https://ntrs.nasa.gov/search.jsp?R=19840023887 2020-03-20T21:11:43+00:00Z

NASA-CR-173839
19840023887

A Reproduced Copy

OF

N84-31957

Reproduced for NASA

by the

NASA Scientific and Technical Information Facility

JUN 7 1988

LANGLEY RESEARCH CENTER
LIBRARY NASA
HAMPTON, VIRGINIA

3 1176 01324 4646

(NASA-CR-173839) SPECIFYING THE BEHAVIOR OF CONCURRENT SYSTEMS (Draper (Charles Stark) Lab., Inc.) 68 p HC A04/MF A01 CSCL 09B

N84-31957

Unclas
G3/61 20167

CSDL-P-1915

SPECIFYING THE BEHAVIOR OF CONCURRENT SYSTEMS

by

Frederick C. Furtek

July 23rd, 1984

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

N84-31957 #

CSDL-P-1915

SPECIFYING THE BEHAVIOR OF CONCURRENT SYSTEMS

by

Frederick C. Furtek

July 23rd, 1984

The Charles Stark Draper Laboratory, Inc.
Cambridge, Massachusetts 02139

ABSTRACT

A framework for rigorously specifying the behavior of concurrent systems is proposed. It is based on the widespread view of a concurrent system as a collection of interacting processes, but unlike previous approaches, no assumptions are made about the mechanisms for process synchronization and communication. One is able to describe the behavioral constraints imposed by such mechanisms without being forced to consider the details of process interaction. A key element of the proposed framework is a formal language that permits the expression of a broad range of logical and timing dependencies, many of which are inexpressible with existing techniques. The language is based on the five logical primitives: 'not', 'and', 'and_next', 'and_next*' and 'reverse'.

ACKNOWLEDGMENT

This work was performed as part of a joint MIT/CSDL program sponsored by the NASA Office of Aeronautics and Space Technology under Grant NAGW-448.

The author is indebted to Daniel Kornhauser, Nancy Lynch, Roger Racine and James Kernan for their invaluable comments and suggestions.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

1.0 INTRODUCTION

A concurrent system, in simplified terms, is a collection of interacting elements. There may be as few as two or three elements, or as many as a thousand or even a million. The elements may be as simple as input switches or indicator lamps, or as complex as entire processors. They may be tightly coupled and located in close proximity to one another, as in a highly parallel computer, or they may be loosely coupled and widely dispersed, as in a nationwide packet-switching network. The interactions may involve the simple synchronization of two elements, or they may entail a complex communication governed by a communication protocol.

A considerable number of attempts have been made to build concurrent systems that fall at different points in this multidimensional spectrum. Some of these attempts have succeeded, but many have been only marginally successful and a few have been outright failures. These experiences reflect a hard fact of life: the tools are not yet in hand that allow us to design concurrent systems in a risk-free fashion. The design of concurrent systems is today a difficult, risky and often painful endeavor.

1.1 Specifying Behavior

While there are undoubtedly several reasons for this state of affairs, one of the principal reasons must surely be our limited ability to specify - in a precise, straightforward way - the behavior of a concurrent system. For example, how do we express for a distributed flight-control system the relationships and dependencies among various sensor outputs, actuator inputs, status bits, and mode switches? The problem is compounded by the fact that some dependencies are functional in nature - engine thrust is a function of throttle-lever position - while other dependencies are temporal in nature - landing gear lowered seven seconds before expected touchdown. Still others combine both functional and temporal requirements.

Because of this limited ability, there is no way to rigorously state the required (intended) behavior of a concurrent system, and without such a formal statement, there is no way to rigorously verify that the actual behavior matches the required behavior. Moreover, there is no way to insure that the requirements are, in fact, consistent. But perhaps most importantly, there is no unambiguous medium for communicating ideas among the sponsors, implementors and users of a system.

To those who object to the need for formal techniques and argue that the present informal methods are sufficient, there are two replies: (1) Informal methods have not been notably successful in alleviating the

serious problems encountered in the design of concurrent systems. (2) Formal (i.e., mathematical) techniques have been extraordinarily successful in a variety of disciplines concerned with modelling system behavior. (One can only wonder where electrical and aeronautical engineering and control theory - to name a few disciplines - would be today without their mathematical underpinnings.)

1.2 Background

We propose a framework for rigorously specifying the behavior of concurrent systems. It is based on the widespread view of a concurrent system as a collection of interacting processes [6] [7] [8] [9] [10] [16] [17] [20] [21] [25] . In this view, the behavior of each process is represented as a sequence of values, which - depending upon the model - are interpreted either as states or events (actions).

Processes interact with one another, and thereby influence each other's behavior, by any one of a number of different mechanisms. It is these synchronization and communication mechanisms that have received the greatest attention. Semaphores [8], *monitors* [16], rendezvous [2] [5] [24], path expressions [6], and exchange functions [9] [25], are some of the methods that have been proposed and investigated. But because all of the above approaches are tied to a particular model of process interaction, they are limited in their generality and expressive power. The specification framework proposed here, however, is independent of the underlying synchronization and communication apparatus. It permits us to describe the behavioral constraints imposed by such mechanisms without forcing us to consider the details of process interaction.

Although this implementation-independent approach increases generality, it also creates a technical problem. We must now be able to represent the composite behavior of a collection of interacting processes. These behavioral representations must reflect the local constraints imposed by individual processes, as well as the global constraints stemming from process interaction. Moreover, it must be possible to express essential constraints on behavior without being forced to include superfluous constraints. For example, it should not be necessary to assign a temporal ordering to two event occurrences if such an ordering is not essential to system behavior - that is, if the two occurrences are 'concurrent'. This last requirement immediately excludes the use of sequences (linear orderings) of states or events to represent concurrent behavior - even though such sequences are used to describe the behavior of individual processes.

The natural solution is to use <u>partial</u> orders on event occurrences (or state holdings) to represent concurrent behavior. In such partial orders, two event occurrences are always ordered if they relate to the same process. If they relate to two different processes, then they may be either ordered or unordered (concurrent). Two interpretations can be

attached to the ordering relation. We may consider the ordering of two occurrences to mean that the first precedes the second in time (which assumes there is a global notion of time). Or we may consider the ordering to mean that there is a causal connection leading from the first to the second (which means that the first precedes the second by every temporal measure).

The use of partial orders to represent concurrent behavior is not novel. There has been research along these lines, [7] [10] [14] [18] , for many years. What is new in the present approach is a technique for characterizing a set of partial orders, a technique that permits us to express a broad range of logical and timing dependencies, many of which are inexpressible within existing approaches.

1.3 Overview

The system model (described in Section 2) provides the basis for a system specification. It is a multiprocess model in which the behavior of an individual process is represented by a sequence of values drawn from the process's 'type'. The composite behavior of an entire system is represented by a partial order on 'instances', each of which associates a process with a value. Instances may be interpreted either as occurrences of events or holdings of states. A partial order on instances is called a 'trace'.

The proposed specification technique has four major components: (1) *Type Definitions*, (2) *Process Declarations*, (3) *Synchronic Structure* and (4) Logical Specification. Each Type Definition (described in Section 3) defines a set of values and a set of operations on those values. The format for Type Definitions is provided by the mechanisms of the Ada[1] programming language for declaring scalar and composite types. Process Declarations (also described in Section 3) assigns to each process a type.

The purpose of the Synchronic Structure and Logical Specification is to specify, through restrictions, the set of permitted (or legal) traces. The restrictions imposed by the Synchronic Structure (described in Section 4) deal only with the structure of a trace when the values associated with instances are ignored. Through the Synchronic Structure, one can assign to a process a metric for time, which provides the basis for expressing timing constraints.

The Logical Specification, in contrast to the Synchronic Structure, deals only with dependencies involving instance values. These dependencies are expressed in a formal language, two versions of which are defined. UPL (for UniProcess Language) (described in Section 5) is

---

[1] Ada is a registered trademark of the U.S. Department of Defense.

the simpler version but is restricted to single-process systems. MPL (for MultiProcess Language) (described in Section 6) has no restrictions but its semantics are more complex than those of UPL. UPL introduces the four logical primitives

not and and_next and_next*

To these four, MPL adds

reverse

In both UPL and MPL, the first four primitives are used to define the five auxiliary constructs

or or_next or_next* implies implies_next

Through these various primitives and constructs, it is possible to express a wide range of logical and timing dependencies. However, because these primitives and constructs are relatively 'low level', UPL and MPL can be extended to include the following sorts of higher-level statements (in which P and Q represent either states or events, depending on context):

- P is followed N time units later by Q.
- Q is inevitable within N time units following P.
- Q for N time units following P.
- Following P, always Q.
- Following P, Q as long as R.
- Following P, Q until R.
- Following P, Q is repeated every N time units.

Each of these statements represents a statement in either standard UPL or standard MPL.

The four components of a system specification - Type Definitions, Process Declarations, Synchronic Structure and Logical Specification - are illustrated (in Section 7) for a simple example: the Alternating-Bit Protocol.

2.0 THE SYSTEM MODEL

A (multiprocess) system is an ordered quadruple $\langle T,P,D,\Sigma \rangle$ where

T is a set of types

P is a set of processes

D is a set of process declarations

Σ is a set of permitted traces

A type is a set of values. Process declarations is a mapping from the set of processes to the set of types. Each process p is thus associated, through its type, with a set of values - denoted Type(p). Both processes and values are considered here to be atomic entities.

Although not essential, it is sometimes useful to consider two classes of processes: event processes and state processes. The values of an event process are interpreted as events, while the values of a state process are viewed as states. A communication port is typical of an event process since the values of the process are most usefully interpreted as the events of sending and receiving particular messages. Sensor outputs, displays, mode switches and status bits, however, are more conveniently represented as state processes since in these cases it is useful to view behavior as a sequence of states.[2] In the parlance of modern software engineering (and the Ada programming language), a state process would be called an 'object'.

The permitted traces of a system represent the allowed (or legal) behaviors of the system. Each trace is a partial order on a set of 'instances'.

2.1 Instances

An instance is a triple $\langle p,v,n \rangle$ where p is a process, v is a value in Type(p), and n is a positive integer.[3] Depending on whether p is interpreted as an event process or state process, $\langle p,v,n \rangle$ can be viewed

---

[2] Note that we are speaking here of 'local' states and not 'global' states.

[3] Adding a positive integer to an instance merely allows us to create distinct instances having the same process and value. The choice of positive integers is arbitrary - any countably infinite set will do.

as either the occurrence of an event or the holding of a state. For the instance <p,v,n>,

$$\text{Process}(\langle p,v,n\rangle) = p$$

$$\text{Value}(\langle p,v,n\rangle) = v$$

We say that Instance <p,v,n> is an instance of Process p. Depending on the interpretation for p - as a state process or event process - <p,v,n> may be regarded as the condition of Process p assuming Value v or as the event of Process p performing the action represented by Value v.

2.2 Traces

A trace is a partial order on a finite set of instances such that all instances belonging to the same process are totally ordered. The restriction on the instances of a process means that the behavior of each process is represented by a (linear) sequence of values.

The instances in a trace, like all instances, each have a positive integer associated with them. However, since these integers have no significance other than to distinguish instances having the same process and value, we consider two traces to be identical if they differ only in their integer assignments.

Let T be a trace defining the partial order $\leq$ over a set I of instances. Then

$$\text{Instances}(T) = I$$

For x,y in Instances(T), x precedes (comes before) y and y follows (comes after) x if $x<y$. x and y are concurrent if x and y are unordered with respect to $\leq$ - that is, if neither $x\leq y$ nor $y\leq x$.

Notice that the foregoing relations depend only on the partial order defined by T and not on the process associated with each instance. That is not the case for two concepts central to our specification approach: the 'next' and 'last' relations. Let $\text{Process}(x)=X$ and $\text{Process}(y)=Y$. Then y is the next instance of Y following x if y follows x and for all z in Instances(T),

$$z \text{ follows } x \text{ and } \text{Process}(z)=\text{Process}(y) \Rightarrow z=y \text{ or } z \text{ follows } y$$

x is the last instance of X preceding y if x precedes y and for all z in Instances(T),

$$z \text{ precedes } y \text{ and } \text{Process}(z)=\text{Process}(x) \Rightarrow z=x \text{ or } z \text{ precedes } x$$

Note that because the instances of each process are totally ordered, if there are any instances of Process Y following (preceding) Instance x, then there must be a next (last) instance following (preceding) x.

2.3 Example of a Trace

A typical trace is illustrated pictorially in Figure 1. Each vertex represents an instance, with the vertex type indicating the associated process. Thus

$$\begin{array}{ll} \text{Process}(a_0) = A & \text{Process}(b_0) = B \\ \text{Process}(a_1) = A & \text{Process}(b_1) = B \\ \text{Process}(a_2) = A & \text{Process}(b_2) = B \\ \text{Process}(a_3) = A & \text{Process}(b_3) = B \end{array}$$

An edge (arrow) leading from Instance x to Instance y means that x precedes y. Notice that the requirement that all instances belonging to the same process be totally ordered is satisfied.

The trace in Figure 1 establishes a number of relationships among the eight instances. A few are listed here:

a_0 precedes a_2

a_2 follows a_0

b_1 precedes a_3

a_3 follows b_1

a_1 and b_2 are concurrent

b_1 and a_2 are concurrent

b_3 is the next instance of Process B following a_1

a_1 is the last instance of Process A preceding b_3

a_3 is the next instance of Process A following b_1

b_2 is the last instance of Process B preceding a_3

Note that the 'next' and 'last' relations are not, in general, converses of one another. If x is an instance of Process X, y an instance of Process Y, and X and Y are not the same process, then the two relations

y is the next instance of Process Y following x

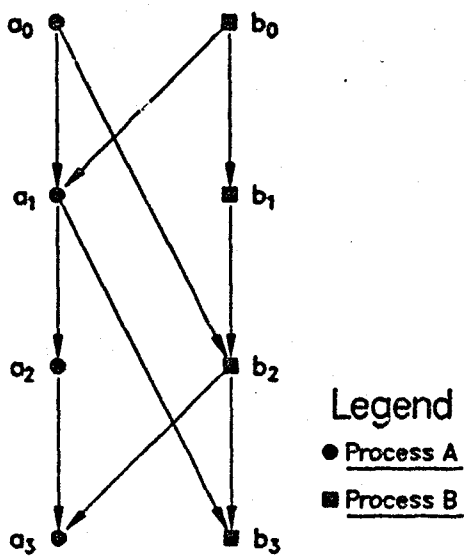


Figure 1. A Trace

x is the last instance of Process X preceding y

are independent. Both relations may hold, neither may hold, or one may hold without the other. For example, in Figure 1:

- b_3 is the next instance of Process B following a_1, and a_1 is the last instance of Process A preceding b_3.
- b_3 is not the next instance of Process B following a_0, and a_0 is not the last instance of Process A preceding b_3.
- a_3 is the next instance of Process A following b_1, but b_1 is not the last instance of Process B preceding a_3.

