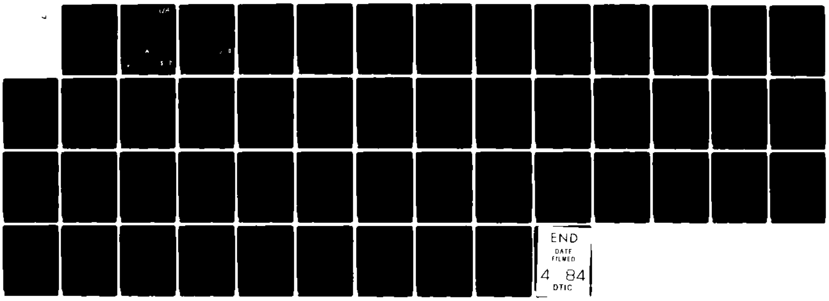
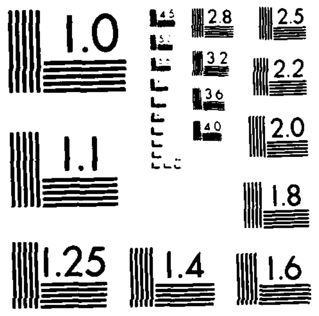
AD-A139 305 FORMAL SPECIFICATION AND VERIFICATION OF DISTRIBUTED SYSTEM(U) MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE B S CHEN ET AL. JUN 83 TR-1295 1/1

UNCLASSIFIED AFOSR-TR-84-0172 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

AFOSR-TR- 84-0172

13

AD A139305

COMPUTER SCIENCE
TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

DTIC FILE COPY

84 03 22 111

(13)

Technical Report #1295 June, 1983

Formal Specification and Verification of Distributed Systems

Bo-Shoe Chen*

Raymond T. Yeh
Computer Science Department
University of Maryland

*Bo-Shoe Chen, Bell Laboratories, Naperville, IL 60566
The work was done when Dr. Chen was with the University of Maryland.

This work was supported in part by the Air Force under contract #F4962080C0001.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) **AFOSR-TR- 84-0172**
6a. NAME OF PERFORMING ORGANIZATION University of Maryland | 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research
6c. ADDRESS *(City, State and ZIP Code)* College Park, MD, 20742	7b. ADDRESS *(City, State and ZIP Code)* Directorate of Mathematical & Information Sciences, Bolling AFB DC 20332
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR | 8b. OFFICE SYMBOL *(If applicable)* NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-80-C-0001
8c. ADDRESS *(City, State and ZIP Code)* Bolling AFB DC 20332	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
61102F	2304	A2	

11. TITLE *(Include Security Classification)*
SEE REMARKS

12. PERSONAL AUTHOR(S) Bo-Shoe Chen and Raymond T. Yeh

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Yr., Mo., Day)*	15. PAGE COUNT
Technical Report	FROM ______ TO ______	June 1983	46

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB. GR.	

19. ABSTRACT *(Continue on reverse 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 of each other, and the entire system may lack an accurate global clock.

Over

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER *(Include Area Code)*	22c. OFFICE SYMBOL
Robert N. Buchal	(202) 767-4939	NM

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

11. "Formal Specification and Verification of Distributed Systems"

19. Con't

In this thesis, we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal 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" relations, representing causality. No assumption about the existence of a global clock is made in the specifications.

The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, including communication systems, process control systems, and a distributed prime number generator [HOA78], are used to demonstrate the power of the technique.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and the structure specification of a system. Both safety and liveness properties can be specified and verified. Furthermore, 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.

Abstract

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 of each other, and the entire system may lack an accurate global clock.

In this thesis, we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal 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" relations, representing causality. No assumption about the existence of a global clock is made in the specifications.

The specification technique has a rather wide range of applications. Examples from different classes of distributed systems, including communication systems, process control systems, and a distributed prime number generator [HOA78], are used to demonstrate the power of the technique.

The correctness of a design can be proved before implementation by checking the consistency between the behavior specification and the structure specification of a system. Both safety and liveness properties can be specified and verified. Furthermore, 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.

A

CONTENTS

1. Introduction

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 of each other, and the entire system may lack an accurate global clock.

Finite-state machines, such as Petri Nets [PET77], SPECIAL [ROB77, SUN79], and Right-Synchronization Controllers [CON79] have been widely used to specify and verify concurrent systems. However, there are several drawbacks to this approach. As the number of possible states increases, analyzing all interactions becomes impossible. 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, which are the requirements that certain events eventually take place, are difficult to state or prove using a state-machine approach.

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 implementations. An operational model specifies a system's requirements by giving an abstract implementation. There is no indication of what aspects of the model should be rigorously followed and what aspects merely illustrate functionality.

For sequential programs, algebraic/axiomatic specification techniques provide the abstraction necessary to state properties of a program without giving an implementation. Unfortunately, time-dependent properties of concurrent systems such as concurrency or mutual exclusion are difficult to specify in standard algebraic/axiomatic approaches.

In this paper, we develop an event-based model to specify formally the behavior (the external view) and the structure (the internal view) of distributed systems. Both safety and liveness 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 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 the structure specification of a system. Moreover, since the specification technique defines the orthogonal properties of a system separately, each of them can then be verified independently. In this way, the proof technique avoids the exponential state-explosion problem found in state-machine specification techniques.

Section 2 gives the conceptual models of distributed systems from both a user's and a designer's view points. Section 3 discusses the event-based model, defining events and event relationships. Section 4 presents our Event-Based Specification Language (EBS) together with the behavior specifications of several examples. EBS structure specification language and the verification technique for a design are then presented in Section 5. Finally, comparisons of our approach with Temporal Logic [PNU77, OWI82], and Trace approaches [MIS81, ZHO81] are discussed and the advantages of our approach are summerized in Section 6.

2. Conceptual Modeling

A distributed system may be described from two different points of view. From a designer's view, it consists of local processes interacting with users and communicating among themselves via the communication media. Each local process can be described by the operations responding to user's commands, messages from other processes, or internal clocks. The structure is depicted in Figure 1.

From a user's view, a distributed system is a shared server, or a black box with only the interfaces visible, as shown in Figure 2. In this case, except for performance issues, there is no essential functional difference between a distributed system and a centralized one. The only things interesting to the users are messages or events happen in the interfaces and the relationships among the messages or the events. This kind of interface description of a system is called its *behavior specification*.

3. The Event Model

The model that our behavior specification is based upon therefore consists of events and their relationships.

3.1 Events

An event is an *instantaneous*, *atomic* state transition in the computation history 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. The basic properties of events are similar to those in the ACTOR model [HEW77] with some modification [CHE82a].

3.2 Event Relationships

3.2.1 **Time Ordering.** In describing the time ordering among events, a system-wide reliable clock is usually assumed to order them totally. Unfortunately, the assumption of a global clock is too strong in describing the computation of a distributed system. Theoretically speaking, it is impossible, in some extreme cases, to order two events totally. Practically speaking, implementing such a global clock is quite expensive and unnecessary in a distributed system having highly autonomous computing nodes. The "precedes" relation [GRE77, LAM78], denoted by "->", is a much weaker, partial-ordering relation that can be used to represent the time order.

The interpretation of "->" as a time ordering means that, *if* e1 and e2 are events in a system and e1->e2, then e1 happens before e2 by any measure of time. To understand the meaning of "->", let us look at Figure 3. Each vertical line in Figure 3 represents the computation history of a (sequential) process. A *process* means an autonomous computing node having its own local clock; different processes may use different time scales. The dots denote events and the dotted line between events denote messages. The relation "->" has the following properties:

1. If e1 and e2 are events in the same process, and e1 comes before e2, then e1->e2 (e.g., p1->p2 in Figure 3).

2. If e1 is the event of sending a message by one process and e2 is the event of receiving the message by another process, then e1->e2 (e.g., p1->q2 in Figure 3).

3. Transitivity- If e1->e2 and e2->e3, then e1->e3 (e.g., p1->r2 in Figure 3).

4. Irreflexivity- ~(e->e).

5. Antisymmetry- If e1->e2, then ~(e2->e1).

3.2.2 **Concurrency.** The concurrency relation can be defined easily by the precedes relation as follows: two distinct ev^nts, say e1 and e2, are concurrent iff ~(e1->e2) and ~(e2->e1). Figure 3, for example, there is no way to tell whether p1 or q ^omes first; they may be concurrent.

3.2.3 **Enables Relation.** Liveness properties are assertions that certain events will happen eventually. Examples of liveness properties are guaranteed message delivery or starvation-free service. Such properties can be specified by the "enables" relation, denoted by "=>", between events. Two events a and b satisfy the relation a=>b iff the existence of event a will cause the occurrence of events b in the future. The relation => has the following properties:

1. Being enabled in the future- if a=>b then a->b.
2. Antisymmetry- If a=>b, then ~(b=>a).
3. Irreflexivity- ~(a=>a).
4. Transitivity- If a=>b and b=>c, then a=>c.

