NASA CONTRACTOR REPORT 181804

SPECIFYING REAL-TIME SYSTEMS WITH INTERVAL LOGIC

Volume 1 of 2

John Rushby, Principal Investigator
Computer Science Laboratory

NASA Contract NAS1-17067, Task 4
Dated: November 30, 1988

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Contents

SUMMARY

The research described in this report is presented in 2 parts.

Part I: *Interval Logic for the Verification of Real Time Systems* presents a technique for the formal expression of the real-time constraints that are critical to the specification of fault-tolerant distributed systems.

Part II: *Extending Interval Logic to Include Processes* describes how the specification language of EHDM can be extended to include processes and interval logic specifications.

Part I

Part I: Interval Logic for the Verification of Real Time Systems

Section 1

The Use of Interval Logic for the Verification of Real Time Systems

Temporal logic has been found useful for specifying distributed asynchronous systems. Traditionally, such specifications have been expressed as interacting machines, but that approach inevitably suffers from over specification, for the state machines represent an implementation. If the application is such that only one implementation is envisaged, an implementation oriented specification may be acceptable; but other applications, for example, communications protocol specifications, envisage many distinct implementations. By specifying the minimum required externally visible behavior, leaving all other aspects to lower levels of description, one can obtain a more general specification that reflects the necessary requirements of the distributed system or protocol. A specification that is oriented towards one implementation may discourage or even preclude other equally valid implementations. Specifications expressed in temporal logic do not suffer as severely from implementation bias as do state machine specifications.

A specification for a distributed system can serve to define the externally observable function of the system, in effect the service provided by

the system. Such specifications are called service specifications. A service specification regards the entire distributed system as a single entity, with multiple interfaces at separate nodes of the distributed system. The specification defines how operations at each interface, performed asynchronously, affect results at other interfaces. Ideally, a service specification defines only the behavior visible at the external interfaces, without suggesting any internal structure for the system.

Many service specifications define that all operations at the external interfaces be serializable, a characteristic that is often desirable for user interfaces. Such specifications can often be expressed with simpler specification languages that provide only the concepts of parallel operation and of atomicity.

Alternatively, a specification can define the manner in which the separate components of the distributed system interact with each other so as to provide the required function. Such a specification is called an implementation specification or a protocol specification. An implementation specification defines separately the behavior of each component, so that each distributed component can be implemented separately. The specifications describe how the components communicate with each other using a communication facility, which is defined by a service specification, as is shown in Figure 1.1. The communication facility is, of course, itself a distributed system for which there is, in addition to the service specification, also an implementation specification dependent on an even more primitive communication mechanism. In many distributed systems, the hierarchy of such specifications is several levels deep.

If there are to be several independent implementations of some of the components, in the future even if not immediately, it is important that the implementation specification describe only how the components interact with each other without unnecessarily constraining the internal implementation of any component. The ideal specification is one in which

- any component, that satisfies its specification, will operate satisfactorily in the system, and
- any component, that operates satisfactorily in the system, will satisfy the specification.

If both a service specification and an implementation specification have been constructed for a distributed system, it is possible to validate the implementation specification by confirming that it satisfies the service specification. This ability is very valuable for the implementation specification is often quite complex and prone to error, while the service specification is much shorter and simpler. Unfortunately, the current state of the art, and particularly of tools, has not yet advanced to the point at which such a validation is feasible for typical distributed systems.

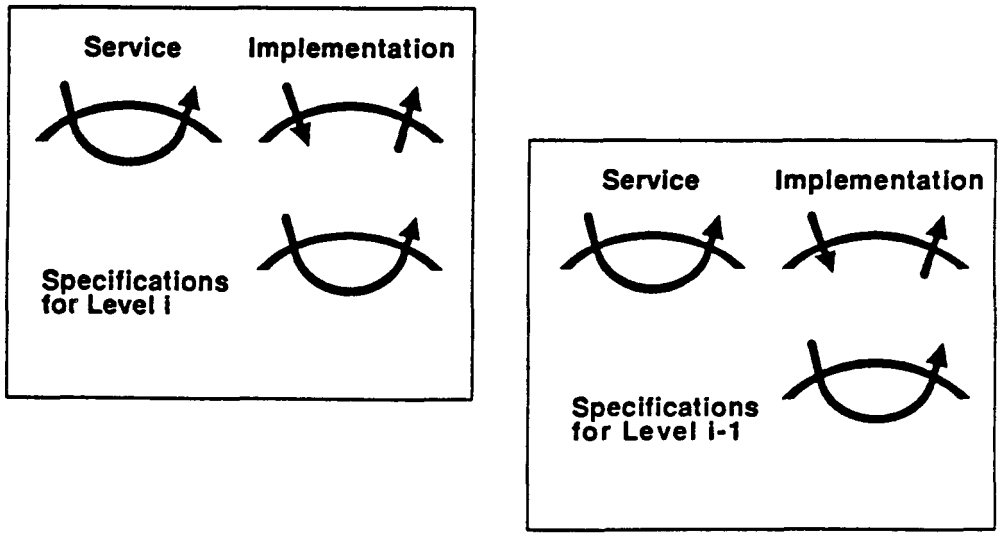


Fig. 1 Specification of a Level in the Protocol Hierarchy

Section 2

The Basic Interval Logic

In a previous survey paper[1], we examined how several different temporal logic approaches express the conceptual requirements for a simple protocol. Our conclusions were both disappointing and encouraging. On one hand, we saw how the very abstract temporal requirements provided an elegant statement of the minimal behavior for an implementation to conform to the specification. We were able to distill a set of requirements expressing the essence of the desired behavior, stating only requirements without implementation-constraining expedients.