If X and Y happen to be the same process, then the two relations are equivalent - they both hold or they both fail to hold. Thus, in Figure 1, a_2 is the next instance of Process A following a_1, and a_1 is the last instance of Process A preceding a_2.

Although each vertex in Figure 1 is labelled with the name of an instance, in many cases we may wish to indicate explicitly the process and value associated with the instance. Since the process is already given by the vertex type, we need add only the value. Suppose, for example, that the type of Process A is Integer and that the type of Process B is Boolean. Suppose, furthermore, that

Value(a_0) = 17	Value(b_0) = T
Value(a_1) = -3	Value(b_1) = F
Value(a_2) = 4	Value(b_2) = F
Value(a_3) = 9	Value(b_3) = T

Then the trace in Figure 1 can be made explicit by replacing instance names with instance values as shown in Figure 2.

2.4 Permitted and Prohibited Traces

Central to the notion of system is the idea of constrained behavior. Except for degenerate (and uninteresting) cases and except for malfunctions, the behavior of a system is constrained by certain bounds. Behaviors lying within those bounds are said to be permitted (or allowed, or legal, or possible), while those lying outside those bounds are said to be prohibited (or disallowed, or illegal, or impossible).

In our system model we have chosen to represent system behaviors as partial orders on instances - traces. The set of permitted traces in the definition of a system (see "The System Model" on page 5) thus represent the permitted behaviors of the system.

2.5 System Specification

Having provided a formal model of system behavior, we turn now to the task of formally specifying that behavior. The framework described in the subsequent sections has four components, each of which can be related to the components of the system model. Type Definitions defines the process types, the values associated with each type, and the operations defined on each type. Process Declarations defines the processes and identifies their types. The Synchronic Structure and the Logical Specification together define the set of permitted traces.

3.0 TYPES

A type defines a set of values and a set of operations on those values. How then to specify a type? For a variety of reasons, it seems prudent to adopt the mechanisms of the Ada programming language for declaring scalar and composite types.[4] These mechanisms, which are quite extensive, provide the ability to define the sorts of structures likely to be encountered in specifying system behavior. Furthermore, by adopting Ada syntax, we insure compatibility of the specification

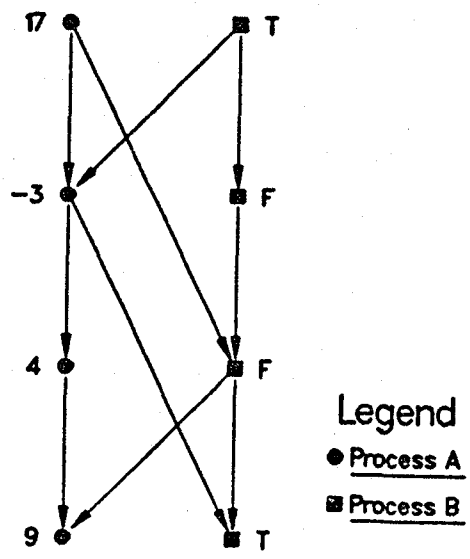


Figure 2. A Trace with Values

language with what is likely to become the pre-eminent programming language for real-time, embedded systems.

We will not attempt to give here a complete description of the Ada constructs for defining scalar and composite types. Any number of references, such as [2], [5] or [24], are adequate for that purpose. It will be helpful, however, to give a brief overview of the scalar and composite types illustrated in Figure 3.

3.1 Enumeration Types

Type definitions take the form

```
type NAME is  ... ;
```

The words 'type' and 'is', as well as all other lower case words in our examples, are reserved words with special meanings in Ada and can be used only as indicated. 'NAME' is a user-supplied word that denotes the name of the type being declared. ... represents the body of the type definition and must be filled in using an appropriate format.

---

[4] In addition to scalar and composite types, Ada also has access, private and task types. These are not needed for specification purposes and are omitted from the discussion.

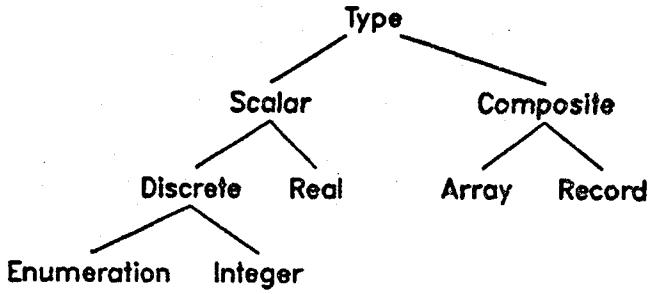


Figure 3. Classification of Types

Enumeration types are the simplest types. The type definition is merely a list of the type's values. Thus

```
type NAME is (VALUE1,VALUE2,VALUE3);
```

declares NAME to be an enumeration type with values: VALUE1, VALUE2, and VALUE3.

Of special interest is the predefined enumeration type BOOLEAN which may be considered to have the definition:

```
type BOOLEAN is (FALSE,TRUE);
```

3.2 Numeric Types

There are two predefined numeric types, INTEGER and REAL. These may be used directly or subsets may be declared using the 'subtype' statement. The statement

```
subtype NAME is INTEGER range M..N;
```

declares NAME to be a subtype of INTEGER that has just those integers in the range M to N inclusive as its values. A similar interpretation holds for the statement

```
subtype NAME is REAL range M..N;
```

3.3 Array Types

The standard format for declaring an array type is:

```
type NAME is array (INDEX1_TYPE,INDEX2_TYPE) of ELEMENT_TYPE;
```

This statement declares NAME to be an array type whose values are two-dimensional arrays. INDEX1 provides the indices for the first dimension, and INDEX2 the indices for the second dimension. ELEMENT_TYPE indicates the type of elements making up the array.

3.4 Record Types

A record is a composite object with named components. A component of a record is accessed through its name using dot notation. If RECORD is a record object with a component named COMPONENT, then RECORD.COMPONENT denotes that component. The simplest format for declaring a record type is:

```
type NAME is
   record
      COMPONENT1: COMPONENT1_TYPE;
      COMPONENT2: COMPONENT2_TYPE;
      COMPONENT3: COMPONENT3_TYPE;
   end record;
```

This statement says that NAME is a record type with three components: COMPONENT1, COMPONENT2 and COMPONENT3. COMPONENTi_TYPE indicates the type of COMPONENTi.

It is sometimes necessary to define a record type in which part of the structure is fixed for all objects of that type and part of the structure is variable. Depending on the value of a special component, called a 'discriminant', the variable part may assume one of several alternative forms. The format for declaring a discriminated-record type is:

```
type NAME (DISCRIMINANT: DISCRIMINANT_TYPE) is
   record
      COMPONENT1: COMPONENT1_TYPE;
      case DISCRIMINANT is
         when DISCRIMINANT_VALUE2 =>
            COMPONENT2: COMPONENT2_TYPE;
         when DISCRIMINANT_VALUE3 =>
            COMPONENT3: COMPONENT3_TYPE;
   end record;
```

Here, DISCRIMINANT is a component of the NAME record type that helps determine the structure of the type. While COMPONENT1 is common to all objects of the type, COMPONENT2 pertains only to those objects for which NAME.DISCRIMINANT takes on the value DISCRIMINANT_VALUE2, and COMPONENT3 pertains only to those objects for which NAME.DISCRIMINANT takes on the value DISCRIMINANT_VALUE3.

3.5 Example of a Type Definition

The Ada syntax just described provides an extensive capability for describing data structures, a capability that is illustrated in the following realistic example. It is a partial definition of the cockpit interface for a typical commercial aircraft. We emphasize that the definition is incomplete.

```
type COCKPIT is
   record
      FLIGHT_CONTROL: FLIGHT_CONTROL_STATE;
      FLIGHT_MANAGEMENT: FLIGHT_MANAGEMENT_STATE;
      AIRCRAFT_SYSTEM_MANAGEMENT: AIRCRAFT_SYSTEM_MANAGEMENT_STATE;
      COMMUNICATION: COMMUNICATION_STATE;
   end record;

type FLIGHT_CONTROL_STATE is
   record
      INERTIAL: INERTIAL_STATE;
      AIR_DATA: AIR_DATA_STATE;
      RADIO_NAVIGATION: RADIO_NAVIGATION_STATE;
      FLIGHT_DIRECTOR: FLIGHT_DIRECTOR_STATE;
      PRIMARY_PILOT_CONTROLS: PRIMARY_PILOT_CONTROLS_STATE;
      SECONDARY_PILOT_CONTROLS: SECONDARY_PILOT_CONTROLS_STATE;
   end record;

type INERTIAL_STATE is
   record
      PITCH: UNITS.DEGREES range -90.0..+90.0;
      ROLE: UNITS.DEGREES range -180.0..+180.0;
      HEADING: UNITS.AZIMUTH_DEGREES;
      RATE_OF_TURN: UNITS.DEGREES_PER_SECOND range -10.0..+10.0;
      SLIP: UNITS.FEET_PER_SECOND2 range -10.0..+10.0;
   end record;

type AIR_DATA_STATE is
   record
      COMPUTED_AIR_SPEED: UNITS.KNOTS range 30.0..450.0;
      MACH_NUMBER: REAL range 0.0..0.9;
      ALTITUDE: UNITS.FEET range -1000..45000;
      VERTICAL_SPEED: UNITS.FEET_PER_MINUTE range -6000.0..+6000.0;
   end record;

type FLIGHT_DIRECTOR_STATE is
   record
      PITCH_COMMAND: DISPLAY_SCALE;
      ROLL_COMMAND: DISPLAY_SCALE;
      SPEED_COMMAND: DISPLAY_SCALE;
      AUTOPILOT: AUTOPILOT_MODE;
   end record;
```

```
type PRIMARY_PILOT_CONTROLS_STATE is
   record
      ROLL_CONTROL_WHEEL: UNITS.DEGREES range -70.0..+70.0;
      PITCH_CONTROL_COLUMN: UNITS.INCHES range -4.0..+10.0;
      RUDDER_PEDAL: UNITS.INCHES range -3.0..+3.0;
   end record;

type MODE_DISCRIMINANT is (DISENGAGED, CONTROL_WHEEL_STEERING, COMMAND);

type AUTOPILOT_MODE (AP_ENGAGE_MODE: MODE_DISCRIMINANT) is
   record
      case AP_ENGAGE_MODE is
         when DISENGAGED =>
            null;
         when CONTROL_WHEEL_STEERING =>
            null;
         when COMMAND =>
            THRUST_SPEED: THRUST_SPEED_SUBMODE;
            VERTICAL: VERTICAL_SUBMODE;
            LATERAL: LATERAL_SUBMODE;
   end record;

type THRUST_SPEED_SUBMODE is
   record
      SPEED_HOLD: BOOLEAN;
      AUTO_THRUST: BOOLEAN;
      COMMANDED_SPEED: KNOTS range 30.0..450.0;
   end record;

type VERTICAL_SUBMODE is
   record
      ALTITUDE_HOLD: BOOLEAN;
      VERTICAL_SPEED: BOOLEAN;
      VERTICAL_NAV: BOOLEAN;
      COMMANDED_SPEED: FEET_PER_MINUTE range -3000.0..+6000.0;
      COMMANDED_ALTITUDE: FEET range 0..45000;
   end record;

type LATERAL_SUBMODE is
   record
      HEADING_HOLD: BOOLEAN;
      VOR: BOOLEAN;
      RNAV: BOOLEAN;
      LOCALIZER: BOOLEAN;
      LAND: BOOLEAN;
      SELECTED_COURSE: UNITS.AZIMUTH_DEGREES;
      COMMANDED_HEADING: UNITS.AZIMUTH_DEGREES;
      RUNWAY_HEADING: UNITS.AZIMUTH_DEGREES;
   end record;
```

```
type UNITS is
   record
      INCHES: REAL;
      FEET: INTEGER;
      FEET_PER_MINUTE: REAL;
      FEET_PER_SECOND2: REAL;
      KNOTS: REAL;
      DEGREES: REAL;
      AZIMUTH_DEGREES: DEGREES range 0.0..359.9;
      DEGREES_PER_SECOND: REAL;
   end record;

type DISPLAY_SCALE is REAL range -100.0..+100.0;
```

3.6 Process Declarations

Once an appropriate set of types has been defined, the processes of a system can be declared. There are three possible formats for a process declaration:

```
PROCESS_NAME is of type TYPE_NAME;

PROCESS_NAME is an event process of type TYPE_NAME;

PROCESS_NAME is a state process of type TYPE_NAME;
```

The first format is used when the values defined by TYPE_NAME are to be left uninterpreted, the second means that the values are to be interpreted as events, while the third means that the values are to be interpreted as states.

4.0 SYNCHRONIC STRUCTURE

The purpose of a synchronic structure - described in this section - and a logical specification - described in the next section - is to specify, through restrictions, the set of permitted traces. The two types of specifications, however, provide two different sorts of restrictions. The constraints imposed by a synchronic structure deal only with the structure of a trace when the values associated with each instance are ignored. For example, one might want to require that between any two instances belonging to Process A there are 7 instances belonging to Process B. This restriction says nothing whatsoever about values. A logical specification, on the other hand, deals entirely with

dependencies involving values. Example: If Process A takes on Value v1, then the next value taken on by Process B is v2.