In other words, the enables relation is also a partial-ordering relation. Properties (2) and (3) can be derived from (1) and the properties of "->" while (1) and (4) are axioms.

3.2.4 System, Environment, and their Interface Ports. It is convenient to categorize the event space into distinct domains for the ease of specification. Three domains are identified: the system, the environment, and the interface ports.

A *system* interacts with its *environment* by exchanging messages through unidirectional interfaces called *ports*, as depicted in Figure 4. An inport directs messages from the environment to the system while an outport directs messages from the system to the environment.

Every port defines sequences of interface events. Every event e in a port history is uniquely identified by an integer, called its *ordinal number*, represented by ord(e). Thus, a port history is a total ordering of events, although the events in system or in environment are only partially ordered.

4. Behavior Specification with the EBS Language

Based on the event concept together with the first-order predicate calculus (with equality "≡"), we develop a language called EBS (Event Based Specification Language) to specify the behavior of distributed systems. The formal syntax of EBS can be found in Appendix A. Examples will be used to show its expressive power.

4.1 Example 1: Reliable Transmission Systems

A reliable transmission system (RT) is one through which messages are transmitted without loss, duplication, reordering, or error from an inport to an outport (see Figure 5).

No messages are lost during transmission when every message sent from the inport A is eventually transmitted to the outport B. This property can be specified as follows:

```
/* RT11(A,B): No loss of messages */ 1
   ∀ a<A ∃ b<B
     a=>b;
```

The operators and their precedence rules in EBS are as follows:

1. unary operators: ∀ (for all), ∃ (there exists) and ~ (logical not);
2. relational operators: <(belongs to), = (equivalent), ≡ (equals to), => (enables), and -> (precedes);
3. logical operators: v (logical or), ^ (logical and); and
4. #> (logical implication) and <#> (two way implication).

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);
```

where the notation "#>" represents logical implication. The reserved word SYS (ENV) refers to the system event set (environment).

1. RT11 will be used to name this property afterwards for convenience.

The property that there is 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->a2 ^ b2->b1)
```

which says that if a1 is sent before a2 then it will also be received before a2 and vice versa.

The property that 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 #> b.msg=a.msg
```

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

These are the minimal properties that a reliable transmission system should have. A very good feature of this kind of orthogonal specification is that a specification can be easily adapted to different applications. For example, a system that not only transmits messages reliably but also performs code conversion between computer systems using different codes (e.g., ASCII and EBCDIC), can be specified by modifying RT12 to

```
/* Message transformer */
   ∀ a<A, b<B
     a=>b #> b.msg= F(a.msg)
```

where F is the code conversion function, leaving others unchanged. Another example is that if properties RT11, RT14 and RT15 are deleted, then the system is an unreliable one that may lose, duplicate or reorder messages (see also Section 5.3).

4.2 Example 2: Multiplexors and Decoders

Two fundamental mechanisms in a packet-switched network to share its expensive transmission capacity are *multiplexing* and *decoding*. A multiplexor (see Figure 6) interleaves packets from various sources into a single communication channel. A decoder (see Figure 7) distributes the packets from a single channel to various destinations.

A multiplexor with two inports can be specified as follows:

```
System MX (A: inport;
           B: inport;
           C: outport);
   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 were defined in the system RT. A decoder is a system that distributes messages reliably from a single inport to several outports according to some predefined distribution criteria. It can be viewed as a set of *filters*. A filter is a system that transmits a message reliably iff it satisfies some predefined criterion. To specify a filter, only RT11 in the Reliable System needs to be modified as follows:

```
     /* A message at A will be sent to B
        iff it satisfies P */
        ∀ a<A
          P(a) <#> (∃ b<B a=> b)
```

where the notation "<#>" represents two way implication. A decoder is essentially a collection of such filters. A decoder with an inport A, two outport B1 and B2, and two distribution functions P1 and P2, can be specified by modifying RT11 as follows:

```
    /* A message at A will be sent to B1 or B2
       iff it satisfies P1 or P2 respectively */
       ∀ a<A
         (P1(a) <#> (∃ b1<B1 a=>b1)) ^
         (P2(a) <#> (∃ b2<B2 a=>b2))
```

while retaining the other RT's for both (A, B1) and (A, B2).

4.3 Example 3: an Engine Monitoring System

A microprocessor aircraft engine monitor for use on both experimental and in-service aircrafts is described in [ALF77]. The capabilities of this Engine Monitoring System are 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 limits and may log the history of each machine.

The system interface structure is depicted in Figure 8. We specify the behavior of this system in EBS as follows:

```
/* Engine Monitoring System */

System EMS (
            engine[i]|newdata: inport;
               /* i from 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 from 1 to 10. */
            warning: outport;
            ring: outport;
            engine-history: outport;
            inwarning: predicate;
            inalarm: predicate;
            realtime: function
           );

  Messagetype

    newdata.msg: record
                   T1, T2, T3, /* temperatures */
                   P1, P2, P3: /* pressures */
                               real;
                   S1, S2:     /* switches */
                           boolean;
                   Time:       /* recording time */
                           real;
                 end;
    log-history.msg: /* engine id whose history is to be
                        logged */
                     integer;
    new-standard.msg:
      record
        id: /* engine id whose standard
```

```
              is to be changed */
           integer;
       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: /* engine id that is in warning range */
                integer;
   ring.msg: boolean;
   engine[i]|readdata.msg: /* i from 1 to 10 */
                           boolean;
End messagetype;

Behavior

   /* Part I: System's response to a newdata */

      /* Part I.1: The relationship between  ports  newdata  and
         warning  is a filter: output a warning iff a newdata is
         in warning range. */

         /* NW11: Output a warning message iff a newdata  is  in
            warning  range with respect to the most recent stan-
            dard set up. */
            ∀ i<{1..10},
              x<engine[i]|newdata,
              mrs<newstand
              mrs.msg=i ^ mrs->x ^
              (∀ c<new-standard
                 c.msg=i ^ c->x #> c≡mrs v c->mrs) ^
              /* mrs is the most recent standard. */
              inwarning(x.msg, mrs.msg)
              <#> ∃ w<warning x=>w;

         /* The specification of properties RT12-RT15 are  simi-
            lar to that of a multiplexor and is omitted here. */

         /* NW21: a warning message returns the id of an  engine
         that is is warning. */
         ∀ i<{1..10},
           x<engine[i]|newdata, w<warning
           x=>w #> w.msg=i;

   /* Part I.2: The relationship between  ports  newdata  and
      ring  is  a filter: output an alarm iff a newdata is in
      alarm range. */

      /* The specification is similar to that in Part I.1 and
         is omitted here. */

/* Part II: system's response to a log-history command */

   /* LH11: all previous engine[i]|newdata will be output  to
      engine-history in response to a log-history(i) command.
      */
      ∀ 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  com-
      mand. */
      ∀ 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
           (x2->x1 ^ h2->h1) v
           (x1≡x2 ^
            ((h1≡h2) v
             (h1.enginedata.Time<h2.enginedata.Time
                                           ^ h1->h2) v
             (h1.enginedata.Time>h2.enginedata.Time
                                           ^ h2->h1)))

/* Part III: The behavior of outport readdata */

   /* Read engine data repeatedly. */
```

```
        ∀ i<{1..10}, x<engine[i]|readdata
          ∃ y<engine[i]|readdata
            x->y;

      /* Read engine data periodically. */
      /* The period is T. The function  realtime()  returns  the
         realtime between two events. */
         ∀ i<{1..10}, x, y<engine[i]|readdata
           ord(y)=ord(x)+1
           #> realtime(x, y)= T;

   End behavior

 End system.
```

This 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. Rather, the side-effects of a command are made visible only when other commands read its message contents. Also note that every engine can have its own local clock to provide a timer value for engine data and to read engine data periodically. A synchronized global clock is by no means necessary.

5. Structure Specification and Verification

In a top-down development methodology, a system behavior (the external view) is specified first. Then the behavior specification is decomposed into a design structure (the internal view). A formal design description of a system is called its *structure specification*. Correctness of a design can then be established by proving the consistency between the behavior specification and the design.

5.1 System Constructs

The constructs that describe a design structure are: the subsystem, the link, and the interface definition.

A system is decomposed into a set of *subsystems* communicating among themselves via *links*, and a set of *interface definitions* to communicate with the environment.

A *subsystem* defines a subset of events from its enclosing system event set; every event in a subsystem is in its enclosing system event set. The computation of a subsystem is described by a behavior specification, which can be further decomposed into a structure specification. In this way, the specification technique supports the hierarchical design methodology.