While we were happy with the level of conceptualization of the specifications, their expression in temporal logic was rather complex and difficult to understand. The relatively low level of the linear-time temporal logic operators encourages the inclusion of additional state components that are not properly part of the specification, but that help to establish the context necessary to express the requirements. Without these components, context can only be achieved by complex nestings of temporal "until" constructs to establish a sequence of prior states. The survey paper showed how the introduction of state simplifies the temporal logic formulas at the expense of increasing the amount of "mechanism" in the specification. The specification that defined only the minimum required externally visible behavior, without any additional internal state components, was also the least read-

able. As a result of this survey, the interval logic was developed to allow the specification of distributed systems in a manner that corresponds more closely to the intuitive intent and understanding of the designers.

At the heart of our interval logic are formulas of the form:

$$[\, I \,]\alpha$$

Informally, the meaning of this is: "The *next* time the interval I can be constructed, the formula α will 'hold' for that interval." This interval formula is evaluated within the current interval context and is vacuously satisfied if the interval I cannot be found. A formula 'holds' for an interval if it is satisfied by the interval sequence, with the present state being the beginning of the interval.

The unary $\Box$ and $\Diamond$ temporal logic operators retain their intuitive meaning within interval logic. The formula $[\, I \,]\Box\alpha$ requires that property α must hold throughout the interval, while $[\, I \,]\Diamond\alpha$ expresses the property that sometime during the interval I, α must hold. For simple state predicate P, the interval formula $[\, I \,]P$ expresses the requirement that P be true in the first state of the interval.

Interval formulas compose with the other temporal operators to derive higher-level properties of intervals. The formula

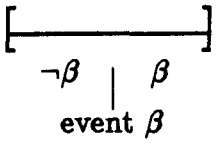
$$[\, I \,][\, J \,]\alpha$$

states that the first J interval contained in the next I interval, if found, will have property α. The property that all J intervals within interval I have property α would be expressed as $[\, I \,]\Box[\, J \,]\alpha$. More globally, the formula $\Box[\, I \,]\alpha$ requires all further I intervals to have property α.

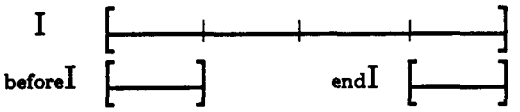
Each interval formula $[\, I \,]\alpha$ constrains α to hold only if the interval I can be found. Thus only when the context can be established need the interval property hold. To *require* that the interval occur, one could write $\neg[\, I \,]$False. The interval language defines the formula $*\, I$ to mean exactly this.

Thus far, we have described how to compose properties of intervals without discussing how intervals are formed. At the heart of a very general mechanism for defining and combining intervals is the notion of an *event.* An event, defined by an interval formula β, occurs when β changes from False to True, i.e., when it *becomes* true. In the simplest case, β is a predicate on the state, such as $x > 5$ or at Dq . Note that, if the predicate is true in the initial state, the event occurs when it changes from False to True, and thus only after the predicate has become False.

Intervals are defined by a simple or composed interval term. The primitive interval, from which all intervals are derived, is the *event interval.* An event, defined by β, denotes the *interval of change* of length 2 containing the $\neg\beta$ and β states comprising the change. Pictorially, this is represented as



Two functions, before and end, operate on intervals to extract unit intervals. For interval term I, before$I$ denotes the unit interval containing the first state of interval I. Similarly, end$I$ denotes the unit interval at the end. Application of the end function is undefined for infinite intervals. Again, pictorially, the intervals selected are



For a P predicate event, the following formulas are valid.

$[\,\text{end}\,P\,]P$

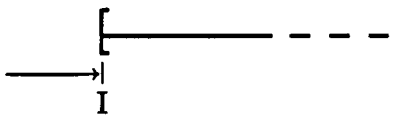
$[\,\text{before}\,P\,]\neg P$

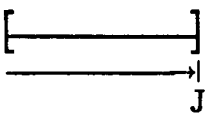
$[\,P\,]\neg P$

2.1 The Interval Operators$\Longrightarrow$ and $\Longleftarrow$

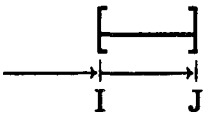
Two generic operators exist to derive intervals from interval arguments. We take the liberty of overloading these operators to allow zero, one or two interval-value arguments. Intuitively, the direction of the operator indicates in which direction and in which order the interval endpoints are located. The endpoint at the tail of the arrow is first located, followed by a search in the direction of the arrow for the second endpoint. A missing parameter causes the related endpoint to be that of the outer context.

The interval term $I \Rightarrow$ denotes the interval commencing at the end of the next interval I and extending for the remainder of the outer context. The right arrow operator, in effect, locates the *first* I interval, relative to the outer context, and forms the interval from the *end* of that I interval onward. With only a second argument present, $\Rightarrow J$ denotes the interval commencing with the first state of the outer context and extending to the *end* of the *first* J interval. Thus,



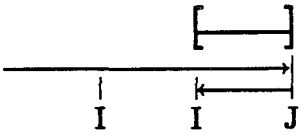


The term $I \Rightarrow J$, with two interval arguments, represents the composition of the two definitions. This constructs the interval starting at the end of interval I and extending to the end of the *next* interval J located in the interval $I \Rightarrow$. Given this definition, the interval formula $[\ I \Rightarrow J\ ]\alpha$ is equivalent to $[\ I \Rightarrow\ ][\ \Rightarrow J\ ]\alpha$. Recall that the formula $[\ I \Rightarrow J\ ]\alpha$ is vacuously true if the $I \Rightarrow J$ interval cannot be found. Pictorially, the interval selected is



The right arrow operator with no interval arguments selects the entire outer context.

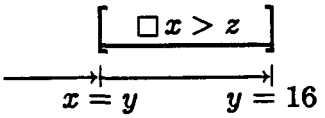
The left arrow operator $\Leftarrow$ is defined analogously. For interval term $I \Leftarrow J$, the first J interval in context is located. From the end of this J interval, the *most recent* I interval is located. The derived interval $I \Leftarrow J$ begins with ${}_{\text{end}}I$ and ends with ${}_{\text{end}}J$. Thus,



Similarly, the interval term $I \Leftarrow$ selects the interval beginning with the end of the last I interval and extending for the remainder of the context. For a context in which an interval I occurs an infinite number of times, the formula $[\, I \Leftarrow \,]\, \alpha$ is vacuously true. The interval terms $\Leftarrow$ and $\Leftarrow J$ are strictly equivalent to $\Rightarrow$ and $\Rightarrow J$, respectively.