The class of restrictions that we have chosen to call 'synchronic' is quite large and encompasses many extremely complex relationships. We will not attempt to specify all such restrictions, but will focus instead on a subset of those restrictions that permit us to express logical and timing dependencies in a unified way.

4.1 Synchronous and Asynchronous Processes

So far we have said nothing about time. We have spoken only of instances and partial orders on instances. It is clear, however, that in order to specify real-time behavior, a way must be found to express timing dependencies.

An obvious approach is to augment the definition of a trace by adding durations either to instances - for a state process - or to edges - for an event process. In the first case, the duration represents the duration of a state holding, while in the second case the duration represents the elapsed time between two event occurrences. This approach, while perhaps workable, introduces a second level of discourse for expressing timing dependencies. It means having to express logical relationships and timing relationships using two separate sets of notions, a troublesome situation when the logic and timing of system behavior are intertwined, as is often the case.

By following a slightly different course, it is possible to express both logical and timing requirements within a single, unified framework. This is accomplished by attaching to a process a granularity (of time). For example, we might declare Process A to have a granularity of one nanosecond. If Process A is a state process, then each instance of Process A represents a state holding having a duration of one nanosecond. A state holding with a longer duration is represented as a sequence of instances. Thus, to represent a holding of five nanoseconds, five consecutive instances are required. If Process A is an event process, then one nanosecond represents the elapsed time separating two successive occurrences of Process A. To represent an elapsed time greater than one nanosecond, it is necessary to separate the two occurrences by the appropriate number of instances. (To accomplish this, null events may have to be introduced.)

The format for the synchronic structure of a system is a list of declarations, each of which is in one of the following two forms:

```
PROCESS_NAME is synchronous with granularity T;

PROCESS_NAME is asynchronous;
```

The meaning of the first statement is apparent. The second statement says that there is no metric for time associated with the sequence of instances representing the behavior of PROCESS_NAME. A single system may contain both synchronous and asynchronous processes, and among the synchronous processes there may be several distinct granularities. There is one restriction, however, when there are multiple granularities. We require that for any two granularities T_1 and T_2, either T_1 is an integer multiple of T_2 or vice versa. This restriction is necessitated by the semantics of a synchronic structure.

4.2 Semantics of a Synchronic Structure

As we noted earlier, one of the purposes of a synchronic structure is to restrict the set of permitted traces. We describe now the form that that restriction takes.

Suppose that A and B are two synchronous processes with granularities T_A and T_B, respectively. Assume that $T_A \geq T_B$ and let $n=T_A/T_B$. This means that there are n instances of Process B for every instance of Process A. How can this property be expressed as a restriction on traces? The approach adopted is to require each instance of Process A to be concurrent with n instances of Process B. This idea is illustrated in Figure 4 for the case $T_A/T_B=5$. Notice that there are exactly five instances of Process B - b_3, b_4, b_5, b_6 and b_7 - concurrent with a_1. The remaining instances of Process B either precede a_1 or follow a_1.

The reader will also note that our requirement does not strictly hold for a_0 and a_2. There are only three instances of Process B - b_0, b_1 and b_2 - concurrent with a_0, and likewise only three instances of Process B - b_8, b_9 and b_{10} - concurrent with a_2. This problem, which is related to the finite nature of a trace, is purely technical and can be remedied by a more precise statement of the requirement. The restatement, which will also clear up some other details, is deferred to a subsequent paper.

5.0 LOGICAL SPECIFICATION OF UNIPROCESS SYSTEMS

In Section 6 we describe a language, called MPL (for MultiProcess Language), for specifying both logical and timing dependencies in the context of the system model introduced earlier. Before addressing the multiprocess case, however, it is useful to consider first the uniprocess case - that is, the case where a system has just a single

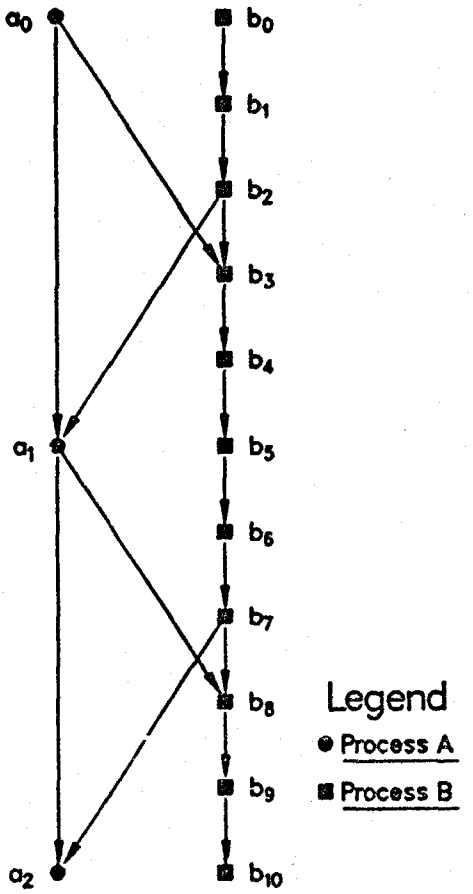


Figure 4. A Trace of Two Synchronous Processes

process. Although the syntax of the languages is essentially the same for both cases, the semantics for uniprocess systems is simpler, requiring less mathematical apparatus. One valuable benefit of this two-step approach is that it permits us to see how the concepts for uniprocess systems generalize in a natural way to multiprocess systems.

Because we are dealing with systems having a single process, the system model can be greatly simplified. A (uniprocess) system is an ordered pair $\langle V,\Sigma \rangle$ where V is a set of values and Σ is a set of permitted value sequences, each of which is of finite length. As in the multiprocess case, values may be interpreted as states or events, or they may simply be left uninterpreted. In addition, the system may be viewed as synchronous or asynchronous.

5.1 Atomic Formulas

UPL (for UniProcess Language) is the language for expressing logical and timing relationships for uniprocess systems, and, like any language, it must have a set of basic building blocks. These are called atomic formulas. Although the precise syntax for these formulas is not important in the present discussion, each such formula must define a predicate on the set of values. The subset of values for which the formula Q holds (is true) is denoted Values(Q). For example, if V is the set of integers and Q is the formula $3<X<7$, where X represents the system process, then Values(Q)={4,5,6}.

5.2 Connectives

UPL is an extension of the language of Boolean expressions. It has four basic connectives: the familiar Boolean connectives 'and' and 'not', the new binary connective 'and_next' and the new unary connective 'and_next*'. The additional connectives 'or', 'or_next', 'or_next*', 'implies' and 'implies_next' are defined as abbreviations:

P or Q	for	not((not P) and (not Q))
P or_next Q	for	not((not P) and_next (not Q))
or_next* Q	for	not(and_next*(not Q))
P implies Q	for	(not P) or Q
P implies_next Q	for	(not P) or_next Q

As a notational convenience when writing long expressions, we adopt the following shortened forms for the various connectives:

not	-	¬
and	-	∧
and_next	-	Δ
and_next*	-	Δ
or	-	∨
or_next	-	∇
or_next*	-	∇
implies	-	->

implies_next - =>

Note that since 'and_next' is a binary connective while 'and_next*' is a unary connective, no confusion should arise from using the same symbol for both connectives. The same observation applies to 'or_next' and 'or_next*'.

5.3 Concatenation

In order to define the semantics for UPL, we need some familiar concepts from formal language theory. (See Hopcroft and Ullman [19].) If α and β are sequences, then $\alpha\beta$ denotes their concatenation. λ denotes the null string, for which the properties $\lambda\alpha=\alpha$ and $\alpha\lambda=\alpha$ hold for all strings α. If A and B are each a set of sequences, then

$$A \cdot B = \{\alpha\beta \mid \alpha \text{ is in } A \text{ and } \beta \text{ is in } B\}$$

Thus, if A = {ab, bb} and B = {b, a, bab}, then A·B = {abb, aba, abbab, bbb, bba, bbbab}.

If A is a set of sequences, then $A^0=\{\lambda\}$ and $A^i=A^{i-1}\cdot A$ for $i>0$. The closure of A, denoted A^*, is the set consisting of the null sequence λ and all finite-length sequences obtained by concatenating sequences in A. Equivalently,

$$A^* = A^0 \cup A^1 \cup A^2 \cdots$$

5.4 Semantics

Each formula of UPL ultimately represents a predicate on V^* (the set of finite-length value sequences). Thus, the ultimate meaning of a formula of UPL is given by the set of value sequences that satisfy the formula. In order to define this set, however, we need to introduce two intermediate quantities for each formula. For a formula P, In(P) and Ex(P), which are both subsets of V^*, represent the 'Included' and 'Excluded' value sequences, respectively, associated with P. In(P) and Ex(P) are defined inductively, first for atomic formulas and then for formulas constructed using each of the four basic connectives. In what follows, Bo(P) denotes In(P) U Ex(P).

5.4.1 Meaning of atomic formulas

Each atomic formula will eventually be interpreted as a predicate on value sequences. Recall, however, that initially each atomic formula represents a predicate on (individual) values, and that Values(P) denotes the set of values that satisfy the atomic formula P. Now interpret each value in Values(P) and in V (the set of all values) as a sequence of length one. Then,[5]

$$In(P) = Values(P)$$

$$Ex(P) = V - Values(P)$$

In(P) is, thus, the set of those sequences of length one whose (only) value satisfies P. Ex(P) is the set of those sequences of length one whose (only) value does not satisfy P.

5.4.2 Meaning of 'not'

The connective 'not' merely interchanges the included and excluded sets for an expression. Thus,

$$In(\text{not } P) = Ex(P)$$

$$Ex(\text{not } P) = In(P)$$

5.4.3 Meaning of 'and'

The definitions of the included and excluded sets for 'P and Q' is consistent with the usual intuition about 'and':

$$In(P \text{ and } Q) = In(P) \cap In(Q)$$

$$Ex(P \text{ and } Q) = Ex(P) \cup Ex(Q)$$

Thus, a sequence is in In(P and Q) if it is in both In(P) and In(Q). The definition of Ex(P and Q) is motivated by the need to have In(P or Q) = In(P) U In(Q) (see below).

Note that if we stopped at this point without introducing the non-classical connectives 'and_next' and 'and_next*', we would have the semantics of classical logic. Let us call a formula classical if it is constructed from the set of atomic formulas using only the classical connectives 'and', 'or', 'not' and 'implies'. Then for each classical formula P, In(P) consists of the set of values that satisfy P in the classical sense, while Ex(P), which is just the set-theoretic complement (with respect to V) of In(P), consists of the set of values that do not

---

[5] If A and B are sets, then A - B is the set containing those elements that are in A but not in B.

satisfy P in the classical sense. Hence, when 'and_next' and 'and_next*' are excluded, 'and', 'or', 'not' and 'implies' can still be used in the classical way.

When 'and_next' and 'and_next*' do become involved, however, In(P) and Ex(P) are no longer necessarily complements of one another, and interpreting the effects of the classical connectives on In(P) and Ex(P) sometimes requires a little thought.

5.4.4 Meaning of 'and_next'

Expressions involving the phrase 'and next' are common in everyday life: "First we'll do this, and next we'll do that." This notion of temporal ordering is captured mathematically using concatenation. If α and β are two sequences, then $\alpha\beta$ embodies the idea 'α and next β'. This theme provides the basis for our definition of the included and excluded sets for 'P and_next Q':[6]

$$In(P \text{ and\_next } Q) = In(P) \cdot In(Q)$$

$$Ex(P \text{ and\_next } Q) = Ex(P) \cdot Bo(Q) \cup Bo(P) \cdot Ex(Q)$$

Each sequence in In(P and_next Q) thus consists of a sequence from In(P) 'and next' a sequence from In(Q). The definition of Ex(P and_next Q) is meant to parallel the definition of Ex(P and Q). Hence, a sequence is in Ex(P and_next Q) if it consists of a sequence from Bo(P) followed by a sequence from Bo(Q) such that either the first sequence is in Ex(P) or the second sequence is in Ex(Q).

To illustrate the definitions for 'and_next', consider the situation where $In(P)=\{a\}$, $Ex(P)=\{b,c\}$, $In(Q)=\{a,b\}$ and $Ex(Q)=\{c\}$. Then

$$In(P \text{ and\_next } Q) = \{aa, ab\}$$

$$Ex(P \text{ and\_next } Q) = \{ba, bb, bc, ca, cb, cc, ac\}$$

Note that all the sequences in In(P), Ex(P), In(Q) and Ex(Q) are of length one, while all the sequences in both In(P and_next Q) and Ex(P and_next Q) are of length two. This is an illustration of a general property: If all the sequences in In(P) and Ex(P) are of length m and all the sequences in In(Q) and Ex(Q) of length n, then all the sequences in both In(P and_next Q) and Ex(P and_next Q) are of length m+n.

We noted earlier that for the case when P is a classical formula, In(P) and Ex(P) are the set-theoretic complements (with respect to V) of one another. We now consider some of the ways in which that simple relationship breaks down when the connective 'and_next' is introduced. Assume P, Q, R and S to be classical formulas throughout the discussion.

---

[6] Recall that $Bo(P) = In(P) \cup Ex(P)$.

Now let F represent the formula 'P and_next Q'. Since In(F) and Ex(F) are not subsets of V, they cannot be set-theoretic complements with respect to V. But they are set-theoretic complements with respect to V^2. In fact, if $P_1 \cdots P_n$ are all classical formulas and F represents the formula 'P_1 and_next $\cdots$ and_next P_n', then In(F) and Ex(F) are set-theoretic complements with respect to V^n.

Now let F represent the formula 'P and (Q and_next R)'. Applying the above definitions, we have

$$In(F) = In(P) \cap (In(Q) \cdot In(R))$$

$$Ex(F) = Ex(P) \cup (Ex(Q) \cdot Bo(R)) \cup (Bo(Q) \cdot Ex(R))$$

Since In(P) contains only sequences of length one and In(Q)·In(R) only sequences of length two, In(F) is empty. Ex(F), on the other hand, is, in general, non-empty and contains an assortment of sequences of length one and length two. There is little that can be said about the relationship between In(F) and Ex(F). They are not set-theoretic complements with respect to any interesting set. They are, however, mutually exclusive. But this is not always the case.