A *link* connects an outport of a subsystem to an inport of another subsystem. The construct "connect(P, Q)==R" specifies a link named R that connects an outport P to an inport Q. When two ports are linked, they are merged into a single port and become identical: any event for one is an event for the other. Note that a link is different from a reliable transmission system in that the latter always introduces finite message delay.

An *interface definition* "X==Y" specifies that the inport (outport) X of a subsystem is used as the inport (outport) Y of its enclosing system.

5.2 Example 4: A Tandem Network

In a packet-switched network, a packet of messages is passed via some intermediate nodes, instead of being sent directly from the source node to the destination node using a long-haul transmission line. A packet is sent reliably from the source node to the intermediate node and then sent reliably from the intermediate node to the destination node. The structure of this communication system can be considered to consist of a set of reliable transmission subsystems connecting in series, which as a whole provides a reliable transmission system service. Such a serial connection of two or more subsystems is call a *tandem network* (See Figure 9).

5.2.1 The Structure Specification of a Tandem Network. The structure of a system SZ, which is composed of a serial connection of two reliable transmission systems SX and SY, can be specified formally as follows:

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

  Structure

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

      Behavior
        RT's(PA, PB); [2]
      End behavior;

    End subsystem;

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

      Behavior
        RT's(PC, PD);
      End behavior;

    End subsystem;

    Network
      connect(SX.PB, SY.PC)==SZ.PE;
    End network;

    Interface
      SZ.PA==SX.PA;
      SZ.PD==SY.PD;
    End interface;

  End structure;

End system.
```

System SZ is composed of two reliable transmission subsystems SX and SY. A system name followed by a dot and a port name denotes a port in the system. A link name PE in system SZ connects outport PB of system SX to inport of SY. The interface part says that system SZ uses system SX's inport PA and system SY's outport as interface ports.

2. See Section 4.1 for the definitions of RT's.

5.2.2 **The Verification of the Tandem Network.** Since the same mathematically sound notations (i.e.,first-order logic and partial ordering relations) are used in the behavior and the structure specifications, the verification can be carried out as proving theorems.

Theorem 1. A tandem connection of two reliable systems behaves as a single reliable one.

Proof The no loss property can be proved as follows:

1. For all p in PA there is a q in PB such that p=>q (RT11 of SX)
2. for all r in PC there is an s in PD such that r=>s (RT11 of SY)
3. Let q≡r (PB and PC are connected)
4. p=>s (since => is transitive)

Other properties can be proved similarly, independent of one another.

5.3 **Example 5: An Alternate-Bit Protocol**

An Alternate-Bit Protocol (ABP) provides a reliable message transfer service over an unreliable transmission medium from a fixed sender to a fixed receiver. A transmission medium is unreliable if it may lose, duplicate or reorder messages; however, there is a nonzero probability of successful message transmission. The "nonzero probability of message transmission" means that if messages having the same contents are sent repeatedly then at least one of them will reach the destination. The behavior of an unreliable system can be specified by deleting RT11, RT14, and RT15 from a reliable system and adding the "nonzero probability" property as follows:

```
/* 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)
```

The precondition of the predicate says there is an unbounded number of messages having the same contents as ai. The postcondition specifies that at least one of them will arrive at B. The service provided by the ABP is simply that of a reliable system.

5.3.1 **Structure Specification of the Alternate-Bit Protocol.** The "nonzero probability" plays a key role in guaranteeing that a message sent from one end is received at the other end. The structure of the ABP is depicted in Figure 10. The SS (Send-Station) accepts a message from IP and sends it repeatedly to the RS (Receive-Station) via the Data Medium until an acknowledgement is received from the RS via the Acknowledgement Medium. The RS acknowledges all messages received. To avoid duplication of messages, a serial (integer) number is attached to each message sent by the SS. RS accepts a message only if its serial number has never appeared before and acknowledges the receipt by sending back the serial number. To avoid reodering messages, a message from IP will not be sent until all previous ones are acknowledged.

These key concepts can be specified formally in EBS as follows:

```
/* Key Specifications in the ABP */
```

Send-Station:

```
/* SS1: Guaranteed message transmission: sends
   the same message repeatedly until get back
   an acknowledgement. */
```

```
∀ ip<IP
  (∃ ds<DS ip=>ds) ^
  ((∃ ar<AR ar.msg=ord(ip)) v
   (∀ d1<DS ip=>d1
      #> ∃ d2<DS ip=>d2 ^ d1->d2));

/* SS2: Sequence control: do not send a new message
   until all previous ones are acknowledged. */
   ∀ ip<IP
     (∀ k<N [3]
       k< ord(ip)
       #> ∃ ar<AR ar.msg=k ^ ar->ip);

/* SS3: Contents of messages: send out a message
   together with a serial number as a unique id. */
   ∀ ip<IP, ds<DS
     ip=>ds #> ds.data=ip.msg ^  [4]
               ds.msgno=ord(ip);
```

Receive-Station:

```
/* RR1: Send acknowledgement for every message
   received back to the Sender. */
   ∀ dr<DR ∃ as<AS
     dr=>as;

/* RR2: Send back the serial number as an
   acknowledgement of receipt. */
   ∀ dr<DR, as<AS
     dr=>as #> as.msg=dr.msgno;

/* RR3: Accept those messages that never come
   before */
   ∀ dr<DR
     (∃ op<OP dr=>op)
     <#> ~(∃ dr'<DR
              dr'-> dr ^
              dr'.msgno=dr.msgno);
```

5.3.2 **The Verification of the Alternate-Bit Protocol.** We will now prove that the ABP makes an unreliable system into a reliable one. Since the Data Medium is an unreliable one, the SS has to send the messages repeatedly to guarantee that at least one will reach the

3. N represents the set of nature numbers.

4. The message in a ds or a dr is of record type having two fields: data and msgno.

RS. However, since the Acknowledgement 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 repeatedly, not only one but 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.

Theorem 2. If the communication medium has a nonzero probability of message transmission, and if an unbounded number of messages having the same contents are sent from A, then not only one but an unbounded number of messages will arrive at B.

Proof By mathematical induction: Since an unbounded number of messages having the same contents are sent from IP, at least one of them, say x, will reach OP (the nonzero probability property). Since the number of messages after x is again unbounded, at least one of them will arrive at OP. The same process goes on and on.

Theorem 3. Every message ip in IP will get back an acknowledgement from RS, carrying ord(ip) as message contents.

Proof By contradiction: Assume there is no acknowledgement for ip from RS (through AR) then an unbounded number of messages will be sent from DS (the SS1 property). By Theorem 2, an acknowledgement will eventually be received.

Theorem 4. The ABP makes an unreliable system behave as a reliable one.

Proof The no loss property (RT11) is easy to prove based on Theorem 3. Other properties can be proved one by one similar to the proofs in the tandem network.

Refer to [CHE82b] for a complete specification and verification of the ABP.

5.4 **Example 6: 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 simply a filter that filters out the non-prime numbers, and is specified by modifying RT11 from the reliable system to:

```
/* Output a number iff it is prime */
   ∀ a<A
     ~(∃ a'<A a'->a ^ a'.msg|a.msg) [5]
     <#> (∃ b<B a=>b);
```

A distributed design [HOA78] to generate prime numbers using the "sieve of Eratosthenes" method, is depicted in Figure 11.

PNG consists of two types of processes: sieves and a printer. To simplify the description, 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 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. The printer has one outport B, which is also the outport of PNG.

5.4.1 The Structure Specification of PNG.

5.4.1.1 A Sieve. The first message p received by a sieve (see Figure 12) from P is sent to the printer through Q. Every subsequent message x received is then checked to see if it is a multiple

5. a|b means a divides b.

of p; if x is a multiple of p it is discarded; otherwise, it is sent on to the next sieve through R. The relations between P and Q, and between P and R, are also "filters".