The following examples illustrate the use of the interval operators.

$$[\, x = y \;\Rightarrow\; y = 16 \,]\, \Box\; x > z \tag{2.1}$$



For the interval beginning with the next event of the variable x becoming equal to y and ending with y changing to the value 16, the value of x is asserted to remain greater than z. The first state of the interval is thus

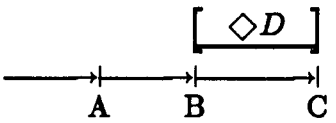
the state in which x is equal to y and the last state is that in which y is next equal to 16. Note that the events $x = y$ and $y = 16$ denote the next *changes* from $x \neq y$ and $y \neq 16$.

To modify the above requirement to allow $x > z$ to become False as y becomes 16, one could write

$$[\, x = y \;\Rightarrow\; \text{before}(y = 16)\,]\,\Box\; x > z \tag{2.2}$$

Nesting interval terms provides a method of expressing more comprehensive context requirements. Consider the formula

$$[\, (A \Rightarrow B) \Rightarrow C\,]\Diamond D \tag{2.3}$$



The formula requires that, if an A event is found, the subsequent B to C interval, if found, must sometime satisfy property D. The outer $\Rightarrow$ operator selects the interval commencing at the end of its first argument, in this case, at the end of the selected $A \Rightarrow B$ interval. The interval then extends until the next C event – establishing the necessary context.

In the previous example, the formula was vacuously true if any of the events A,B, or C could not be found in the established context. In order to easily express a requirement that a particular event or interval *must* be found if the necessary context is established, we introduce an interval term modifier $*$. For interval term I, $*I$ adds an additional requirement that B must be found in the designated context. The formula

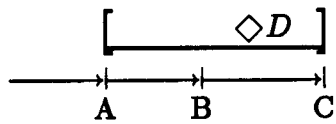
$$[\, (A \Rightarrow\; *B) \Rightarrow C\,]\Diamond D \tag{2.4}$$

strengthens formula (3) by adding the requirement that, if an A event occurs, a subsequent B event *must* occur. This is equivalent to formula (3) conjoined with $[\, A \Rightarrow\,]*B$.

The $*$ modifier can be applied to an arbitrary interval term. The formula $[\ *(A \Rightarrow B) \Rightarrow C\ ]\Diamond D$, for example, would be equivalent to (3) conjoined with $*(A \Rightarrow B)$, or equivalently, $*A \ \wedge\ [\ A \Rightarrow\ ]\ *B$. The $*$ modifier adds only linguistic expressive power and can be eliminated by a simple reduction (given in the Appendix).

As an example of specifying context for the end of the interval, consider the formula

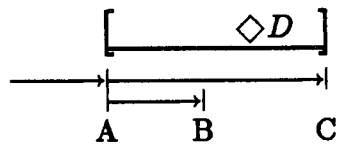
$$[\ A \Rightarrow (B \Rightarrow C)\ ]\Diamond D \tag{2.5}$$



Here, the interval begins with the next occurrence of A and terminates with the first C that follows the next B.

By modifying formula (3) to begin the interval at the beginning of $A \Rightarrow B$, i.e.,

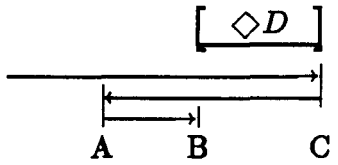
$$[\ \text{before}(A \Rightarrow B) \Rightarrow C\ ]\Diamond D \tag{2.6}$$



we obtain a requirement similar to that of (5), but allowing events B and C to be *arbitrarily ordered.*

Introducing the use of backward context, to find the interval $A \Rightarrow B$ in the context of C, we have

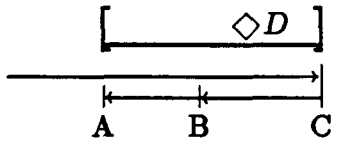
$$[\ (A \Rightarrow B) \Leftarrow C\ ]\Diamond D \tag{2.7}$$



Here the occurrence of the first C event places an endpoint on the context, within which the most recent $A \Rightarrow B$ interval is found. Note the order of search: looking forward, the next C is found, then backward for the most recent A, then forward for the next B. Thus, the formula is vacuously true if no B is found between C and the most recent A.

As a last example, consider

$$[\,\textbf{before}(A \Leftarrow B) \Leftarrow C\,]\Diamond D \tag{2.8}$$



The interval extends back from the first C event to the beginning of the most recent $A \Leftarrow B$ interval.

2.2 Parameterized Operations

Within the language of our interval logic we include the concept of an *abstract operation.* For an abstract operation O, state predicates $\textbf{at}O$, $\textbf{in}O$, and $\textbf{after}O$ are defined. These predicates carry the intuitive meanings of being "at the beginning", "within", and "immediately after" the operation. Formally, we use the following temporal axiomatization of these state predicates.

1. $[\,\textbf{at}O \Rightarrow \textbf{before}\ \textbf{after}O\,]\,\Box\textbf{in}O$

2. $[\ \textbf{after}O \Rightarrow \textbf{before at}O\ ]\,\Box\neg\textbf{in}O$

3. $[\ \neg\textbf{at}O \Rightarrow \textbf{after}O\ ]\,\Box\neg\textbf{at}O$

4. $[\ \neg\textbf{after}O \Rightarrow \textbf{at}O\ ]\,\Box\neg\textbf{after}O$

Axioms 1 and 2 together define $\textbf{in}O$ to be true exactly from $\textbf{at}O$ to the state immediately preceding $\textbf{after}O$. Axiom 3 allows $\textbf{at}O$ to be true only at the beginning of the operation, and axiom 4 requires that $\textbf{after}O$ be true only immediately following an operation. Note that, in axiom 1 for example, the predicate $\textbf{at}O$ used as an event term defines the interval commencing with the *entry* to the operation.