Let a, b, c, d and e be arbitrary values, and let P, Q, R and S be defined such that

a is in In(P)
b is in In(Q)
c is in Ex(Q)
c is in In(R)
d is in Ex(R)
e is in Ex(S)

Now let F represent the formula

$$((PV(P\Delta Q))\Delta Q) \Delta (R\nabla((R\nabla S)\wedge S))$$

(The meanings of 'or' and 'or_next' are given below.) It is easily verified that the sequence abcde is in both In(F) and Ex(F). Although this example may seem counter-intuitive, it presents no problem in defining the semantics for UPL and, in fact, there is a natural interpretation for it (see Section 5.4.11).

5.4.5 Meaning of 'and_next*'

In order to express such temporal relations as "until", "as long as" and "following", we need the ability to represent for a formula P the following infinite expression:[7]

Λ or P or (P and_next P) or (P and_next P and_next P) or $\cdots$

The included set for this expression, obtained using the definitions already given, is

$$\{\lambda\} \cup In(P) \cup In(P)\cdot In(P) \cup In(P)\cdot In(P)\cdot In(P) \cup \cdots$$

while the excluded set is

$$\phi \cap Ex(P) \cap Ex(P \text{ and\_next } P) \cap Ex(P \text{ and\_next } P \text{ and\_next } P) \cap \cdots$$

The first quantity is just $(In(P))^*$, while the second quantity reduces to the null set. We are thus led to the following meaning for 'and_next* P', our represention for the above infinite expression:

$$In(\text{and\_next* } P) = (In(P))^*$$

$$Ex(\text{and\_next* } P) = \phi$$

Notice that for the special case when P is a classical formula, In(and_next* P) is just the set of all value sequences α such that each value in α satisfies P.

5.4.6 Meaning of 'or'

The connectives 'or', 'or_next', 'or_next*', 'implies' and 'implies_next' are all abbreviations for expressions involving the four basic connectives 'not', 'and', 'and_next' and 'and_next*'. The meanings of these five additional connectives, therefore, follow directly from the preceding definitions.

For the connective 'or', we have

$$In(P \text{ or } Q) = In(P) \cup In(Q)$$

$$Ex(P \text{ or } Q) = Ex(P) \cap Ex(Q)$$

A sequence is thus in In(P or Q) if it is in either In(P) or In(Q).

5.4.7 Meaning of 'or_next'

By applying the appropriate manipulations to the meaning of 'and_next', we obtain for 'or_next',

$$In(P \text{ or\_next } Q) = In(P)\cdot Bo(Q) \cup Bo(P)\cdot In(Q)$$

$$Ex(P \text{ or\_next } Q) = Ex(P)\cdot Ex(Q)$$

A sequence is in In(P or_next Q) if it consists of a sequence from Bo(P) followed by a sequence from Bo(Q) such that either the first sequence is in In(P) or the second sequence is in In(Q). Each sequence in

---

[7] Λ denotes the 'null formula'. By convention, $In(\Lambda)=\{\lambda\}$ and $Ex(\Lambda)=\phi$. ϕ denotes the empty set.

Ex(P or_next Q) consists of a sequence from Ex(P) 'and next' a sequence from Ex(Q).

As an illustration, consider the same example we gave for 'and_next', where In(P)={a}, Ex(P)={b,c}, In(Q)={a,b} and Ex(Q)={c}. Then

$$In(P \text{ or\_next } Q) = \{aa, ab, ac, ba, ca, bb, cb\}$$

$$Ex(P \text{ or\_next } Q) = \{bc, cc\}$$

5.4.8 Meaning of 'or_next*'

The definitions for 'or_next*' parallel those for 'and_next*':

$$In(\text{or\_next* } P) = \phi$$

$$Ex(\text{or\_next* } P) = (Ex(P))^*$$

For the special case when P is a classical formula, Ex(or_next* P) consists of all value sequences α such that each value in α satisfies 'not P'.

5.4.9 Meaning of 'implies'

The connective 'implies', which is used almost exclusively in the context of classical formulas, has a meaning that is consistent with that usage:

$$In(P \text{ implies } Q) = Ex(P) \cup In(Q)$$

$$Ex(P \text{ implies } Q) = In(P) \cap Ex(Q)$$

5.4.10 Meaning of 'implies_next'

In specifying the logical behavior of systems, one repeatedly finds the need to express the dependency: "Whenever Behavior 1 occurs, it must be immediately followed by Behavior 2." If we assume Behavior 1 to be represented by Formula P and Behavior 2 by Formula Q, then, as shown below, this dependency can be expressed as 'P implies_next Q'. The definitions that permit this interpretation are:

$$In(P \text{ implies\_next } Q) = Ex(P) \cdot Bo(Q) \cup Bo(P) \cdot In(Q)$$

$$Ex(P \text{ implies\_next } Q) = In(P) \cdot Ex(Q)$$

As an illustration, consider once again the example given above where In(P)={a}, Ex(P)={b,c}, In(Q)={a,b} and Ex(Q)={c}. Then

In(P implies_next Q) = {ba, bb, bc, ca, cb, cc, aa, ab}

Ex(P implies_next Q) = {ac}

5.4.11 Satisfaction and Truth

In the preceding sections, we have shown how to calculate the quantities In(P) and Ex(P) for any formula P in UPL. Recall, however, that ultimately a formula is to be interpreted as a predicate on V^* and that the ultimate meaning of a formula is given by the set of value sequences that 'satisfy' it. We now define, with the aid of Ex(P), what it means for a sequence of values to 'satisfy' a formula P of UPL. We select Ex(P), rather than In(P), for the role because we are interested in expressing properties that constrain, or restrict, the set of permitted value sequences. It is the excluded sequences of a formula that provide these restrictions.

A sequence of values α <u>satisfies</u> a formula P of UPL if and only if α contains no member of Ex(P) as a subsequence.[8] Thus, if β is in Ex(P) and γ is an extension of β (γ contains β as a subsequence), then γ cannot satisfy P. To illustrate this idea, consider the example used several times earlier in which In(P)={a}, Ex(P)={b,c}, In(Q)={a,b} and Ex(Q)={c}. Let F represent the expression 'P implies_next Q'. As noted above, Ex(F)={ac}. A sequence, therefore, satisfies F if and only if it does not have ac as a subsequence. Examples of such sequences are: λ, a, b, c, ca, bb, abc, abab and abbca. Examples of sequences that do not satisfy F are: ac, aac, acb, aacc and bbacb.

The reader will note that because satisfaction is defined using only Ex(P), the possibility that In(P) and Ex(P) may not be set-theoretic complements or that they may intersect presents no technical problems. Some situations do arise, however, that do not exist in classical logic, but 'make sense' in the context of our model. For example, if P is a formula in UPL, then it is possible for a sequence of values to satisfy both P and $\neg$P, or to satisfy neither P nor $\neg$P.[9] To illustrate, let F represent the formula 'P implies_next Q' where P and Q are classical formulas. Because all the sequences in both In(F) and Ex(F) are of length two, all sequences of length one satisfy, by default, both F and $\neg$F. This is natural. If a formula of UPL expresses a constraint on sequences of length n, then we expect all sequences of length less than n to satisfy the formula by default. Now consider any formula F for which In(F) and Ex(F) are both non-empty. Let α be a member of In(F) and β a member of Ex(F). The sequence $\alpha\beta$ then has subsequences in both

---

[8] Sequence α is a <u>subsequence</u> of sequence β if and only if there exist sequences γ and δ such that $\beta=\gamma\alpha\delta$.

[9] Do not confuse the statement 'α does not satisfy P' with the statement 'α satisfies $\neg$P'. The first statement says that α has a subsequence that is in Ex(P), while the second says that no subsequence of α is in Ex($\neg$P).

In(F) and Ex(F). But since In(F)=Ex(¬F), αβ has subsequences in both Ex(F) and Ex(¬F) and hence satisfies neither F nor ¬F. This too is natural. If a formula of UPL expresses a constraint on sequences of length n, then we expect there to be sequences of length greater than n that violate the constraints of F in one location and violate the constraints of ¬F in another location.

The last order of business in providing the semantics for UPL is the notion of 'truth'. A formula P of UPL is <u>true</u> (with regard to a given system) if and only if every permitted value sequence satisfies P. In other words, P is true if and only if no permitted sequence has a member of Ex(P) as a subsequence. True formulas thus specify, by restriction, the set of permitted value sequences, and they are the mechanism by which a logical specification constrains system behavior.

5.5 Algebraic Properties

In much of the preceding discussion, we have tacitly made use of certain 'algebraic' properties of UPL. For example, in writing 'P and_next P and_next P' we assumed that 'and_next' represents, in some sense, an associative operator. Let us say that two formulas of UPL are 'equal' if their included and excluded sets are the same. Then the following algebraic properties involving the classical operations Λ, V and ¬ follow from the above definitions.

$x \Lambda x = x$ and $x V x = x$ (idempotence)

$x \Lambda y = y \Lambda x$ and $x V y = y V x$ (commutativity)

$x \Lambda (y \Lambda z) = (x \Lambda y) \Lambda z$ and $x V (y V z) = (x V y) V z$ (associativity)

$x \Lambda (x V y) = x$ and $x V (x \Lambda y) = x$ (absorption)

$x \Lambda (y V z) = (x \Lambda y) V (x \Lambda z)$ and $x V (y \Lambda z) = (x V y) \Lambda (x V z)$ (distributivity)

$\neg (x \Lambda y) = \neg x V \neg y$ and $\neg (x V y) = \neg x \Lambda \neg y$ (DeMorgan's Laws)

$\neg \neg x = x$ (involution)

An algebra satisfying these properties is called a <u>DeMorgan algebra</u>. (See Balbes and Dwinger [1], Chapter XI.) The interesting thing about a DeMorgan algebra is that it satisfies nearly all the usual properties of a Boolean algebra. In fact, it would be a Boolean algebra with the addition of the Law of the Excluded Middle: $x V \neg x = 1$.

The properties that involve the non-classical connectives Δ, ∇ are:

$x \Delta (y \Delta z) = (x \Delta y) \Delta z$ and $x \nabla (y \nabla z) = (x \nabla y) \nabla z$ (associativity)

$\neg (x \Delta y) = \neg x \nabla \neg y$ and $\neg (x \nabla y) = \neg x \Delta \neg y$ (DeMorgan's Laws)

5.6 Examples of Statements about Uniprocess Behavior

Having provided the formal semantics for UPL, we now show how some common logical/temporal dependencies can be expressed within UPL. For simplicity, we assume in the following examples that P, Q and R are classical formulas - that is, formulas constructed from the set of atomic formulas using only the connectives 'and', 'or', 'not' and 'implies'. In addition, we define T (F) to be an atomic formula that is satisfied by all (no) values.

5.6.1 Example 1 (Invariance)

The simplest assertion that one can make about system behavior has the form: "P is true", where P is a classical formula. From the semantics provided above, it follows that P is true if and only if every value in every permitted value sequence satisfies P. Therefore, saying that P is true is equivalent to saying that P is <u>always</u> true. A statement that is always true is commonly called an <u>invariant</u>.

5.6.2 Example 2

Consider the statement:

"P is followed three values later by Q."

This is a shorthand way of saying: "If a value satisfies P, then the third value following this value must satisfy Q." Or expressed a little differently: "If a value satisfying P is (immediately) followed by two arbitrary values, then the next value must satisfy Q". When stated in this way, the dependency is seen to have the same meaning as the following two (equivalent) formulas of UPL:

```
(P and_next T and_next T) implies_next Q

P implies_next (F or_next F or_next Q)
```

5.6.3 Example 3 (Inevitability)

Consider the statement:

"Q is inevitable within three values following P."

In other words: "If a value satisfies P, then at least one of the next three values must satisfy Q." Expressed a little differently: "If a value satisfies P, then the next value or the next value or the next value must satisfy Q." When put this way, the statement has a direct translation into UPL:

```
P implies_next (Q or_next Q or_next Q)
```

Note the parallel with the second formula in Example 2. Note also that this approach to expressing inevitability does not generalize to 'unbounded' inevitability. To express "Q is inevitable following P" would require the infinite expression: "P implies_next (Q or_next Q or_next ···".

5.6.4 Example 4

Consider the statement:

"Q for three values following P."

Or equivalently: "If a value satisfies P, then the next value and the next value and the next value must all satisfy Q", the obvious translation of which is:

P implies_next (Q and_next Q and_next Q)

This formula, however, does not precisely capture the intended meaning of the above statement. The formula expresses a constraint on value sequences of length four (and, by extension, to sequences of length greater than four), but places no constraints on sequences of length less than four. For example, if the first value of a length-two sequence satisfies P, then the formula imposes no restrictions on the second value - it may, or may not, satisfy Q. A formula that correctly expresses the intended meaning of the above statement is the following:

P implies_next Q

and

(P and_next T) implies_next Q

and

(P and_next T and_next T) implies_next Q

5.6.5 Example 5 (following)

Consider the statement:

"Following P, Q."

This is shorthand for: "If a value satisfies P, then all future values must satisfy Q". Or put another way: If a value satisfying P is followed by a finite number (including zero) of arbitrary values, then the next value must satisfy Q." When expressed in this way, the statement can be translated directly into UPL as:

(P and_next (and_next* T)) implies_next Q

5.6.6 Example 6 (as long as)

Consider the statement:

"Following P, Q as long as R."

This is another way of saying: "If a value satisfies P, then any following value must satisfy Q if that value and all intervening values satisfy R." Or put another way: "If a value satisfying P is (immediately) followed by a finite number (including zero) of consecutive values satisfying R, then the last value (in the sequence of consecutive values satisfying R) must satisfy Q." In UPL the statement becomes:

```
(P and_next (and_next* R)) implies_next (R implies Q)
```

5.6.7 Example 7 (until)

Consider the statement:

"Following P, Q until R."