```
/* Relation between P and Q: A message
   is sent to Q iff it is the first in P
*/
    ∀ p<P
      ord(p)=1 <#> (∃ q<Q p=>q);

/* Relation between P and R: A message is
   sent to R iff it is not a multiple of
   the first one message
*/
    ∀ p1, p2 <P
      (ord(p1)=1 ^ ord(p2)>1 ^ ~(p1.msg|p2.msg)
      <#> (∃ r<R p2=>r)

/* Messages will be received in order by
   Q and R
*/
    ∀ p1, p2<P, q<Q, r<R
      p1=>q ^ p2=>r
      #> (p1->p2 ^ q->r) v (p2->p1 ^ r->q)
```

5.4.1.2 **The Printer.** The printer 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. The behavior of a printer is simply as a "multiplexor" (see Section 4.2) with a large amount of inports. Once each subsystem has been specified, the structure specification of PNG is straightforward and is omitted here.

5.4.2 **The Verification of PNG.** Since a message is sent to the printer iff it first arrives at a sieve, a critical step in the verification is to prove that a number will first arrive at a sieve iff it is prime. This can be proved by the following lemmas and theorems.

Lemma 1. The message sequence in every port is in ascending order.

Proof By induction on sieves: because each sieve does not reorder messages.

Lemma 2. If a number x appears at port P[i] then no number in Q[1], ... Q[i-1] divides x.

Proof By contradiction: If x is divisible by a y in Q[j], j<i, then x is divisible by a z *in* P[j] (since RT12(P,Q)). X should have been filtered in Sieve[j] and cannot appear in P[i].

Lemma 3. If a number x first arrives at port P[i] then every number that is less than x is divisible by some number in Q[1], ..., Q[i-1].

Proof By contradiction: If a number y less than x is indivisible by all numbers in Q[1], ..., Q[i-1] then it is indivisible by all the first p's in P[1], ..., p[i-1], and will appear at P[i]. By Lemma 1, y will come before x. This is a contradiction to the assumption "x first arrive at P[i]".

Theorem 5. If a number x first arrives at port P[i] then no previous number divides x.

Proof By contradiction: If y is less than x then y is divisible by some z in Q[1], ..., Q[i-1], by Lemma 3. If y divides x then z divides x. This contradict to the fact that x is indivisible by z, by Lemma 2.

Lemma 4. Every number terminates at some port.