The axioms do not imply any specific granularity, duration or mapping of the operation symbol to an implementation. ***Any interpretation of these state predicate symbols satisfying the above axioms is allowed.*** In addition, no assumption of operation termination is made. To require an operation to always terminate, one could state as an axiom

$$[\ \textbf{at}O \Rightarrow\ *\ \textbf{after}O\ ]\text{True}$$

Abstract operations may take entry and result parameters. For an operation taking n entry parameters of types $T_1, \ldots, T_n$, and m result parameters of types $T_{n+1}, \ldots, T_{n+m}$, the **at** and **after** state predicates are overloaded to include parameter values. $\textbf{at}O(v_1, \ldots, v_n)$ is true in any state in which $\textbf{at}O$ is true and the values of the parameters are $v_1, \ldots, v_n$. The predicate **after** is similarly overloaded.

As an example of an interval requirement involving parameterized operations, consider an operation O with a single entry parameter. To require that this parameter increase monotonically over the call history, one could state

$$\forall a, b\ \Box[\ \textbf{at}O(a) \Rightarrow \textbf{at}O(b)\ ]\ b > a$$

Since a and b are free variables, for all a and b such that we can find an

interval commencing with an $_{\mathrm{at}}O(a)$ and ending with an $_{\mathrm{at}}O(b)$, b must be greater than a. Recall that the formula is vacuously true for any choice of a and b such that the interval cannot be found.

It is also useful to be able to designate the *next* occurrence of the operation call, and to bind the parameter values of that call. The event term $_{\mathrm{at}}O : (a)$ designates the next event $_{\mathrm{at}}O$ and binds the free variable a to the value of the parameter for that call. Thus the previous requirement constraining all pairs of calls, can be restated in terms of successive calls as

$$\Box[\ {}_{\mathrm{at}}O(a) \Rightarrow {}_{\mathrm{at}}O : (b)\ ]\ b > a$$

The requirement is now that for every a, the call $_{\mathrm{at}}O(a)$ is followed by a call of O whose parameter is greater than a. This parameter binding convention has a general reduction, which we omit here. For this specific formula, the reduction gives

$$\Box[\ {}_{\mathrm{at}}O(a) \Rightarrow\ ]\ \ (\ [\mathrm{end}\ {}_{\mathrm{at}}O\ ]{}_{\mathrm{at}}O(b)\ ) \supset [\ \Rightarrow {}_{\mathrm{at}}O\ ]b > a$$

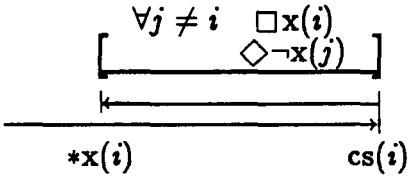
2.3 Some Example Specifications

Consider a queue with two operations, Enq which takes a single parameter value, which it enqueues, and Dq which removes the value at the front of the queue and returns that value as its result. We assume in this specification that the queue is unbounded, and require that values enqueued must be distinct. No assumptions are made about the atomicity of, or temporal relationships between, the Enq and Dq operations. These operations can overlap in an arbitrary manner. We do assume that at most one instance of the Enq and Dq operations will be active at any given time.

The specification expresses the fundamental first-in first-out behavior that characterizes a queue. It requires that, for all a and b, if we dequeue b, then any other value a will be dequeued in the interim if and only if it was enqueued prior to b. Further axioms are needed to express liveness requirements on the two operations.

Queue. $[\Leftarrow_{\text{after}}\text{Dq}(b)](*_{\text{after}}\text{Dq}(a) \equiv *(_{\text{at}}\text{Enq}(a) \Leftarrow _{\text{at}}\text{Enq}(b)))$

As a second example, consider a specification to ensure exclusive access to a shared critical section by some set of processes. Each process is to make an independent decision based on a shared global data structure. In stating the specification, we assume a state predicate cs(i) which, for process i, indicates that i is in the critical section. For a shared global data structure, we assume a state predicate x(i) which, for process i, indicates i's intention to enter the critical section. We wish to state minimal requirements on the use of state predicate x by a process to ensure mutual exclusion. Pictorially we represent the required behavior as follows:



As shown, an entry of the critical section by process i must be preceded by an earlier setting of x(i) to true. Throughout this interval x(i) must remain true, and, for every other process j, there must be some moment within the interval at which x(j) is false. This specification imposes no requirement on the order or frequency of inspecting the x(j)s; it suffices that, *at some time* during the interval, each x(j) is false. Herein lies the basic reason for exclusion. x(i) remains true through the interval, and no other x(j) can be true for that interval. Thus no other process j can find x(i) false between the time that i signals his intention and the time that i leaves the critical section (or abandons his claim). The specification does not, however, ensure the absence of deadlock.

In interval logic, we express these requirements as follows.

Init. $\forall m\ \neg \mathrm{x}(m)$

A1. $i \neq j \ \supset\ [\,\mathrm{x}(i) \Leftarrow \mathrm{cs}(i)\,]\Diamond\neg \mathrm{x}(j)$

A2. $\mathrm{cs}(i) \ \supset\ \mathrm{x}(i)$

Given an initial condition in which all processes have relinquished their claims, axiom A1 expresses our previous pictorial requirement that, if process i enters the critical section, then for the interval back to the most recent setting of $\mathrm{x}(i)$, each $\mathrm{x}(j)$ must be found to be false. Axiom A2 requires that $\mathrm{x}(i)$ remains true while i is in the critical section. We have not needed to state explicitly that there must be a setting of $\mathrm{x}(i)$ prior to the entry; this is deducible from the specification. Similarly we can deduce that $\mathrm{x}(i)$ remains true through that interval.