Which is to say: "If a value satisfies P, then any following value must satisfy Q provided that no intervening value satisfies R." Or equivalently: "If a value satisfying P is (immediately) followed by a finite number (including zero) of consecutive values satisfying 'not R', then the next value must satisfy Q." The corresponding formula in UPL is:

```
(P and_next (and_next* (not R))) implies_next Q
```

5.6.8 Example 8

Consider the statement:

"Q holds for all odd-numbered values following P."

Thus, if a value satisfies P, then the first, third, fifth ··· value following that value must satisfy Q. This constraint is expressed in UPL as:

```
(P and_next (and_next* (T and_next T))) implies_next Q
```

5.6.9 Remarks

We make two observations about the preceding examples. First, except for Example 1, each statement is in the form of an implication stating what must be true in the future given some current condition. It should be clear, however, that there are analogous statements involving past behavior. For example, "Preceding P, Q" is the counterpart to "Following P, Q". Such statements, which are as

legitimate as those predicting future behavior, are expressible within UPL because the language has no fundamental bias towards either the past or the future.

The second observation about the above examples concerns their interpretation when the system process is declared to be synchronous and is assigned a granularity. With such a declaration, formulas of UPL take on a temporal meaning. Suppose, for example, that the system process is declared to be synchronous with a granularity of one millisecond. Then the formulas in Examples 2, 3 and 4 can be reinterpreted as follows:

"P is followed three milliseconds later by Q."

"Q is inevitable within three milliseconds following P."

"Q for three milliseconds following P."

Such is the way in which logical and timing constraints are integrated into a single framework.

6.0 LOGICAL SPECIFICATION OF MULTIPROCESS SYSTEMS

MPL (for MultiProcess Language) is the language for specifying both logical and timing dependencies in multiprocess systems. It has much in common with UPL. Except for the addition of a unary connective (called 'reverse'), the syntax of MPL is identical to that of UPL. The basic connectives 'not', 'and', 'and_next' and 'and_next*' are used in the same way, while the auxiliary connectives 'or', 'or_next', 'or_next*, 'implies' and 'implies_next' are defined as the same abbreviations.

Like the uniprocess case, the semantics for each formula P of MPL is given by the two sets In(P) and Ex(P) which, again like the uniprocess case, are defined inductively, first for atomic formulas and then for formulas constructed using each of the basic connectives. Furthermore, In(P) and Ex(P) retain their original forms - as expressions involving union, intersection and concatenation - when P is constructed using any of the original UPL connectives. For example, Ex(P and_next Q) = Ex(P)·Bo(Q) U Bo(P)·Ex(Q) in both the uniprocess and the multiprocess case. However, because the formulas of MPL are eventually to be interpreted as predicates on (partially ordered) traces rather than predicates on (totally ordered) value sequences, it is necessary to use a different type of structure within the included and excluded sets for a formula P. In the uniprocess case, In(P) and Ex(P) are each a set of value sequences, while in the multiprocess case, In(P) and Ex(P) are each a set of trace-like objects, called 'templates'. This change from sequences to templates entails two modifications to uniprocess

semantics: (1) the included and excluded sets for atomic formulas must be redefined and (2) the notion of concatenation must be adapted to templates.

The following sections focus primarily on the structure of templates, their use in included and excluded sets, and their ultimate interpretation as 'templates' for traces.

6.1 MPL Syntax

As in the case of uniprocess systems, we assume the existence of a set of atomic formulas. As above, we are not concerned with the particular syntax for atomic formulas, but we do assume that each atomic formula Q is associated with a unique process - denoted Process(Q) - and represents a predicate on the values belonging to Process(Q)'s type. The subset of values in Type(Process(Q)) for which Q holds (is true) is denoted Values(Q).

Composite formulas of MPL are constructed from the set of atomic formulas using the four basic connectives of UPL - 'not', 'and', 'and_next' and 'and_next* - plus the new unary connective 'reverse' - denoted ~. The auxiliary connectives 'or', 'or_next', 'or_next*, 'implies' and 'implies_next' retain their original meanings as abbreviations.

6.2 Templates

Informally, a 'template' is a directed graph whose vertices are instances[10] and whose edges (arrows) come in two types: those labelled 'next' and those labelled 'last'. Formally, a template is an ordered triple <I,N,L> where I is a finite set of instances and where N and L are each a set of directed edges on I. Although there are no restrictions on the structure of a template, only those templates that correspond to partial orders - those without (directed) circuits - will be of interest. For a template <I,N,L>,

[10] Recall that an instance is a triple <p,v,n> where p is a process, v is a value in Type(p), and n is a positive integer.

$$\text{Instances}(\langle I,N,L\rangle) = I$$

$$\text{Next}(\langle I,N,L\rangle) = N$$

$$\text{Last}(\langle I,N,L\rangle) = L$$

The head (tail) of a template is the set of those instances that have no emergent (entrant) edges. The null template $\langle\phi,\phi,\phi\rangle$ is denoted by λ. To illustrate these ideas, let T be the template depicted in Figure 5. Then

$$\text{Instances}(T) = \{x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7\}$$

$$\text{Head}(T) = \{x_6, x_7\}$$

$$\text{Tail}(T) = \{x_0, x_1\}$$

$$\text{Next}(T) = \{\langle x_0,x_2\rangle, \langle x_1,x_3\rangle, \langle x_2,x_4\rangle, \langle x_3,x_4\rangle\}$$

$$\text{Last}(T) = \{\langle x_4,x_5\rangle, \langle x_5,x_6\rangle, \langle x_5,x_7\rangle\}$$

6.3 A Partial Order on Templates

In defining In(P) and Ex(P) for a formula P of MPL, we will make use of a partial order on templates which is intended to capture the idea of one template being a 'simpler' or less 'restrictive' version of another. The ordering depends on the notion of a 'morphism' between two templates.

Let T_1 and T_2 be templates. A mapping ψ from $\text{Instances}(T_1)$ to $\text{Instances}(T_2)$ is called a morphism from T_1 to T_2 if for all x,y in $\text{Instances}(T_1)$,

- $\text{Process}(x) = \text{Process}(\psi(x))$
- $\text{Value}(x) = \text{Value}(\psi(x))$
- $\langle x,y\rangle$ in $\text{Next}(T_1) \Rightarrow \langle\psi(x),\psi(y)\rangle$ in $\text{Next}(T_2)$
- $\langle x,y\rangle$ in $\text{Last}(T_1) \Rightarrow \langle\psi(x),\psi(y)\rangle$ in $\text{Last}(T_2)$
- x in $\text{Head}(T_1) \Rightarrow \psi(x)$ in $\text{Head}(T_2)$
- x in $\text{Tail}(T_1) \Rightarrow \psi(x)$ in $\text{Tail}(T_2)$

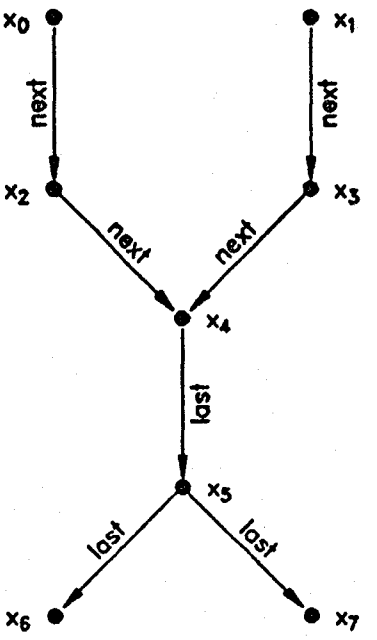


Figure 5. A Template

If the mapping Ψ is one-to-one, then the morphism is also said to be one-to-one.[11]

We now use the notion of morphism to define the partial order $\leq$ on templates.[12] If T_1 and T_2 are templates, then $T_1 \leq T_2$ if and only if at least one of the following conditions holds:

(1) T_1 and T_2 are both the null template.

(2) T_1 and T_2 are both non-null and there exists a one-to-one morphism from T_1 to T_2.

(3) T_1 and T_2 are both non-null and there exists a morphism from T_1 to T_2 but no morphism from T_2 to T_1.

We note first that the null template λ is isolated by $\leq$ from all other templates. Condition 2 says, in effect, that there is an exact copy (except for instance numbering) of T_1 embedded in T_2 and that the head (tail) of this copy is contained in the head (tail) of T_2. Condition 3

---

[11] A mapping Ψ is one-to-one if distinct elements in the domain of Ψ have distinct images under Ψ.

[12] In claiming that $\leq$ is a partial order, we consider two templates to be identical if they differ only in their instance numbers.

says that there is a 'collapsed' version of T_1 embedded in T_2 and that the head (tail) of this version is contained in the head (tail) of T_2. Condition 3 also requires that there be no similar version of T_2 embedded in T_1.

To illustrate the concept of template ordering, consider the three templates in Figure 6. Assume that like-named instances in separate templates have the same process and value. Assume, furthermore, that $Process(x_0)=Process(x_0')$ and that $Value(x_0)=Value(x_0')$ Notice that an identical copy of T_1 is embedded in T_2 and that the head (tail) of this copy is contained in the head (tail) of T_2. Thus, $T_1 \leq T_2$ by Condition 2. (This relationship also follows from Condition 3 if x_1 and x_2 in T_2 differ in either process or value.) Now notice that there is a 'collapsed' version of T_2 embedded in T_3 and that the head (tail) of this version is contained in the head (tail) of T_3. Notice also that there is no similar version of T_3 embedded in T_2 (assuming that x_1 and x_2 differ in either process or value). Hence, $T_2 \leq T_3$ by Condition 3.

To help motivate the last requirement in Condition 3, suppose that x_1 and x_2 in template T_2 of Figure 6 have the same process and value. There is then a morphism from T_2 to T_1. Now if the last requirement in Condition 3 were omitted, it would follow that $T_2 \leq T_1$. But because $T_1 \leq T_2$ by Condition 2, the antisymmetry property of $\leq$ would be violated. Hence, the need for the last requirement in Condition 3.

6.4 Concatenation of Templates

Concatenation of templates is analogous to concatenation of sequences. If T_1 and T_2 are templates, then $T_1 \cdot T_2 = \langle I,N,L \rangle$ where[13]

$$I = Instances(T_1) \cup Instances(T_2)$$

$$N = Next(T_1) \cup Next(T_2) \cup (Head(T_1) \times Tail(T_2))$$

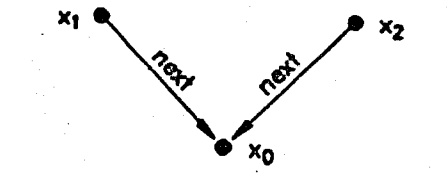
$$L = Last(T_1) \cup Last(T_2)$$

$T_1 \cdot T_2$ is thus obtained by connecting the head of T_1 to the tail of T_2 with a set of 'next' edges.

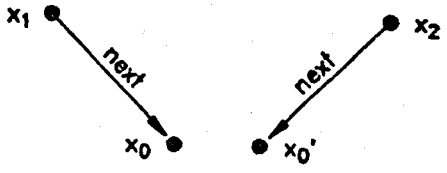
As an illustration of concatenation, consider the two templates T_1 and T_2 in Figure 7. T_1 consists of the instances x_0, x_1, x_2 and x_3 and the two edges connecting those instances. T_2 consists of the instances x_4, x_5, x_6 and x_7 and the three edges connecting those instances. $T_1 \cdot T_2$ is the composite graph obtained by connecting T_1 to T_2 with the two edges $\langle x_2,x_4 \rangle$ and $\langle x_3,x_4 \rangle$, which are indicated by dashed lines.

---

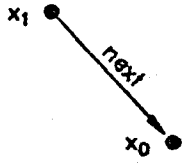
[13] $\times$ denotes Cartesian product.



(c) Template T_3



(b) Template T_2



(a) Template T_1

Figure 6. Three Ordered Templates

As in the case of sequences, sets of templates can also be 'concatenated'. If T_1 and T_2 are each a set of templates, then $T_1 \cdot T_2$ is the set of all those templates T for which there exist T_1 in T_1 and T_2 in T_2 such that Instances$(T_1) \cap$ Instances$(T_2) = \phi$ and $T_1 \cdot T_2 \leq T$. [14] (Note that $\{T_1\} \cdot \{T_2\} \neq \{T_1 \cdot T_2\}$.) T^n and T^* are defined as before for sequences.

[14] When concatenating two templates, we want to consider them as two separate and distinct objects. This cannot be done, however, when there is a naming conflict between the instances of the two templates. Thus the requirement in the above definition that Instances(T_1) and Instances(T_2) be disjoint. For the sets of templates we will be considering, no concatenated templates will be lost because of this restriction since it will always be possible to replace conflicting templates by equivalent, non-conflicting ones.

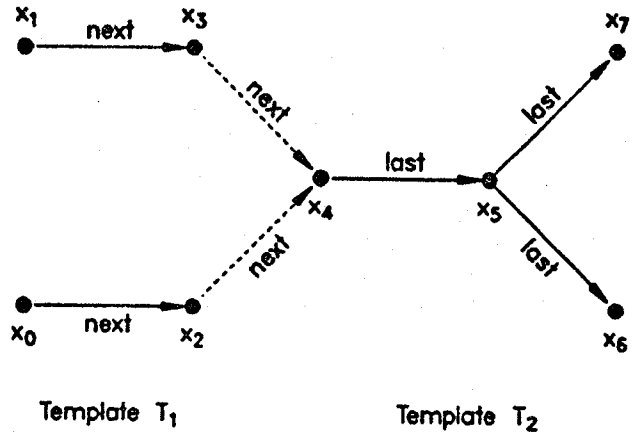


Figure 7. Concatenating Two Templates

6.5 Well-Structured Sets of Templates

As already noted, In(P) and Ex(P), for a formula P of MPL, will be defined as sets of templates. These sets will turn out to have two important properties: 'upward closure' and the 'minimality condition'.