Proof In particular, a number x cannot survive beyond P[x]: Assume x appears at P[x+1]. If a number y greater than x appears in some Q[i], $i \leq x$, then y first arrive at P[i] (even before x), contradicting to Lemma 2. Thus all numbers in Q[1], ..., Q[x] are less than x. However, it is impossible to have x different numbers (numbers are different because of the "no duplication" property of a sieve) less than x.

Theorem 6. Every prime number first arrives at some port.

Proof By Lemma 4, a prime x will terminate at some port, say P[i]. If x had not arrived first then it would have been divisible by the first-arrived number in P[i] (otherwise x would have been sent to P[i+1]). This contradicts to the fact that x is a prime.

Based on Theorems 5 and 6, the following theorem is easy to prove.

Theorem 7. The distributed PNG is a prime number generator.

Refer to [CHE82b] for a complete specification and verification of the distributed PNG.

6. Conclusions and Comparisons to Other Approaches

6.1 The Temporal Logic Approach

Temporal logic was first introduced by Pnueli [PNU77] for defining the semantics of computer programs, and has been used in [OWI82] to specify and verify concurrent systems.

Several properties of concurrent systems can be stated using two temporal operators: [] (henceforth) and <> (eventually). However, 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 are helpful in the "implementation" verification, they are difficult to specify and understand. Proofs of global invariants also require the consideration of "all" possible event interleaving of parallel processes even though there might be no interaction among them.

The time order relation among events in a computation is *implicitly* expressed by the temporal operators. As the number of temporal operators increases in a specification, it becomes quickly very difficult to understand the meaning of the specification. The "precedes" relation in EBS seems to maintain the understandability of the expressions better.

6.2 The Trace Approach

The notion of traces of communicating sequential processes was introduced by Hoare [HOA78], and was used in the specifications of networks of processes by Misra & Chandy [MIS81], and Zhoa & Hoare [ZHO81]. 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" [ZHO81]. In terms of EBS, a trace is simply a sequence of interface events.