From this specification, we can demonstrate (omitted here) the mutual exclusion property that henceforth no pair of processes can both be in the critical section at the same time, i.e.,

$$\forall m\ \neg \mathrm{x}(m) \wedge i \neq j \ \supset\ \Box\neg(\,\mathrm{cs}(i) \wedge \mathrm{cs}(j)\,)$$

Section 3

Extending Interval Logic to Real Time Systems

Temporal logic has suffered from its orientation towards eventuality rather than immediacy in real time; indeed, pure temporal logic makes no reference to time! A temporal logic specification defines only invariants, eventuality, and order constraints on the sequence of states resulting from the execution of the distributed system without reference to when the states actually occur. But the specification of distributed systems typically depends critically on the specification of real time properties.

Surprisingly, in view of the orientation of temporal logic towards eventuality, there are useful eventuality properties, superficially independent of real time, that cannot be written without reference to real time. As an elementary example, the service specification for an elevator, without consideration of the possibility of elevator failure, can be expressed as a requirement that if a request is made for floor a then, eventually, the elevator will be at floor a with the door open.

$$\Box\ (\ \text{Request}(a) \supset \Diamond\ \text{atfloor}(a) \wedge \text{dooropen}(a)\ )$$

Unfortunately, any practical elevator inevitably has occasions when it is out of service, as expressed as

$\Box\Diamond\neg$inservice.

If we are to avoid expedients such as regarding an out of service state as a terminal state, or of requiring that the elevator remember the request for floor a through the out of service state (an unreasonable requirement), we would like to modify the service specification to state that the elevator will eventually be at floor a unless it goes out of service first. There is no way to express that requirement; the best that can be achieved is

$$\Box \left(\text{ request}(a) \supset \Diamond \left((\text{atfloor}(a) \wedge \text{dooropen}(a)) \vee \neg\text{inservice} \right) \right)$$

Careful examination shows that this specification is completely satisfied by the eventual out of service condition and it thus contributes nothing to the requirement that a request be serviced by moving to the requested floor. In effect, the elevator can satisfy the specification doing nothing but wait until it breaks.

To overcome this problem, we must place a real time bound on the period of time throughout which the elevator must be operational to guarantee that the service will be provided. The service specification then becomes:

$$\Box \left[\text{ request}(a) + \text{max\_service\_time} \right]$$

$$\Box \text{ inservice} \supset \Diamond (\text{atfloor}(a) \vee \text{dooropen}(a))$$

This states that for an interval commencing with the request and of length max_service_time, if the elevator is never out of service during the interval, then the service will be provided within that interval.

Thus we need to extend interval logic to include real time constraints, but we do not want, in so doing, to destroy what is valuable about the logic. Temporal logics are valuable because they allow the expression of necessary properties while precluding other forms of expression that would be inappropriate. For example, if time is represented explicitly as a numeric variable in our specifications, it is possible to express any useful temporal property, including those involving real time constraints. But, the explicit representation of time makes possible expressions that have no meaning, such as those in which a property depends on whether the time is even or

permit the construction of undesired expressions in which time is manipulated inappropriately.

The decidability of interval logic is unaffected by these extensions. It is not appropriate to digress here into a lengthy analysis of decidability, but rather we give only a brief outline of the necessary extensions to the decision process. A decision procedure for interval logic can be constructed as a standard semantic tableau, building a graph of possible states. The transitions between states are determined by the order of events, and thus the predicates on the states comprise the conjunction of the normal state predicates with a set of relations on the order of events.

To extend this semantic tableau decision process to the real time version of interval logic, the real time relational operators are first reduced to terms involving event constructors, as described above. The semantic tableau procedure is applied, as before, but order relations on events are regarded as linear inequalities in a real number domain, and real time event constructors are replaced by arithmetic operations in that domain. Linear arithmetic and linear inequalities in a real number domain are decidable by a Presburger procedure, thus maintaining the decidability of the logic.

3.1 The Elevator Example

The objective of the Interval Logic specification is to express precisely and formally the behavior required from the elevator, including its behavior under failure conditions. It is also an objective to express as few constraints on that behavior as possible while still ensuring correct behavior. It is, perhaps, easier to provide a specification that describes the elevator in minute and mechanistic detail, but to do so precludes, or at least makes much less obvious, many valid implementations that are structured rather differently. Our specification, indeed, permits quite a wide range of behaviors; elevators can still be found in operation that demonstrate some of the less obvious strategies permitted by the specification.

odd! Thus the extension must not expose the numeric nature of time.

Further, temporal logics mask quantifications over time. An explicit representation of time could require that those temporal quantifications be explicit, complicating both the formulae and also deduction involving the formulae. If it is possible to hide the quantifiers, and to process them automatically during deduction, as it is with temporal logics, we should try to do so.

The interval logic can be extended to include real time by:

- imposing real time bounds on the length of intervals, and
- allowing events to be defined by real time offsets from other events.

Defining events by real time offsets is achieved by two new operators $+, -$ syntactically defined by

$+, -$: event $\times$ duration constant $\rightarrow$ event.

Thus if E is an event then so are $E+1$ second and $E-1$ day.

Bounds on the length of intervals are provided by two relational operators, syntactically defined by

$>, <$: duration constant $\rightarrow$ boolean.

These relational operators are monadic because they relate the length of the enclosing context to the duration constant. Used within an interval, they therefore relate the length of that interval to the constant. Thus, if I is an interval, $[I]$ <1 second is a boolean predicate on the length of that interval. Similarly, we might write $[I]$ >10 seconds $\wedge \Diamond$ x=4.

The relational operators can be derived form the event constructors by defining a event offset from the start of the interval and determining whether that event lies within the interval. However, the availability of the relational operators adds directness and clarity to the specifications.

These extensions to interval logic are clean and appear sufficient to describe almost all real time constraints directly and easily. They do not

Floors

The floors are 0 to n, and the elevator will not go outside this range. There can be no down button on floor 0, and no up button on floor n.