Let X be a set of elements partially ordered by ≤. X is said to be upwardly closed (with respect to ≤) if

$$x \leq y \text{ and } x \text{ in } X \Rightarrow y \text{ in } X$$

X satisfies the minimality condition (with respect to ≤) if for each element y in X there exists a minimal element x of X such that $x \leq y$.[15] When X is both upwardly closed and satisfies the minimality condition, we say that it is well structured. An important property of a well-structured set X is that it can be characterized by its set of minimal elements - denoted Min(X).

The following are four important results relating to well-structured sets of templates.

PROPERTY 1. If T_1 and T_2 are well-structured sets of templates, then $T_1 \cup T_2$, $T_1 \cap T_2$ and $T_1 \cdot T_2$ are also well-structured sets of templates.

In other words, the operations of union, intersection and concatenation preserve well-structuredness.

---

[15] Note that a minimal element - unlike a minimum element - need not be unique. A set may have more than one minimal element.

PROPERTY 2. If T_1 and T_2 are well-structured sets of templates, then

$$\mathrm{Min}(T_1 \cup T_2) = \mathrm{Min}(\mathrm{Min}(T_1) \cup \mathrm{Min}(T_2))$$

Thus, to find the minimal templates of $T_1 \cup T_2$ we need only look for the minimal templates in $\mathrm{Min}(T_1) \cup \mathrm{Min}(T_2)$.

PROPERTY 3. If T_1 and T_2 are well-structured sets of templates, then $\mathrm{Min}(T_1 \cap T_2) = \mathrm{Min}(A)$ where A is the set of templates T for which there exist templates T_1 and T_2 such that

- T_1 is in $\mathrm{Min}(T_1)$ and T_2 is in $\mathrm{Min}(T_2)$
- $T_1 \leq T$ by morphism ψ_1 and $T_2 \leq T$ by morphism ψ_2
- Each instance in T is the image under ψ_1 of an instance in T_1 or the image under ψ_2 of an instance in T_2.
- Each 'next' ('last') edge in T is the image under ψ_1 of a 'next' ('last') edge in T_1 or the image under ψ_2 of a 'next' ('last') edge in T_2.

Property 3 says something non-obvious. It says that to find the minimal templates of $T_1 \cap T_2$ we do not look for the minimal templates in $\mathrm{Min}(T_1) \cap \mathrm{Min}(T_2)$. Instead, we look among the templates formed by 'merging' a template from $\mathrm{Min}(T_1)$ and a template from $\mathrm{Min}(T_2)$.

PROPERTY 4. If T_1 and T_2 are well-structured sets of templates, then

$$\mathrm{Min}(T_1 \cdot T_2) = \mathrm{Min}(\mathrm{Min}(T_1) \cdot \mathrm{Min}(T_2))$$

The minimal templates of $T_1 \cdot T_2$ can, thus, all be found in $\mathrm{Min}(T_1) \cdot \mathrm{Min}(T_2)$.

6.6 Semantics

Each formula of MPL will ultimately represent a predicate on the set of traces. Thus, the ultimate meaning of a formula in MPL is given by the set of traces that satisfy the formula. However, as in the case of UPL, we need first to introduce the intermediate quantities In(P) and Ex(P) for each formula P of MPL. For UPL, these quantities were sets of sequences, but for MPL they are sets of templates. As with UPL, In(P) and Ex(P) are defined inductively, first for atomic formulas and then

for formulas constructed using each of the five basic connectives. (Recall that we now have an additional basic connective, 'reverse'.)

For all cases, In(P) and Ex(P) will be well-structured sets of templates. The definitions of these two quantities for the case when P is an atomic formula will insure that the property is met for the basic building blocks of HPL. The definitions of In(P) and Ex(P) for composite formulas will all be expressed in terms of the operations of union, intersection and concatenation, each of which preserves well-structuredness (Property 1). By having the included and excluded sets of all formulas well-structured, we are able to understand the meanings of the various connectives in terms of their effects on minimal templates.

6.6.1 Meaning of atomic formulas

Let P be an atomic formula. P, therefore, has associated with it a process - denoted Process(P) - and a set of values - denoted Values(P). Let V = Type(Process(P)). For the uniprocess case, In(P) is defined as the set of those sequences of length one whose (only) value is in Values(P). Ex(P) is defined as the set of those sequences of length one whose (only) value is in V-Values(P).

For the multiprocess case, In(P) is the set of all those templates T for which there exists an instance x such that

- Process(x) = Process(P)
- Value(x) is in Values(P)
- $\langle\{x\},\phi,\phi\rangle \leq T$

In other words, In(P) is the set of all those templates T that have embedded within them an isolated instance x such that Process(x) = Process(P) and Value(x) is in Values(P). (Note that x is in both the head and tail of T.) From the definition it follows that In(P) is well structured and, therefore, is characterized by its minimal templates. These are easily described. They are all templates of the form $\langle\{x\},\phi,\phi\rangle$ where Process(x) = Process(P) and where Value(x) is in Values(P). These minimal templates are in one-to-one correspondence (ignoring instance numbers) with the length-one sequences that define In(P) for the uniprocess case.

Ex(P) in the multiprocess case is the set of all those templates T for which there exists an instance x such that

- Process(x) = Process(P)
- Value(x) is in V-Values(P)
- $\langle\{x\},\phi,\phi\rangle \leq T$

Ex(P) is, thus, the set of all those templates T that have embedded within them an isolated instance x such that Process(x) = Process(P) and Value(x) is in V-Values(P). Like In(P), Ex(P) is well-structured and its minimal templates are easily described. They are all templates of the form <{x},ϕ,ϕ> where Process(x) = Process(P) and where Value(x) is in V-Values(P).

6.6.2 Meaning of 'not'

As in the uniprocess case, the connective 'not' simply interchanges the included and excluded sets of an expression. Thus,

$$In(\text{not } P) = Ex(P)$$

$$Ex(\text{not } P) = In(P)$$

6.6.3 Meaning of 'and'

The definitions of the included and excluded sets for 'P and Q' have the same forms used for the uniprocess case:

$$In(P \text{ and } Q) = In(P) \cap In(Q)$$

$$Ex(P \text{ and } Q) = Ex(P) \cup Ex(Q)$$

Thus, a template is in In(P and Q) if it is in both In(P) and In(Q), and is in Ex(P and Q) if it is in either Ex(P) or Ex(Q). To understand these definitions and to see how they differ from the ones given earlier for UPL, let us look at the definitions in terms of minimal templates.

Suppose that P and Q are atomic formulas. Then the minimal templates in In(P), In(Q), Ex(P) and Ex(Q) are all of the form <{x},ϕ,ϕ>. From Property 3, it follows that Min(In(P and Q)) contains two classes of templates:

(1) those in Min(In(P)) ∩ Min(In(Q))

(2) those of the form <{x,y},ϕ,ϕ> such that <{x},ϕ,ϕ> is in Min(In(P)) but not Min(In(Q)) and <{y},ϕ,ϕ> is in Min(In(Q)) but not Min(In(P))

In analyzing these classes, two cases need to be considered: Process(P) = Process(Q) and Process(P) ≠ Process(Q). When Process(P) = Process(Q), the templates in Min(In(P)) ∩ Min(In(Q)) are in one-to-one correspondence (ignoring instance numbers) with the values in Values(P) ∩ Values(Q). This corresponds to the classical, uniprocess case. The templates in the second class above, however, represent a divergence from the classical, uniprocess view.[16] When Process(P) = Process(Q), these templates are in one-to-one correspondence with pairs of values {x,y} such that x is in Values(P)-Values(Q) and y is in Values(Q)-Values(P). (Such 'nonclassical' templates turn out to be useful in expressing certain logical dependencies.) For the case above

when Process(P) ≠ Process(Q), the templates in the second class are the only templates in Min(In(P and Q)) since Min(In(P)) ∩ Min(In(Q)) is empty. These templates are all templates of the form <{x,y},ϕ,ϕ> such that <{x},ϕ,ϕ> is in Min(In(P)) and <{y},ϕ,ϕ> is in Min(In(Q)).

The situation regarding Ex(P and_next Q) is considerably simpler than that for In(P and_next P). From Property 2, we see that

$$\text{Min}(\text{Ex}(P \text{ and } Q)) = \text{Min}(\text{Ex}(P)) \cup \text{Min}(\text{Ex}(Q))$$

(for both atomic and non-atomic formulas). Thus, the definition of Ex(P and_next Q) for MPL corresponds completely to the definition for UPL.

Having considered the case when P and Q are both atomic, let us consider a second special case. Suppose that the templates in In(P) and In(Q) are Independent in the sense that no instance appearing in a template of In(P) has the same process and value as an instance appearing in a template of In(Q). From Property 3, it follows that Min(In(P and Q)) = Min(A) where A is the set of composite templates obtained by 'juxtaposing' a template from In(P) with a template from In(Q). For example, if Min(In(P)) consists of the single template shown in Figure 8(a), if Min(In(Q)) consists of the single template shown in Figure 8(b) and if x_0 and x_1 differ from x_2 and x_3 in either process or value, then Min(In(P and Q)) consists of the composite template shown in Figure 8(c).

6.6.4 Meaning of 'and_next'

The meaning of 'and_next' in the context of MPL parallels the definition given earlier for UPL. There is no change in the expressions defining the included and excluded sets but concatenation now applies to templates instead of sequences. Hence

$$\text{In}(P \text{ and\_next } Q) = \text{In}(P) \cdot \text{In}(Q)$$

$$\text{Ex}(P \text{ and\_next } Q) = \text{Ex}(P) \cdot \text{Bo}(Q) \cup \text{Bo}(P) \cdot \text{Ex}(Q)$$

Each template in In(P and_next Q) thus consists of a template from In(P) 'and next' a template from In(Q). A template is in Ex(P and_next Q) if it consists of a template from Bo(P) 'and next' a template from Bo(Q) such that either the first template is in Ex(P) or the second template is in Ex(Q).

As we did for 'and', let us consider the special case when P and Q are both atomic. The minimal templates in In(P), In(Q), Ex(P) and Ex(Q) are then all of the form <{x},ϕ,ϕ>. From Property 4, it follows that the templates in Min(In(P and_next Q)) are all those with the structure

---

[16] This departure from UPL changes the meanings of certain formulas representing invariants (see ection 6.8.1).

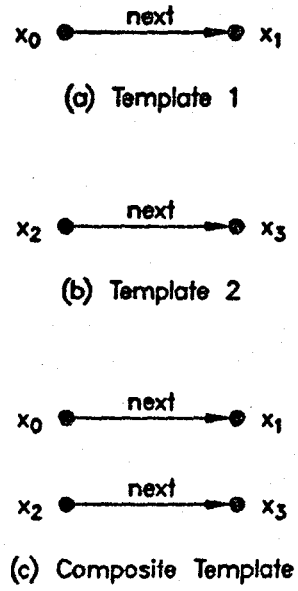


Figure 8. Juxtaposing Two Templates

<{x,y}, {<x,y>}, ϕ> such that <{x},ϕ,ϕ> is in Min(In(P)) and <{y},ϕ,ϕ> is in Min(In(Q)). It also follows that the templates in Min(Ex(P and_next Q)) are all those with the structure <{x,y}, {<x,y>}, ϕ> such that either (1) <{x},ϕ,ϕ> is in Min(Ex(P)) and <{y},ϕ,ϕ> is in Min(Bo(Q)) or (2) <{x},ϕ,ϕ> is in Min(Bo(P)) and <{y},ϕ,ϕ> is in Min(Ex(Q)). As an illustration, let Min(In(P)) consist of the two single-instance templates shown in Figure 9(a), Min(Ex(P)) the single-instance template in Figure 9(b), Min(In(Q)) the two single-instance templates in Figure 9(c), and Min(Ex(Q)) the single-instance template in Figure 9(d). Then Min(In(P and_next Q)) consists of the four single-edge templates in Figure 9(e) and Min(Ex(P and_next Q)) the five single-edge templates in Figure 9(f). Note the parallel with the UPL example in Section 5.4.4. There, the elements of In(P and_next Q) and Ex(P and_next Q) are value sequences of length two. Here, the elements of Min(In(P and_next Q)) and Min(Ex(P and_next Q)) are templates, each consisting of two instances connected by a 'next' edge.

The principles illustrated for the case when P and Q are both atomic extend in a straightforward way to the case when either P or Q is non-atomic.

6.6.5 Meaning of 'and_next*'

As in the uniprocess case, there is a need to represent the infinite formula:

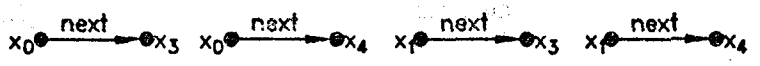
Λ or P or (P and_next P) or (P and_next P and_next P) or ···

(a) Templates in Min(In(P))

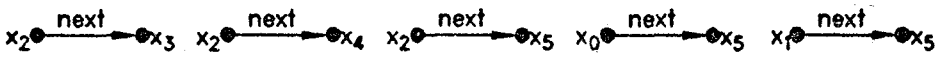
(b) Templates in Min(Ex(P))

(c) Templates in Min(In(Q))

(d) Templates in Min(Ex(Q))



(e) Templates in Min(In(P and_next Q))



(f) Templates in Min(Ex(P and_next Q))

Figure 9. Meaning of 'P and_next Q'

As before, the included set for this formula is given by the infinite expression

$$\{\lambda\} \cup In(P) \cup In(P)\cdot In(P) \cup In(P)\cdot In(P)\cdot In(P) \cup \cdots$$

while the excluded set is given by

$$\Phi \cap Ex(P) \cap Ex(P \text{ and\_next } P) \cap Ex(P \text{ and\_next } P \text{ and\_next } P) \cap \cdots$$

The first quantity is still just $(In(P))^*$, while the second quantity still reduces to the null set. Hence, the expressions defining the

included and excluded sets for 'and_next* P', which represents the above infinite formula, remain the same:

$$In(\text{and\_next* } P) = (In(P))^*$$

$$Ex(\text{and\_next* } P) = \phi$$

For the special case when P is an atomic formula, Min(In(and_next* P)) is just the set of all 'linear' templates whose instances belong to In(P) and whose edges are labelled with 'next'.