The specifications of system computations are expressed in traces exclusively and the entire proof technique deals only with propositions on traces. The notations for sequences such as "concatenation of sequences", "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. Second, since notations for sequences are used exclusively, trace specifications are awkward in expressing properties whose data structure are not well-defined sequences, such as properties in the unreliable systems. Third, a rather serious deficiency is that the "liveness" properties usually cannot be specified and verified using the trace notion directly. In [MIS81], only "safety" properties of the distributed PNG are proved with the author's notice that the "liveness" properties may be impossible to prove using the trace approach.

In comparison, events in EBS are only partially ordered; the concurrency is expressed by the lack of ordering. The concept of events is more fundamental than that of traces (sequences of events); consequently, some properties that can be easily specified in EBS can only be expressed in traces with difficulty.

We conclude our discussion by listing the advantages of the Event-Based Specification Language:

- Formality- partial ordering relations and the first order predicate calculus are mathematically sound.

- Generality- safety, liveness, data-related and control-related properties can be specified and verified.

- Accuracy- the inherent concurrent behavior of distributed systems is represented by the *lack* of ordering among events; the mutual exclusion among events is specified by the precedes relation.

- Orthogonality- properties are specified separately which makes a specification minimal and extensible, and controls the complexity of the verification process.

7. Acknowledgement

We would like to express our special appreciations for Prof. Tang Chih-Sung, whose comments concerning both the first order logic and distributed systems have been extremely helpful. Thanks go to members of the distributed system research group in the university of Maryland, Dr. Joy Reed, Mr. Gary Lackenbourgh, Professors Yuan Pu-Chia, and Zhoa Pao-Suan, from whom our original ideas got refinements. Thanks go also to Professors Harlen Mills, John Gannon, Virgil Gligor, and Pamela Zave for their guidance and comments on the early draft of this paper.

References

[ALF77] Alford, M. W. et al "Requirement Development using SREM Technology" Vol. 1, Technical Report CDRL COOH, Oct. 1977