1. $\neg\text{atfloor}(-1) \wedge \neg\text{atfloor}(n+1)$
2. $\neg\text{light}(0, \text{down}) \wedge \neg\text{light}(n, \text{up})$
3. $\neg\text{request}(0, \text{down}) \wedge \neg\text{request}(n, \text{up})$

The elevator is at only; one floor at a time and moves only to adjacent floors.

4. $b \neq a \wedge \text{atfloor}(a) \supset \neg\text{atfloor}(b)$
5. $[\text{atfloor}(a) \Rightarrow_{\text{before}} (\text{atfloor}(a+1) \vee \text{atfloor}(a-1))] \Box \text{atfloor}(a)$

Derived Predicates

To simplify the specifications, we introduce a derived event *newsrequest,* since requests are significant only if there is not already an outstanding request, if the elevator does not already have its doors open at the requested floor, and if the elevator is in service.

6. $\text{newrequest}(a, dir) = \text{request}(a, dir) \wedge \neg\text{light}(a, dir) \wedge \text{closed}(a) \wedge \text{inservice}$

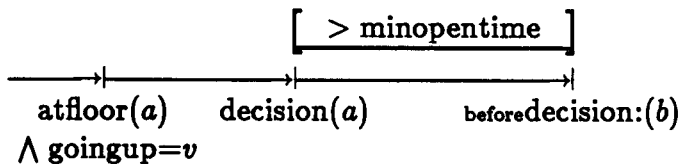
We also introduce an auxiliary event *decision* to represent the moment at which the elevator decides what to do next. The event decision(a) occurs sometime after the doors open and before the elevator leaves at that floor. If the elevator does not stop at floor a, the event occurs some time between being at floor and not being at floor a. Note that, at the time of the event decision(a), atfloor(a) must still be true.

7. $[\text{atfloor}(a) \Rightarrow_{\text{before}} \neg\text{atfloor}(a)] \neg * \text{open}(a) \supset * \text{decision}(a)$
8. $[(\text{atfloor}(a) \Rightarrow \text{open}(a)) \Rightarrow_{\text{before}} \neg\text{atfloor}(a)] * \text{decision}(a)$

The predicate *goingup* is introduced to represent the decision made by the elevator about which direction to move. The predicate is true if the next floor that will be visited is above the current floor, and false if it is below.

It must, of course, retain that value until the next decision point. The curious option of remaining at the same floor and thus making a second decision at that floor is necessary in the case that the elevator arrives at a floor in response to a request indicating continued travel in the same direction, but the request then made inside the elevator is for travel in the other direction. The real time constraint is imposed to allow the passengers time to enter the elevator and press a button.

9. $[((\text{atfloor}(a) \wedge \text{goingup} = v) \Rightarrow \text{decision}(a))$
$\Rightarrow_{\text{before}} \text{decision} : (b)] > \text{min\_open\_time}$
$\wedge\, b > a \supset \Box\text{goingup}$
$\wedge\, b < a \supset \Box\neg\text{goingup}$
$\wedge\, b = a \supset \Box\text{goingup} = v$

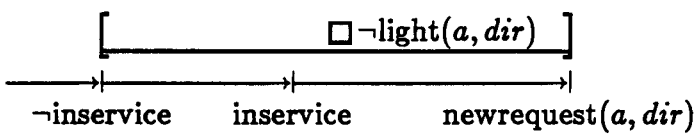


Lights

The lights are used not only to represent the lights visible to the passengers, but also to provide the memory of pending requests. Others might prefer to introduce an additional predicate to represent the pending requests explicitly.

While out of service the lights must not be lit, and following a return to service the lights must not be lit until a new request has been made.

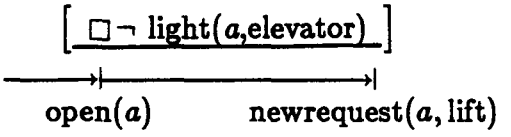
10. $[\neg\text{inservice} \Rightarrow (\text{inservice} \Rightarrow_{\text{before}} \text{newrequest}(a, dir))]\,\Box\neg\text{light}(a, dir)$



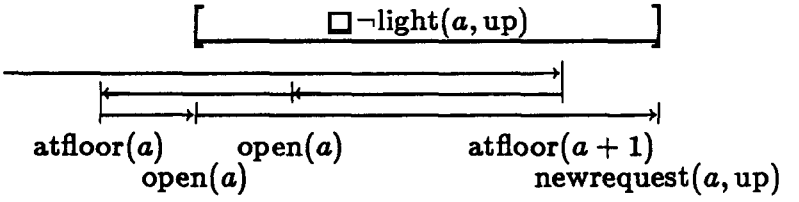
Three axioms define when the lights must not be lit between the satisfaction of a request and the making of the next request. The case for

the elevator light is simple,but the other cases must consider the direction of motion of the elevator and also ensure that the prohibition applies from the first time that the doors open at that floor.

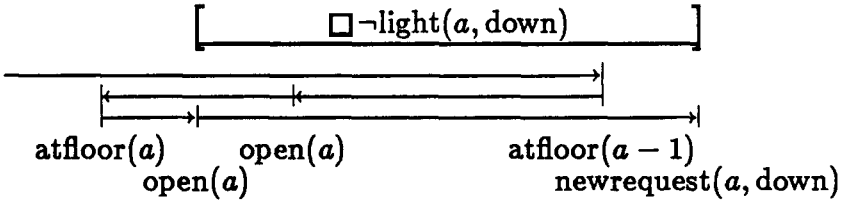
11. $\left[\ \text{open}(a \Rightarrow \textbf{before} \text{newrequest}(a,\text{elevator})\ \right] \Box\neg\text{light}(a,\text{elevator})$



12. $[\textbf{before}((\text{atfloor}(a)\Rightarrow\text{open}(a))\Leftarrow\text{open}(a)\Leftarrow\text{atfloor}(a+1))$
$\Rightarrow\textbf{before}\text{newrequest}(a,\text{up})]\,\Box\neg\text{light}(a,\text{up})$