6.6.6 Meaning of 'reverse'

Although a template is defined as having two types of edges, 'next' and 'last', the discussion so far has focused only on the first type. To see how 'last' edges enter the picture, consider what would happen if all the arrows in a trace were reversed. This is equivalent to reversing the past and future. If Instance x precedes (follows) Instance y in the original trace, then x follows (precedes) y in the reversed trace. Moreover, if x is the last instance of Process X preceding y in the original trace, then x is the next instance of Process X following y in the reversed trace. Similarly, if x is the next instance of Process X following y in the original trace, then x is the last instance of Process X preceding y in the reversed trace. As an illustration, consider the two templates in Figure 10. Each is the reverse of the other. Observe that a_0 precedes b_3 in the left trace but follows b_3 in the right trace. Note also that a_3 is the next instance of Process A following b_1 in the left trace but is the last instance of Process A preceding b_1 in the right trace.

These observations motivate the definitions for the 'reverse' connective, whose purpose is to reverse the 'polarity' of a formula P of MPL. This polarity reversal is accomplished by performing a simple transformation on each edge in each template of In(P) and Ex(P). If we take a 'next' edge from Instance x to Instance y to mean intuitively that y is a next instance following x, and if we take a 'last' edge from x to y to mean intuitively that x is a last instance preceding y, then it is clear from the above discussion that the appropriate transformation is:

$$x \text{ next } y \quad \text{--->} \quad y \text{ last } x$$

$$x \text{ last } y \quad \text{--->} \quad y \text{ next } x$$

These transformations are reflected in the following definitions for 'reverse P'.[17]

$$In(\text{reverse } Q) = \{\langle I, L^{-1}, N^{-1}\rangle \mid \langle I,N,L\rangle \text{ is in } In(Q)\}$$

$$Ex(\text{reverse } Q) = \{\langle I, L^{-1}, N^{-1}\rangle \mid \langle I,N,L\rangle \text{ is in } Ex(Q)\}$$

As an illustration of these definitions, suppose that the template depicted in Figure 5 on page 34 is in In(Q). Then the 'reversed'

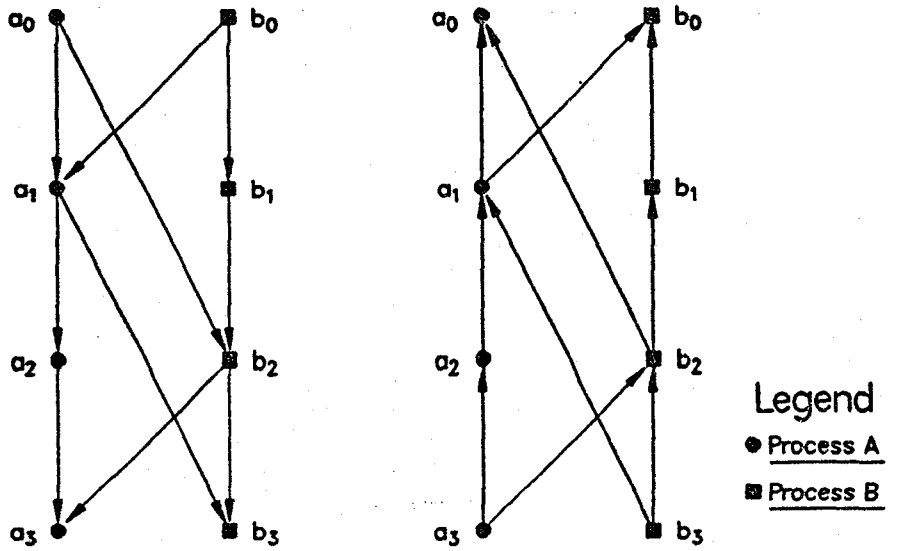


Figure 10. A Trace and its Reverse

template shown in Figure 11 is in In(reverse Q). To construct the template of Figure 5 in the first place, let $P_0 \cdots P_7$ be atomic formulas such that Process(x_i) = Process(P_i) and Value(x_i) is in Values(P_i) for $0 \le i \le 7$. The template is then contained in In(Q) where Q is the formula:

$$((P_0 \text{ and\_next } P_2) \text{ and } (P_1 \text{ and\_next } P_3)) \text{ and\_next}$$

$$(\text{reverse } ((P_6 \text{ and } P_7) \text{ and\_next } P_5 \text{ and\_next } P_4))$$

6.6.7 Meanings of auxiliary connectives

The connectives 'or', 'or_next', 'or_next*', 'implies' and 'implies_next' are all abbreviations for expressions involving the four basic connectives 'not', 'and', 'and_next' and 'and_next*'. The meanings of these additional connectives, therefore, follow directly from the preceding definitions. We list here those meanings and refer the reader to the uniprocess discussion for a further elaboration.

$$\text{In}(P \text{ or } Q) = \text{In}(P) \cup \text{In}(Q)$$

$$\text{Ex}(P \text{ or } Q) = \text{Ex}(P) \cap \text{Ex}(Q)$$

---

[17] R^{-1} denotes the converse of the binary relation R. $\langle x,y \rangle$ is in R^{-1} if and only if $\langle y,x \rangle$ is in R.

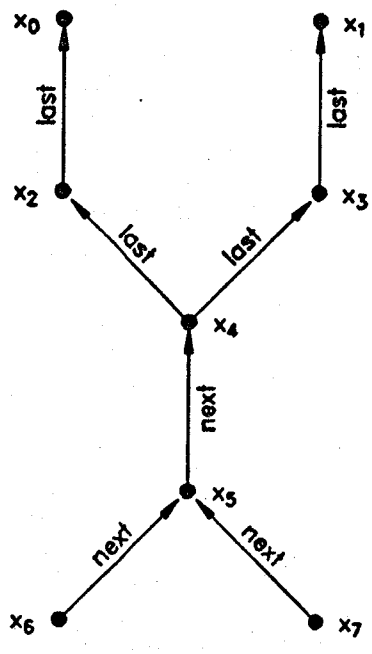


Figure 11. A Reversed Template

$$In(P \text{ or\_next } Q) = In(P) \cdot Bo(Q) \cup Bo(P) \cdot In(Q)$$

$$Ex(P \text{ or\_next } Q) = Ex(P) \cdot Ex(Q)$$

$$In(\text{or\_next* } P) = \phi$$

$$Ex(\text{or\_next* } P) = (Ex(P))^*$$

$$In(P \text{ implies } Q) = Ex(P) \cup In(Q)$$

$$Ex(P \text{ implies } Q) = In(P) \cap Ex(Q)$$

$$In(P \text{ implies\_next } Q) = Ex(P) \cdot Bo(Q) \cup Bo(P) \cdot In(Q)$$

$$Ex(P \text{ implies\_next } Q) = In(P) \cdot Ex(Q)$$

6.6.8 Satisfaction and Truth

In defining the semantics for UPL, we said that a sequence of values α 'satisfies' a formula P if α contains no member of Ex(P) as a

subsequence. A parallel notion is used for MPL. The definition here is stated not in terms of a value sequence containing a subsequence, but in terms of a trace having a template that 'fits' it.

A template T <u>fits</u> a trace Z if there exists a mapping Ψ from Instances(T) to Instances(Z) such that for all x,y in Instances(T),

- Process(x) = Process(Ψ(x))
- Value(x) = Value(Ψ(x))
- <x,y> in Next(T) => Ψ(y) is the next instance of Process(y) following Ψ(x) within Z
- <x,y> in Last(T) => Ψ(x) is the last instance of Process(x) preceding Ψ(y) within Z

Consider the template, trace and mapping Ψ (indicated by dashed lines) depicted in Figure 12. Assume that like-named instances in the template and trace have the same process and value. Thus, the process and value of a_2 in the template and the process and value of a_2 in the trace are the same. Now observe that for each 'next' edge <x,y> in the template, Ψ(y) is the next instance of Process(y) following Ψ(x) within the template. Specifically, a_3 is the next instance of Process A following a_2, and a_3 is the next instance of Process A following b_1. Notice also that for each 'last' edge <x,y> in the template, Ψ(x) is the last instance of Process(x) preceding Ψ(x) within the template. In particular, b_0 is the last instance of Process B preceding a_2. We conclude that the template fits the trace.

Using the notion of a template 'fitting' a trace, we define the concept of satisfaction. A trace Z <u>satisfies</u> a formula P of MPL if and only if there is no template in Ex(P) that fits Z. A formula P of MPL is <u>true</u> (with respect to a given system) if and only if every permitted trace satisfies P.

Justification for our practice of using the minimal elements of a set of templates to represent that set is provided by Property 6, which is a direct consequence of Property 5.

PROPERTY 5. Let T_1 and T_2 be templates and let Z be a trace. Then

$$T_1 \leq T_2 \text{ and } T_2 \text{ fits } Z \Rightarrow T_1 \text{ fits } Z$$

PROPERTY 6. A trace Z satisfies a formula P of MPL if and only if there is no template in Min(Ex(P)) that fits Z.

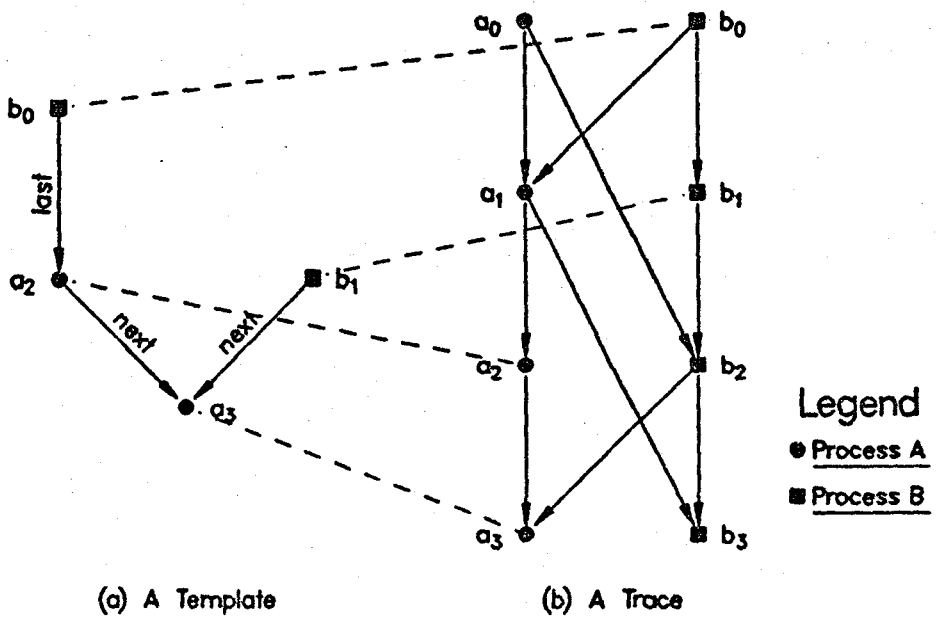


Figure 12. Fitting a Template to a Trace

6.7 Algebraic Properties

All the algebraic properties listed for UPL carry over to MPL. Thus, the operations of 'not', 'and' and 'or' form a DeMorgan algebra on the formulas of MPL (in the sense described above). Moreover, 'and_next' and 'or_next' are associative and satisfy DeMorgan's Laws.

6.8 Examples of Statements about Multiprocess Behavior

MPL is capable of expressing a broad range of complex logical and timing dependencies. An attempt to delineate the expressive power of MPL, however, is beyond the present scope. We limit ourselves instead to discussing how the examples given earlier for UPL translate to the multiprocess case.

6.8.1 Invariance

An invariant (in both the uniprocess and multiprocess case) is a true formula constructed from atomic formulas belonging to a single process using only the connectives 'and', 'or' and 'not'. For the uniprocess case, the meaning of an invariant Q (interpreted as a true formula) is straightforward: Every value in every permitted value

sequence must satisfy Q. For the multiprocess case, there is a slight variation.

In giving the meaning earlier for the connective 'and', we observed that when P and Q are both atomic formulas belonging to the same process, In(P and Q) contains certain templates that have no analog in the uniprocess case. These nonclassical templates do not affect the interpretation of 'P and Q' since it is Ex(P and Q) - not In(P and Q) - that determines which traces satisfy the formula. The situation is reversed, however, for the formula 'P or Q' because the nonclassical templates now appear in Ex(P or Q). Suppose, for example, that P and Q are atomic formulas belonging to Process A and that v_0, v_1, v_2 and v_3 are the values of A's type. Suppose, furthermore, that:[18]

$$Min(In(P)) = \{\{v_0\}, \{v_1\}\}$$

$$Min(Ex(P)) = \{\{v_2\}, \{v_3\}\}$$

$$Min(In(Q)) = \{\{v_0\}, \{v_2\}\}$$

$$Min(Ex(Q)) = \{\{v_1\}, \{v_3\}\}$$

It then follows that:

$$Min(In(P \text{ or } Q)) = \{\{v_0\}, \{v_1\}, \{v_2\}\}$$

$$Min(Ex(P \text{ or } Q)) = \{\{v_3\}, \{v_1, v_2\}\}$$

The single-instance template $\{v_3\}$ in Min(Ex(P or Q) has a counterpart in the uniprocess case, but the two-instance template $\{v_1, v_2\}$ has no such counterpart. Let us consider what each template says. The first says that Process A can never take on the value v_3 in a permitted trace. The second says, in effect, that Process A can never take on the values v_1 and v_2 in the same permitted trace.

A little reflection reveals the difference in interpreting the formula 'P or Q' for the uniprocess and multiprocess cases. For the uniprocess case, 'P or Q' means that 'P or Q' holds for all values of Process A in a permitted value sequence. For the multiprocess case, 'P or Q' means that either P holds for all values of Process A or Q holds for all values of Process A in a permitted trace. (This difference in interpreting the connective 'or' extends to the case when P and Q are non-atomic formulas.)

---

[18] Since only one process is being considered and since the templates under discussion contain no edges, we represent each template by the values associated with its instances.