[BAR69] Bartlett, K. A. et al. "Note on Reliable Full Duplex Transmission on Half Duplex Links", CACM 12(5): 260-261, May 1969

[CHE81] Chen, B. and Yeh, T. Y. "Event-Based Behavior Specification of Distributed Systems", Proc. of IEEE Symp. of Reliability in Distributed Software and Database Systems: 46-52, July, 1981, Pittsburgh, Penn.

[CHE82a] Chen, B. "Event-Based Specification and Verification of Distributed Systems" Ph. D dissertation, Dept. of Comp. Science Univ. of Maryland, May 1982

[CHE82b] Chen, B. and Yeh, R. T. "Formal Specification and Verification of Distributed Systems" Proc. of the 3rd Int. Conf. on Dist. Comp. Syst.: 380-385, Oct. 1982, Miami, Florida

[CON79] Conner, M. H. "Process Synchronization by Behavior Controllers" Ph. D. dissertation, Univ. of Texas at Austin, Aug. 1979

[DAN80] Danthine, A. A. S. "Protocol Representation with Finite-State Models" IEEE Trans. on Comm. COM-28(4): 632-642, April 1980

[END72] Enderton, H. B. "A Mathematical Introduction to Logic", Academic Press, 1972, Chap. 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

[GRE77] Greif, I. "A Language for Formal Problem Specification", CACM 20(12): 931-935, Dec. 1979

[HAI80] Hailpern, B. and Owicki, S. "Verifying Network Protocols Using Temporal Logic" In Trends and Appl. 1980: Comp. Network Protocols, IEEE Computer Society, May 1980

[HEW77] Hewitt, C. and Baker, H. J. "Laws for Communicating Parallel processes", IFIP 987-992, 1977

[HOA78] Hoare, C. A. R. "A Model for Communicating Sequential Processes" Comp. Lab., Oxford Univ., Dec. 1978

[LAM78] Lamport, L. "Time, Clocks, and the Ordering of Events in a Distributed System", CACM 21(7): 558-565, July 1978

[MIS81] Misra, J. and Chandy, K. M. "Proofs of Networks of Processes" IEEE Trans. on Soft. Engr. SE 7(4):417-426, July 1981

[OWI82] Owicki, S. and Lamport, L. "Proving Liveness Properties of Concurrent Programs" ACM Trans. on Prog. Lang. and Syst. (4) 3: 455-495, July 1982

[PET77] Peterson, J. L. "Petri Nets" ACM Computing Survey 9(3): 223-253, Sept. 1977

[PNU77] Pnueli, A. "The Temporal Logic of Programs" Proc. of the 18th Symp. on the Foundation of Comp. Science, IEEE: 46-57, Province, Nov. 1977

[ROB77] Robinson, L. and Roubine, D. "SPECIAL: A SPECIfication 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 the 2nd Int.

Conf. on Dist. Comp. Syst.: 446-454, Paris, France, April 1981