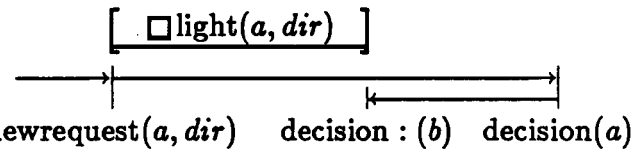


13. $[\textbf{before}((\text{atfloor}(a)\Rightarrow\text{open}(a))\Leftarrow\text{open}(a)\Leftarrow\text{atfloor}(a-1))$
$\Rightarrow\textbf{before}\text{newrequest}(a,\text{down})]\,\Box\neg\text{light}(a,\text{down})$



Another axiom defines when the lights are required to be illuminated. The lights can be turned off as early as the previous decision point, i.e., shortly before reaching the requested floor. They can remain lit for longer, but other axioms require that they be out at least by the time that the doors are open at the requested floor. The lights need only remain on so long as the elevator is in service.

14. $[\text{newrequest}(a, dir) \Rightarrow$
$\text{before}(\text{decision} : (b) \Leftarrow \text{decision}(a) \wedge ((dir = \text{up} \wedge \text{goingup}) \vee$
$(dir = \text{down} \wedge \neg\text{goingup}) \vee$
$dir = \text{elevator}))\ ]$
$\Box\text{inservice} \supset \Box\text{light}(a, dir)$
$\wedge [\Rightarrow \neg\text{inservice}]\, \Box\text{light}(a, dir)$



Movement

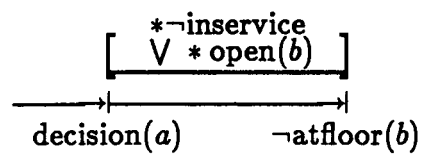
This axiom is an elevator scheduling constraint that requires continued motion in one direction so long as there are further requests outstanding in that direction. When the elevator decides to change its direction of motion, i.e., when goingup changes from false to true or from true to false, there must be no further request outstanding in the original direction of motion.

15. $b < a \quad \supset \quad [\text{before}\,\text{goingup} \Rightarrow]\,\text{atfloor}(a) \supset \neg\text{light}(b, dir)$

16. $b > a \quad \supset \quad [\text{before}\,\neg\text{goingup} \Rightarrow]\,\text{atfloor}(a) \supset \neg\text{light}(b, dir)$

When appropriate, the elevator will stop and open its doors. Fast elevators need time to decelerate and stop, time that is not provided by this version of the specifications. The necessary modifications do not affect these two axioms but rather impose a speed dependent advance on the decision point defined in axiom 7.

17. $b \geq a \quad \supset \quad [(\text{decision}(a) \wedge \text{goingup} \wedge (\text{light}(b, \text{up}) \vee \text{light}(b, \text{elevator})))$
$\Rightarrow \neg\text{atfloor}(b)]\, * \text{open}(b) \vee\ * \neg \mathit{inservice}$

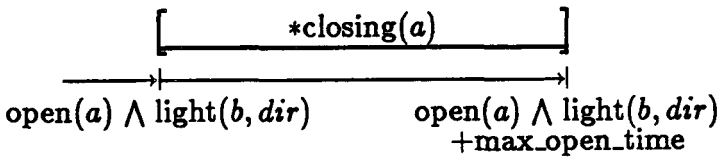


18. $b \leq a \supset [(\text{decision}(a) \wedge \neg\text{goingup} \wedge (\text{light}(b, \text{down}) \vee \text{light}(b, \text{elevator})))$
$\Rightarrow \neg\text{atfloor}(b)]* \text{open}(b) \vee * \neg\text{inservice}$

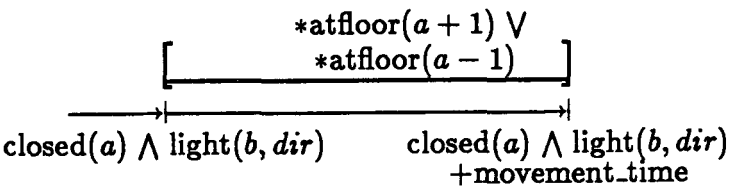
These requirements allow the wide range of behavior that we encounter in elevators, as for instance in allowing the elevator to always return to the ground floor, in allowing the elevator a home floor when inactive, or even in allowing the cattle car to stop at every floor regardless.

The local liveness axioms require that the elevator should not stay at one floor indefinitely if there are requests outstanding from other floors. The first of the two axioms constrains the doors to close within a time constraint if they are not obstructed. The second requires timely movement to an adjacent floor if the elevator is in service.

19. $b \neq a \supset [(\text{open}(a) \wedge \text{light}(b, dir)) \Rightarrow$
$(\text{open}(a) \wedge \text{light}(b, dir)) + \text{max\_open\_time}]$
$\Box(\text{inservice} \wedge \neg\text{obstructed}(a)) \supset * \text{closing}(a)$



20. $b \neq a \supset [(\text{closed}(a) \wedge \text{light}(b, dir)) \Rightarrow$
$(\text{closed}(a) \wedge \text{light}(b, dir)) + \text{movement\_time}]$
$\Box\text{inservice} \supset (*\text{atfloor}(a+1) \vee * \text{atfloor}(a-1))$



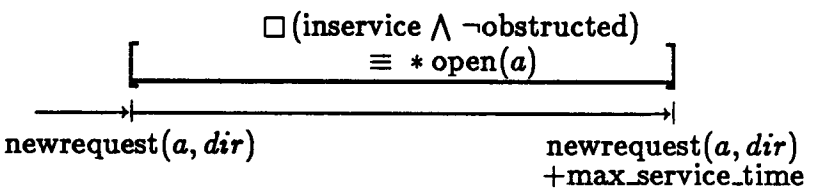
Service Specification

We must next provide our elevator with a service specification. Basically, the service specification states that if a request is made for floor a, then

eventually the elevator will be at floor a with the doors open. As discussed above, we must temper this idealistic requirement with the possibility that the elevator may go out of service. We must also allow for the possibility that the doors may be obstructed to prevent them from closing. We can now state an informal service requirement:

If a request is made for floor a by pressing a button inside the elevator or at that floor, and if, throughout a sufficiently long interval commencing with the request, the elevator is never out of service and the doors are never obstructed, the elevator will eventually be at floor a with its doors open.