6.8.2 Other Examples

Examples 2 through 8 in Section 5.6 carry over to the multiprocess case with little change. One difference applies to the formulas in Examples 2, 4, 5 and 8. Since we are no longer dealing with just a single process, the atomic formula T, which is universally true in the uniprocess case, must be replaced by an atomic formula that is associated with a particular process. Thus, in Example 2, the atomic formula T must be replaced by T_A where A is the process associated with either the atomic formula P or the atomic formula Q. T_A is satisfied by all the values in Type(A).

6.9 Extended MPL

Although the five basic connectives and five auxiliary connectives of MPL provide a concise and powerful set of primitives for specifying multiprocess behavior, it often awkward to express certain relationships in terms of these primitives alone. For this reason, we permit higher-level constructs to be introduced. Some suggested ones are the following: (Their translation into standard formulas of MPL should be apparent from the discussions in Sections 5.6 and 6.8.)

- P followed N time_units later_by Q.
- Q inevitable_within N time_units following P.
- Q for N time_units following P.
- Following P, Q.
- Following P, Q as_long_as R.
- Following P, Q until R.
- Following P, Q repeated_every N time_units.

6.10 Format for Logical Specifications

Each of the first three components of a system specification - type declarations, process declarations and synchronic structure - have a syntax that either is borrowed directly from the Ada programming language or is adapted from Ada. That convention is followed again for the logical specification. The format for a logical specification is:

```
specification SYSTEM_NAME is

   declaration1;
   declaration2;
   declaration3;

begin

   MPL statement1;
   MPL statement2;
   MPL statement3;

end;
```

Each declaration, which has the form of an Ada object declaration, provides a variable, universally quantified over a specified type, for use in the MPL statements.

7.0 SPECIFICATION EXAMPLE: THE ALTERNATING-BIT PROTOCOL

The alternating-bit protocol [3] [4] [15] [22] [23] provides a simple mechanism for achieving reliable communication over an unreliable channel. We consider the simplified case in which a 'source' accepts 'messages' for transmission to a 'destination'. To each accepted message, the source attaches a 'bit' and then transmits the resulting 'packet' repeatedly until an acknowledgement with the same bit is received. After such an acknowledgement, the source accepts a new message for transmission but this time the bit attached to the message is reversed (hence the name of the protocol).

Seven asynchronous, event processes are used to model the source, destination and initializer:

SOURCE_INPUT - input port for messages

TRANSMISSION_ACK - acknowledges transmission of message

SOURCE_SEND - transmitting port for packets

SOURCE_RECEIVE - receiving port for acknowledgement bits

DESTINATION_RECEIVE - receiving port for packets

DESTINATION_SEND - transmitting port for acknowledgement bits

DESTINATION_OUTPUT - output port for messages .

INITIALIZER - initializes source and destination

We present the formal specification of the alternating-bit protocol in terms of the constraints imposed on these seven processes. The next four sections contain, respectively: type definitions, process declarations, synchronic structure and logical specification.

7.1 Type Definitions

```
type MESSAGE_TYPE is INTEGER;

type PACKET_TYPE is
   record
      MESSAGE: MESSAGE_TYPE;
      BIT: BOOLEAN;
   end record;

type ACKNOWLEDGE_TYPE is BOOLEAN;

type INITIALIZE_TYPE is (INIT);
```

7.2 Process Declarations

```
SOURCE_INPUT is event process of MESSAGE_TYPE; [19]

TRANSMISSION_ACK is event process of ACKNOWLEDGE_TYPE;

SOURCE_SEND is event process of PACKET_TYPE;

SOURCE_RECEIVE is event process of ACKNOWLEDGE_TYPE;

DESTINATION_RECEIVE is event process of PACKET_TYPE;

DESTINATION_SEND is event process of ACKNOWLEDGE_TYPE;

DESTINATION_OUTPUT is event process of MESSAGE_TYPE;

INITIALIZER is event process of INITIALIZE_TYPE;
```

[19] In addition to the values in its type, we assume that each event process may also take on the value NULL. NULL is distinct from all other values in a type and is used to indicate the absence of all (non-null) values.

7.3 Synchronic Structure

```
SOURCE_INPUT is asynchronous;

TRANSMISSION_ACK is asynchronous;

SOURCE_SEND is asynchronous;

SOURCE_RECEIVE is asynchronous;

DESTINATION_RECEIVE is asynchronous;

DESTINATION_SEND is asynchronous;

DESTINATION_OUTPUT is asynchronous;

INITIALIZER is asynchronous;
```

7.4 Logical Specification

```
specification ALTERNATING_BIT_PROTOCOL is

   MSG: MESSAGE_TYPE;
   BIT: ACKNOWLEDGEMENT_TYPE;

begin


      Following
   SOURCE_INPUT/=NULL,
   SOURCE_INPUT=NULL
      until
   TRANSMISSION_ACK/=NULL;

-- A new message is not given to the source until
-- the preceding message is acknowledged.[20]
```

[20] A double dash (--) is the Ada convention for a comment.

```
   . Following
   INITIALIZER=INIT,
   SOURCE_SEND=NULL
      until
   SOURCE_INPUT/=NULL;

-- Following initialization, no packets are
-- sent before the first message is sent.


   INITIALIZER=INIT
      and_next
   (and_next* SOURCE_INPUT=NULL)
      and_next
   SOURCE_INPUT=MSG
      implies_next
   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=0;

-- The bit of the first packet sent following
-- initialization is 0.


      Following
   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=BIT,
   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=BIT
      until
   SOURCE_RECEIVE=BIT;

-- A packet is sent repeatedly until appropriate
-- acknowledgement is received.


   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=BIT
      and_next
   SOURCE_RECEIVE=BIT,
      implies_next
   TRANSMISSION_ACK=1;

-- When a packet is acknowledged, the transmission of
-- the message is acknowledged.


      Following
   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=BIT
      and_next
   SOURCE_RECEIVE=BIT,
   SOURCE_SEND=NULL
      as_long_as
   SOURCE_INPUT=NULL;

-- Once a packet is acknowledged, no new packets
-- are sent before a new message is provided.
```

```
   SOURCE_SEND.BIT=BIT
      and_next
   (and_next* SOURCE_SEND=NULL)
      and_next
   SOURCE_INPUT=MSG
      implies_next
   SOURCE_SEND.MESSAGE=MSG and SOURCE_SEND.BIT=(not BIT);

-- The bit for a new packet is alternated.


      Following
   INITIALIZER=INIT,
   DESTINATION_OUTPUT=NULL and DESTINATION_SEND=NULL
      as_long_as
   DESTINATION_RECEIVE=NULL;

-- No messages are received and no acknowledgments
-- are sent before the first packet following
-- initialization is received.


   INITIALIZER=INIT
      and_next
   (and_next* DESTINATION_RECEIVE=NULL)
      and_next
   DESTINATION_RECEIVE.MESSAGE=MSG and DESTINATION_RECEIVE.BIT=0
      implies_next
   DESTINATION_OUTPUT=MSG and DESTINATION_SEND=0;

-- If the first packet following initialization
-- has a 0 bit, then the message is received
-- and 0 is acknowledged.


      Following
   DESTINATION_SEND=BIT,
   DESTINATION_SEND=BIT
      until
   DESTINATION_RECEIVE.BIT=(not BIT);

-- The same acknowledgment is sent repeatedly until a packet
-- with an alternated bit is received.
```

```
   DESTINATION_RECEIVE.BIT=BIT
      and_next
   (and_next* DESTINATION_RECEIVE=NULL)
      and_next
   DESTINATION_RECEIVE.MESSAGE=MSG and DESTINATION_RECEIVE.BIT=(not BIT)
      implies_next
   DESTINATION_OUTPUT=MSG and DESTINATION_SEND=(not BIT);

-- When a change occurs in the bit of a received packet,
-- the message is supplied to the output and the new
-- bit is acknowledged.


      Following
   DESTINATION_RECEIVE.MESSAGE=MSG and DESTINATION_RECEIVE.BIT=BIT
      and_next
   DESTINATION_OUTPUT=MSG,
   DESTINATION_OUTPUT=NULL
      as_long_as
   DESTINATION_RECEIVE.BIT/=(not BIT);

-- A message is supplied to the destination output only
-- when there is a bit change on the received packet.


end;
```

8.0 CONCLUSIONS

We have described a rigorous framework for specifying the behavior of concurrent systems. Among its features are:

- Generality
- Expressiveness
- Naturalness

The generality stems from a model of system behavior that introduces a minimal set of primitive concepts - just values and processes - and makes a minimal assumption - the behavior of a process is represented by a linear sequence of values. The expressiveness reflects the approach adopted to specify the permitted traces of a system. Through the synchronic structure and logical specification, it is possible to express an extremely broad range of logical and timing dependencies. Many of these dependencies are simply not expressible with any other existing technique. The naturalness (or readability) results from the

absence of arcane notation and obscure terminology. There is very little new notation and the only new terminology consists of the connectives 'and_next', 'and_next*', 'or_next', 'or_next*', 'implies_next' and 'reverse'. The technical meanings of these connectives closely parallels their informal, intuitive meanings. The readability of a specification is enhanced when MPL is extended to include such higher-level constructs as "following", "until" and "as long as".

8.1 Future Work

Further development of the specification framework needs to proceed along several lines:

- Improvements and Extensions
- Verification Capabilities
- Hierarchical Specification
- Larger Methodologies

The need to improve and extend the framework will inevitably arise as applications experience is gained. One area in need of improvement that has already been identified is the synchronic structure, which is presently limited in the sorts of synchronic relationships it can express.

The desire to rigorously verify system behavior has provided much of the impetus for the present effort, and without a deductive capability the present framework remains incomplete. Such a capability has already been provided for a precursor to the present theory [11] [12] [13], and it is possible that some of principles underlying this earlier effort may generalize to the present case.

Composing (or decomposing) a specification in a hierarchical fashion is the most effective way of dealing with complexity. Appropriate mechanisms for 'connecting' different levels of a hierarchical specification need to be developed.

Although formal specifications of intended behavior and actual behavior are important elements in the design of a system, to be used effectively, they must be integrated with other elements in the system development process. The ultimate goal is a unified methodology encompassing:

- Specification of Mission Requirements
- Specification of Functional Requirements

- Specification of Behavioral Requirements
- Design
- Verification
- Testing
- Configuration Control
- Maintenance

INDEX

LIST OF REFERENCES

1. Balbes, R. and Dwinger, P., Distributive Lattices, University of Missouri Press, 1974, Chapter XI.

2. Barnes, J.G.P., Programming in Ada, Addison-Wesley, 1981.

3. Bartlett, K.A., Scantlebury, R.A. and Wilkinson, P.T., "A Note on Reliable Full-Duplex Transmission over Half-Duplex Links", Communications of the ACM, Vol. 12, No. 5, May 1969, pp. 260-261.

4. Bochmann, G.V. and Gecsei, J., "A Unified Method for the Specification and Verification of Protocols", Proceedings IFIP Congress, Toronto, Canada, August 1977, pp. 229-234.

5. Booch, G., Software Engineering with Ada, Benjamin/Cummings, 1983.

6. Campbell, R.H. and Haberman, A.N., "The Specification of Process Synchronization by Path Expressions", Lecture Notes in Computer Science 16., 1974, pp. 89-102.

7. Chen, B., Event-Based Specification and Verification of Distributed Systems, University of Maryland, (PhD thesis), 1982.

8. Dijkstra, E.W., "Co-Operating Sequential Processes", Structured Programming, Academic Press, 1968, pp. 43-112.

9. Fitzwater, D.R. and Zave, P., "The Use of Formal Asynchronous Process Specifications in a System Development Process", 6th Texas Conference on Computer Systems, November 1977, pp. 2B/21-2B/30.

10. Furtek, F.C., The Logic of Systems, Massachusetts Institute of Technology Laboratory for Computer Science Technical Report MIT/LCS/TR-170, December 1976.

11. Furtek, F.C., "The Theory of Constraints", submitted for publication.

12. Furtek, F.C., "A Necessary and Sufficient Condition for a Product Relation to be Total", Journal of Combinatorial Theory - Series A, November 1984, (to appear).

13. Furtek, F.C., "Constraint Logic", 1983 Conference on Information Sciences and Systems, The Johns Hopkins University, March 1983, pp. 491-494.

14. Greif, I., "A Language for Formal Problem Specification", Communications of the ACM, Vol. 20, No. 12, December 1977, pp. 931-935.

15. Hailpern, B. and Owicki, S., "Verifying Network Protocols Using Temporal Logic", Proceedings 1980 Trends and Applications Symposium

on Computer Network Protocols, National Bureau of Standards, Gaithersburg, MD, May 1980, pp. 18-28.

16. Hoare, C.A.R., "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10, October 1974, pp. 549-557.

17. Hoare, C.A.R., "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

18. Holt, A.W. et al., Final Report of the Information System Theory Project, Technical Report No. RADC-TR-68-305, Rome Air Development Center, Griffis Air Force Base, New York, September 1968.

19. Hopcroft, J.E. and Ullman, J.D., Formal Languages and their Relation to Automata, Addison-Wesley, 1969.

20. Lynch, N.A. and Fischer, M.J., "On Describing the Behavior and Implementation of Distributed Systems", Theoretical Computer Science, Vol. 13, 1981, pp. 147-171.

21. Misra, J. and Chandy, K.M., "Proofs of Networks of Processes", IEEE Transactions on Software Engineering, Vol. SE-7, No. 4, July 1981, pp. 147-171.

22. Schwartz, R.L. and Melliar-Smith, P.M., "Temporal Logic Specification of Distributed Systems", The 2nd International Conference on Distributed Computing Systems, Paris, April 1981, pp. 446-454.

23. Sunshine, C.A. et al., "Specification and Verification of Communication Protocols in AFFIRM Using State Transition Models", IEEE Transactions on Software Engineering, Vol. SE-8, No. 5, September 1982, pp. 460-489.

24. United States Department of Defense, Reference Manual for the Ada Programming Language, ANSI/MIL STD-1815A, 22 January 1983.

25. Zave, P., "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 250-269.

END

DATE

FILMED

OCT 27 1984

LANGLEY RESEARCH CENTER
3 1176 01324 4646