[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 Corp., Sept. 1979

[YEH80] Yeh, R. T. and Zave, P. "Specifying Software Requirements" Proc. of IEEE, Oct. 1980

[ZAV81] Zave, P. and Yeh, R. T. "Executable Requirements for Enbedded Systems" Proc. of the 5th Int. Conf. on Soft. Engr., San Diego, 1981

[ZHO81] Zhoa, C. C. and Hoare, C.A. R. "Partial Correctness of Communicating Sequential Processes" Proc. of the 2nd Int. Conf. on Dist. Comp. Syst.: 1-12, Paris, France, April 1981

Appendix

The syntax of EBS is defined in the 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>
<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 (with equality) separated by semicolons. The structure, subsystem, network, and interface parts are used in system structure specification. 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 system development.

Refer to [ENT72] for the definitions of expressions, terms, atomic formulas, and well-formed formulas. The abbreviation rules for wffs are given below. The precedence rules can be found in Section 4.1.

(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) Rules similar to (9) are for other two-place predicates
(11) x<>y abbreviates ~= x y
(12) S1 <#> S2 abbreviates S1 #> S2 ^ S2 #> S1
(13) Outermost parenthesis may be dropped.

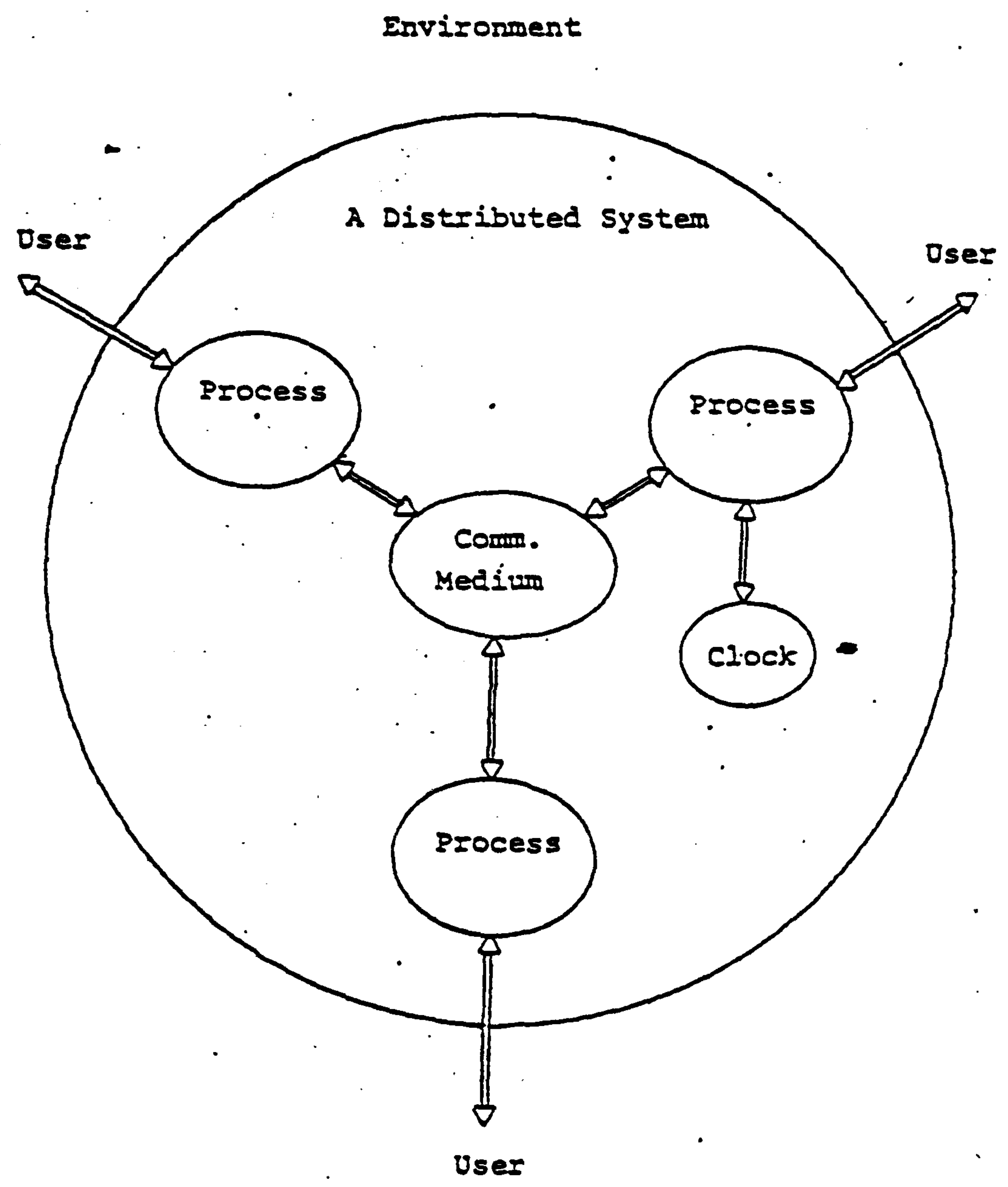


Figure 1. A Distributed System: a Designer's View

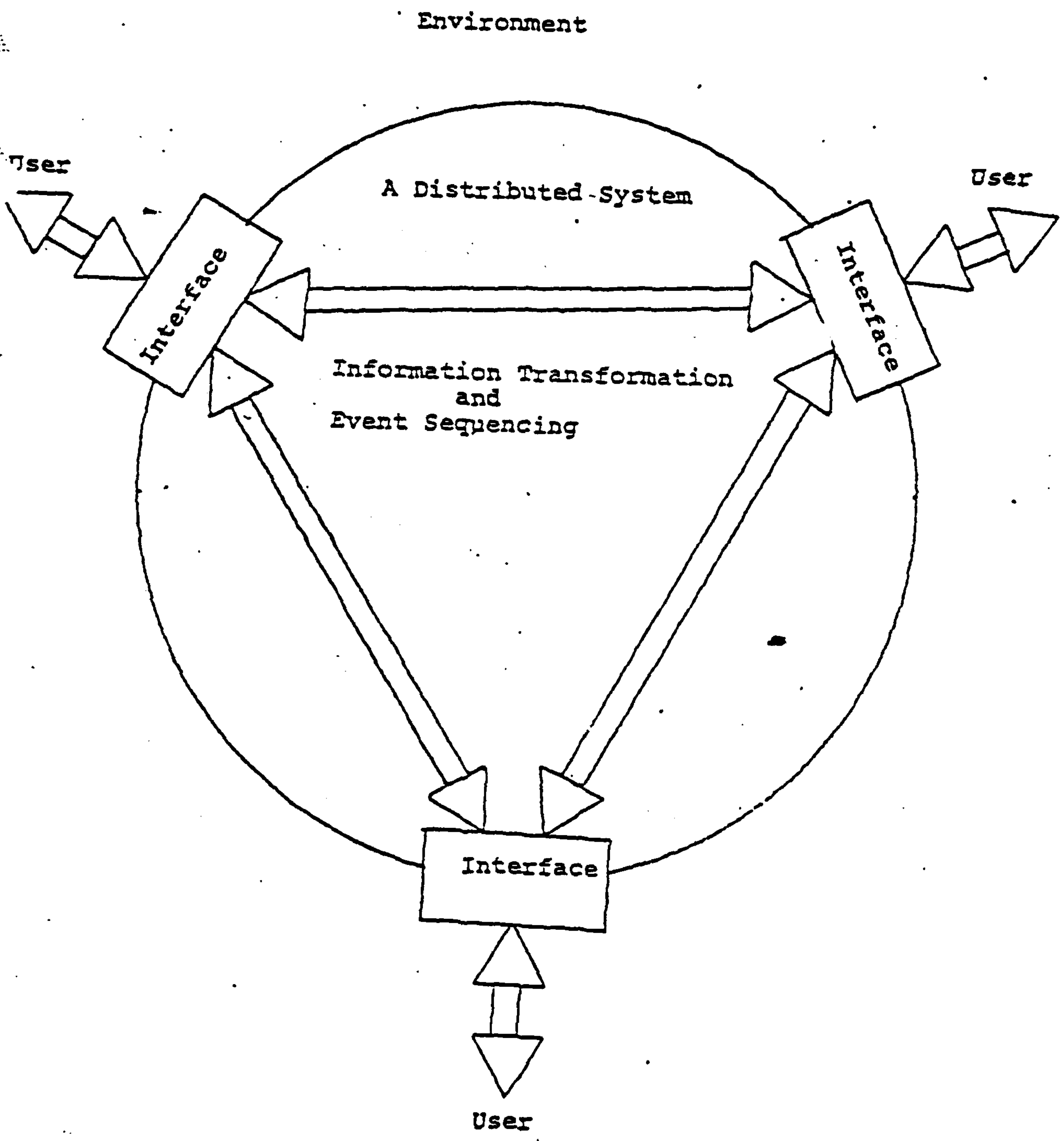


Figure 2. A Distributed System: A User's or a System Analysist's View

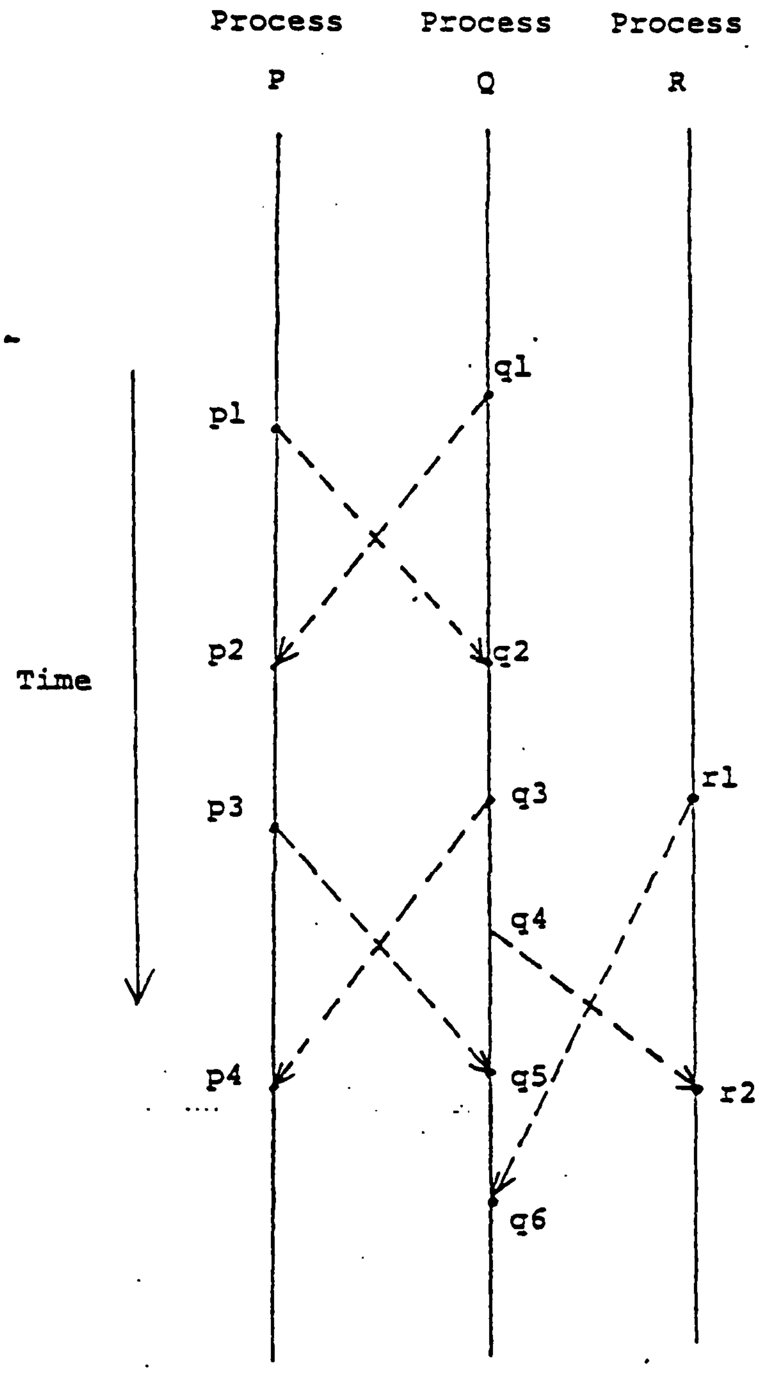


Figure 3. Precedes Relation between Events in Distributed Systems

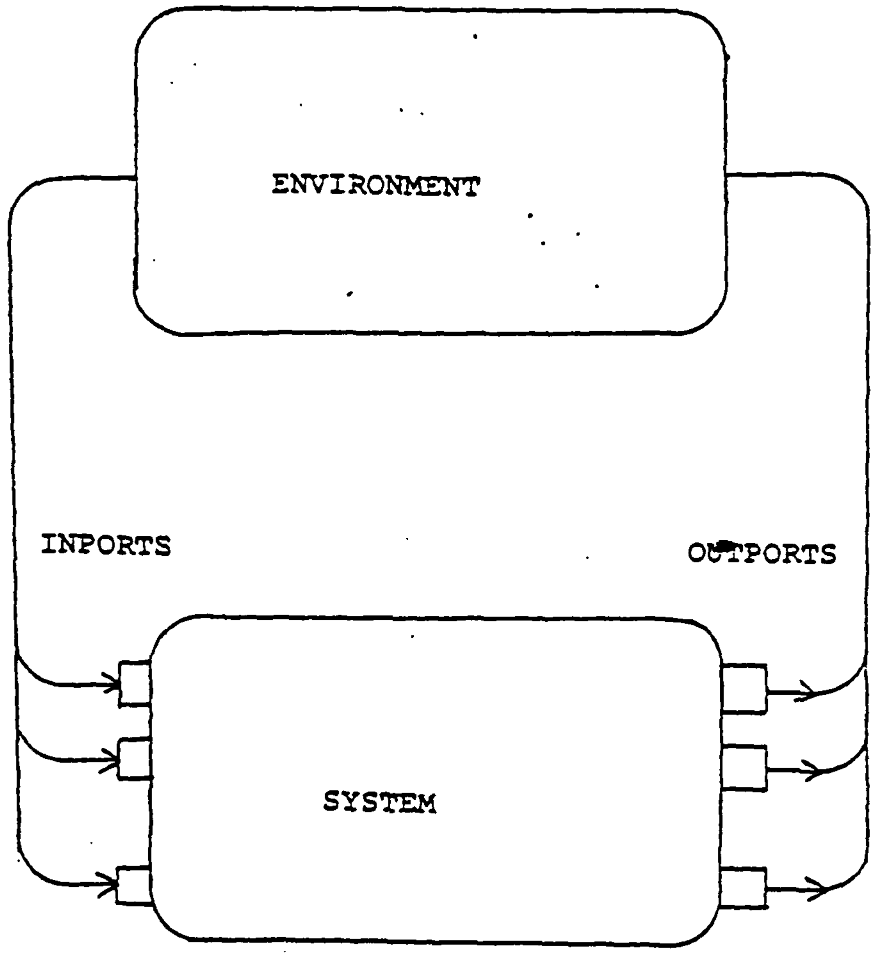


Figure 4. System, Environment and and Their Interfaces

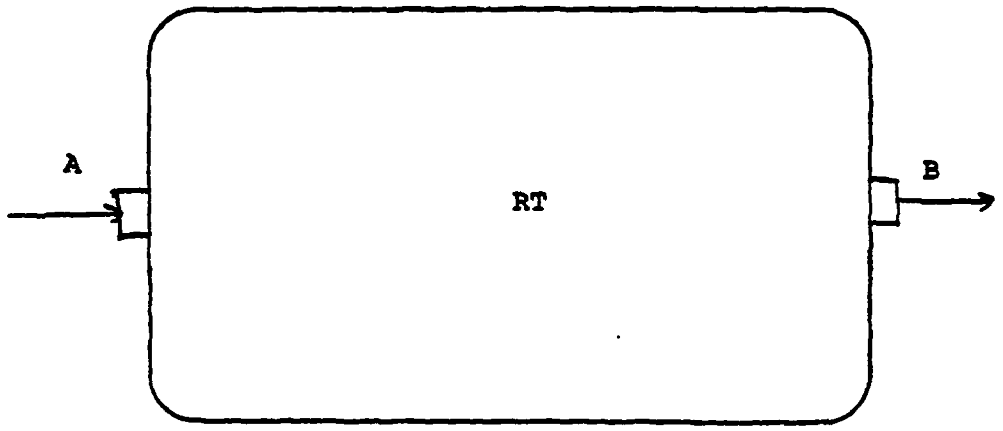


Figure 5. A Reliable Transmission System

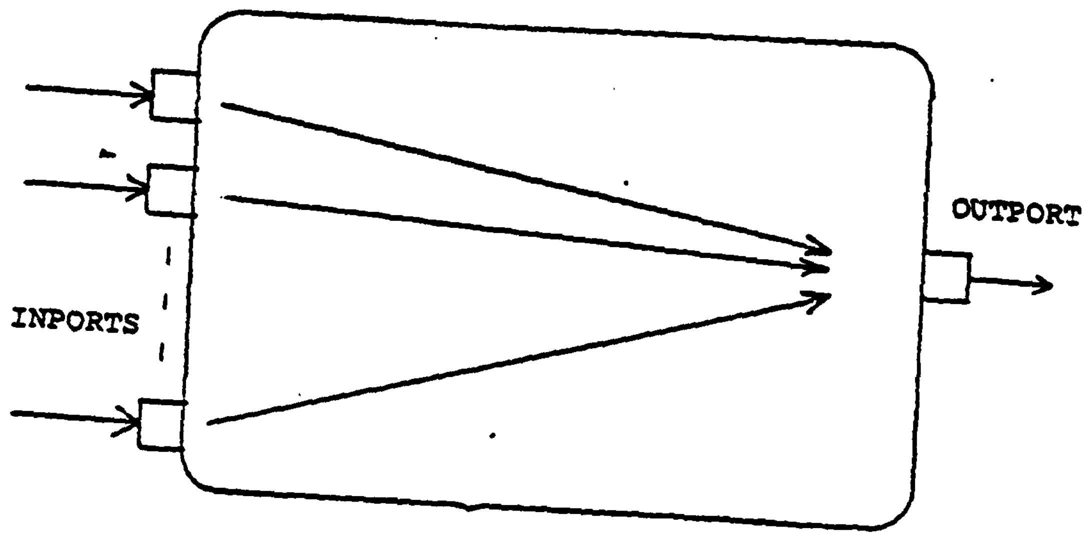


Figure 6. A Multiplexor

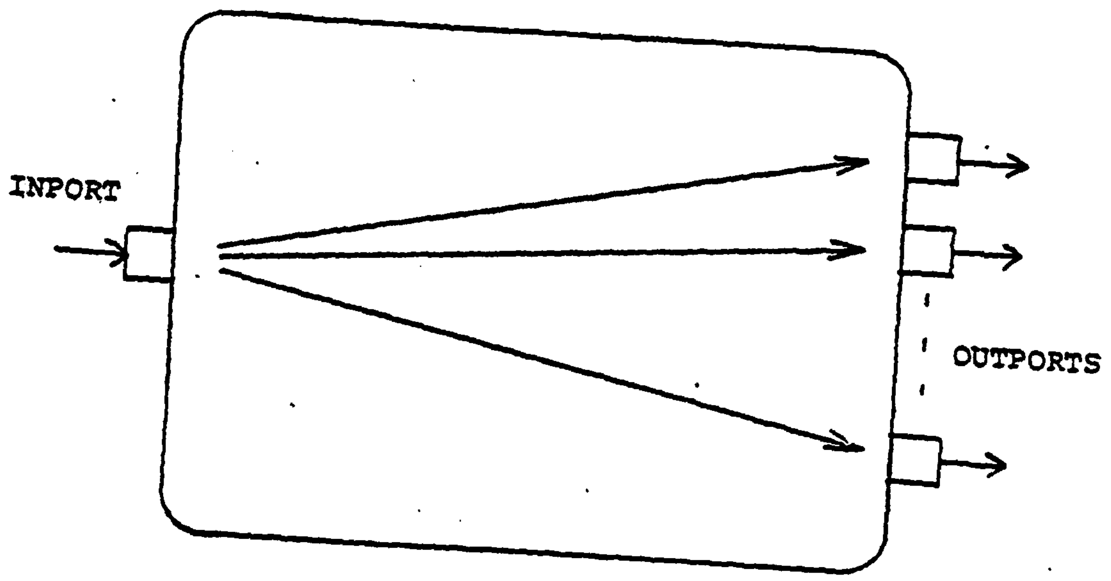


Figure 7. A Decoder

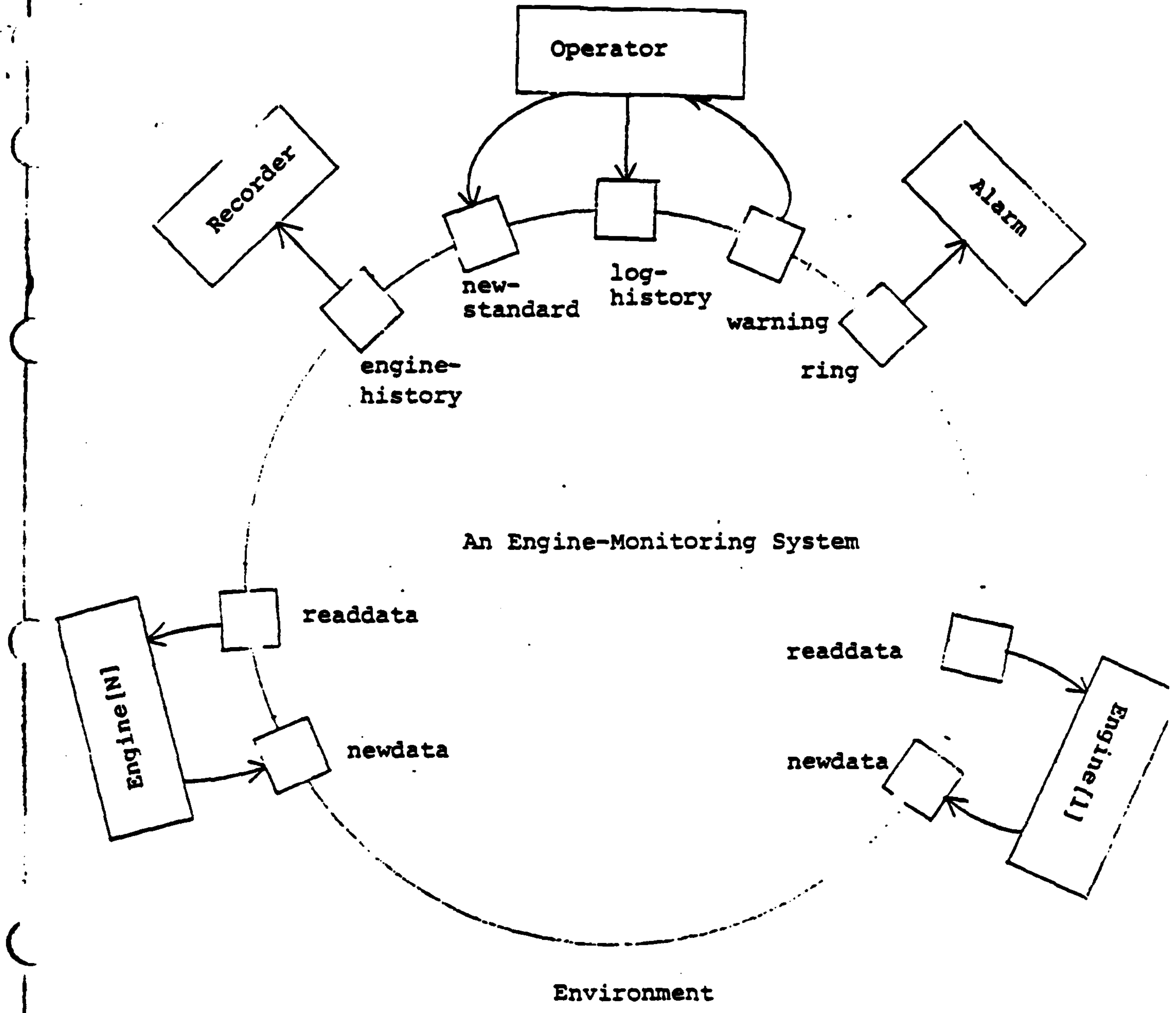


Figure . 8. An Engine-Monitoring System

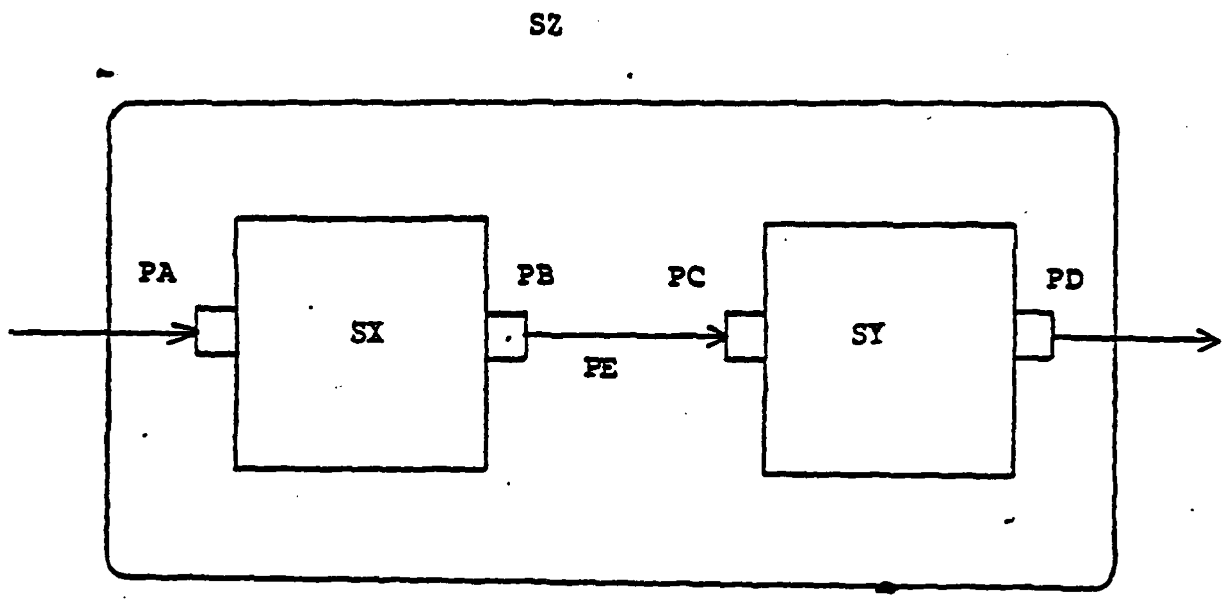


Figure 9. A Tandem Network

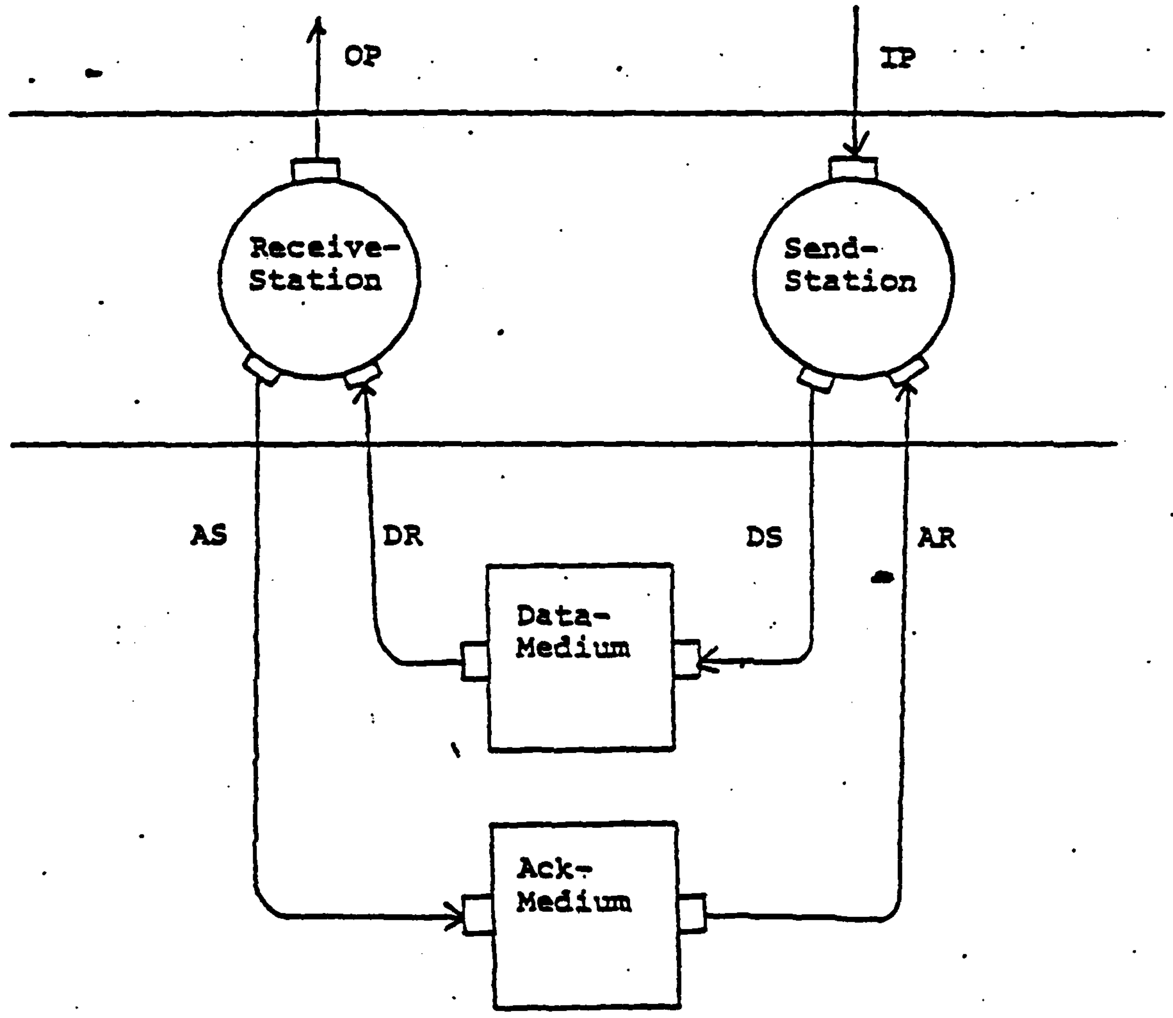


Figure 10 . An Implementation Structure of the Alternate-Bit Protocol

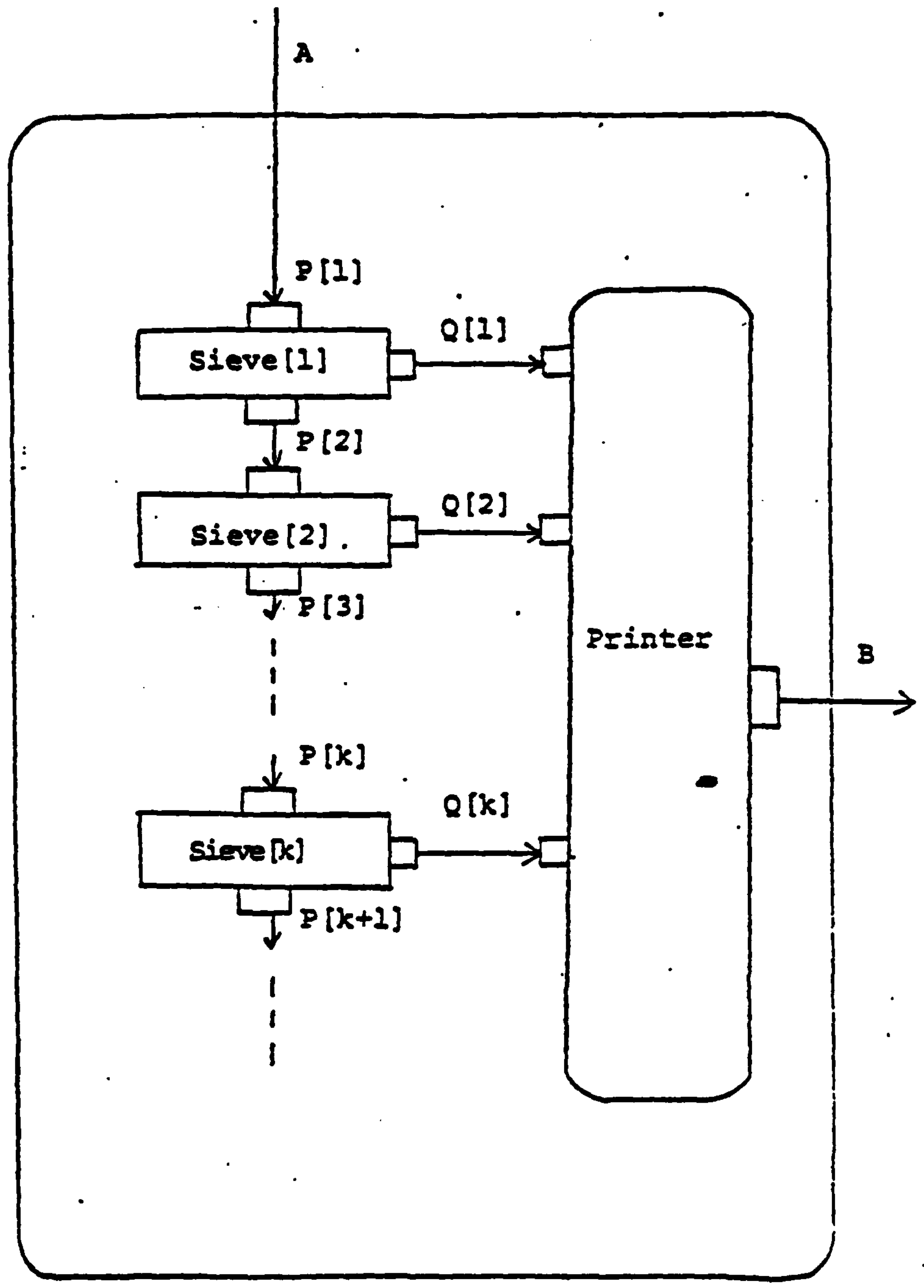


Figure 11. A Distributed Prime Number Generator

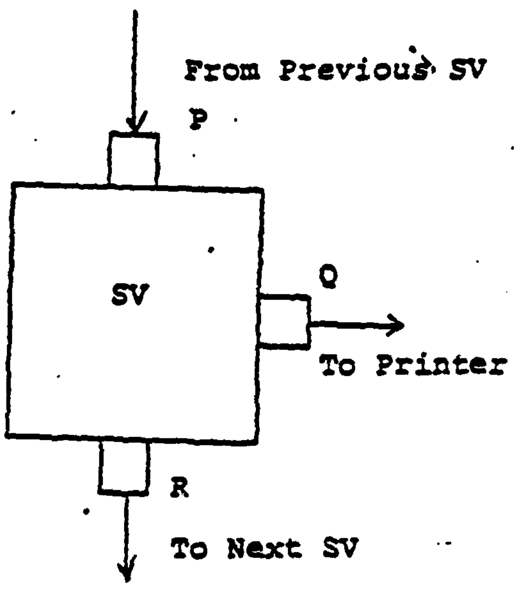


Figure 12. A Sieve Process