21. $[\text{newrequest}(a, dir) \Rightarrow \text{newrequest}(a, dir) + \text{max\_service\_time}]$
$\Box(\text{inservice} \wedge \neg\text{obstructed}) \quad \equiv \quad * \text{open}(a)$



It is possible to elaborate this requirement to allow occasional obstruction of the doors while still guaranteeing service, but at the cost of greatly complicating the specification. The complexity arises not from any inability of the specification language but from the inherent complexity of determining to what extent it is possible to obstruct the doors while still requiring the elevator to provide timely service.

Door opening and closing

We now encounter a sequence of relatively simple axioms that closely control the opening and closing of the doors. Their interest lies largely in the extent to which real time constraints are necessary to specify this aspect of the elevator.

Opening, open, closing, and closed are complete and mutually exclusive.

22. $\text{opening}(a) \vee \text{open}(a) \vee \text{closing}(a) \vee \text{closed}(a)$
$\wedge \quad (\text{opening}(a) \vee \text{open}(a)) \equiv \neg(\text{closing}(a) \vee \text{closed}(a))$
$\wedge \quad (\text{opening}(a) \vee \text{closing}(a)) \equiv \neg(\text{open}(a) \vee \text{closed}(a))$

23. $[\text{open}(a) \Rightarrow_{\text{before}} \text{closing}(a)]\,\Box\,\text{open}(a)$
$\wedge \quad [\text{closed}(a) \Rightarrow_{\text{before}} \text{opening}(a)]\,\Box\,\text{closed}(a)$

The elevator must be at a floor to open its doors and the doors of the elevator and that floor open and close together.

24. $\text{opening(elevator)} \equiv \quad \exists a : 0 \leq a \leq n \wedge \text{opening}(a)$

25. $0 \leq a \leq n \supset [\text{opening}(a) \Rightarrow \text{closed}(a)]\,\Box\,\text{atfloor}(a)$
$\wedge \quad \text{opening(elevator)} \equiv \text{opening}(a)$
$\wedge \quad \text{open(elevator)} \equiv \text{open}(a)$
$\wedge \quad \text{closing(elevator)} \equiv \text{closing}(a)$
$\wedge \quad \text{closed(elevator)} \equiv \text{closed}(a)$

The next five axioms place real time constraints on the sequence of opening and closing actions of the doors, allowing for the possibility that the doors may be obstructed. The last axiom states that the doors are only obstructed while closing.

26. $[\text{opening}(a) \Rightarrow \text{open}(a)]\,\Box\,\text{inservice} \supset \; < \text{opening\_time}$

27. $[\text{closing}(a) \Rightarrow \text{closed}(a)]\,\Box\,(\textit{inservice} \wedge \neg\text{obstructed}(a))$
$\supset \; < \text{closing\_time}$

28. $[\text{obstructed}(a) \Rightarrow \text{opening}(a)]\,\Box\text{inservice} \supset \; < \text{reaction\_time}$

29. $[(\text{obstructed}(a) \Rightarrow \text{open}(a)) \Rightarrow \text{closing}(a)]\,\Box\,\text{inservice} \supset \; < \text{dwell\_time}$

30. $[\text{open}(a) \Rightarrow \text{closing}(a)] > \text{min\_open\_time}$

31. $[\text{obstructed}(a) \Rightarrow]\text{closing}(a)$

Part II

Extending Interval Logic to Include Processes

Section 4

Extending Interval Logic to include Processes

4.1 Introduction

Temporal logic has been found useful for specifying distributed asynchronous systems. Traditionally, such specifications have been expressed as interacting state machines, but that approach inevitably suffers from a risk of over specification since the state machine describes an implementation rather than the requirement. If the application is such that only one implementation is envisaged, an implementation oriented specification may be acceptable. Other applications, for example communications protocol specifications, envisage many distinct implementations. By specifying the minimum required externally visible behavior, leaving all other aspects to lower levels of description, a more general specification can be obtained that reflects the essential requirements of the distributed system or protocol. A specification that is oriented towards one implementation may discourage or even preclude other equally valid implementations. Specifications expressed in temporal logic do not suffer as severely from implementation bias as do state machine specifications.

Specifications for distributed systems can serve two purposes:

- Service specifications define the externally observable function, the service, provided by the system.
- Implementation or protocol specifications define the manner in which the separate components of the distributed system interact to provide the required function.

The service specification for a distributed system defines the externally observable function of the system, in effect the service provided by the system. A service specification regards the entire distributed system as a single entity, with multiple interfaces at separate nodes of the distributed system. The specification defines how operations at each interface, performed asynchronously, affect results at other interfaces. Ideally, a service specification defines only the behavior visible at the external interfaces, without suggesting any internal structure for the system.

Many service specifications define that all operations at the external interfaces be serializable, a characteristic that is often desirable for user interfaces. Such specifications can often be expressed with simpler specification languages that provide only the concepts of parallel operation and of atomicity.

The implementation or protocol specification for a distributed system defines the manner in which the separate components of the distributed system interact with each other so as to provide the required function. The internal implementation of a non-trivial distributed system is rarely serializable, and thus the language for expressing implementation specifications requires more flexibility.

An implementation specification defines separately the behavior of each component, so that each distributed component can be implemented independently. The specifications describe how the components communicate with each other using a communication facility, which is defined by a service specification, as is shown in Figure 4.1. The communication facility

is, of course, itself a distributed system for which there is, in addition to the service specification, also an implementation specification dependent on an even more primitive communication mechanism. In many distributed systems, the hierarchy of such specifications is several levels deep.

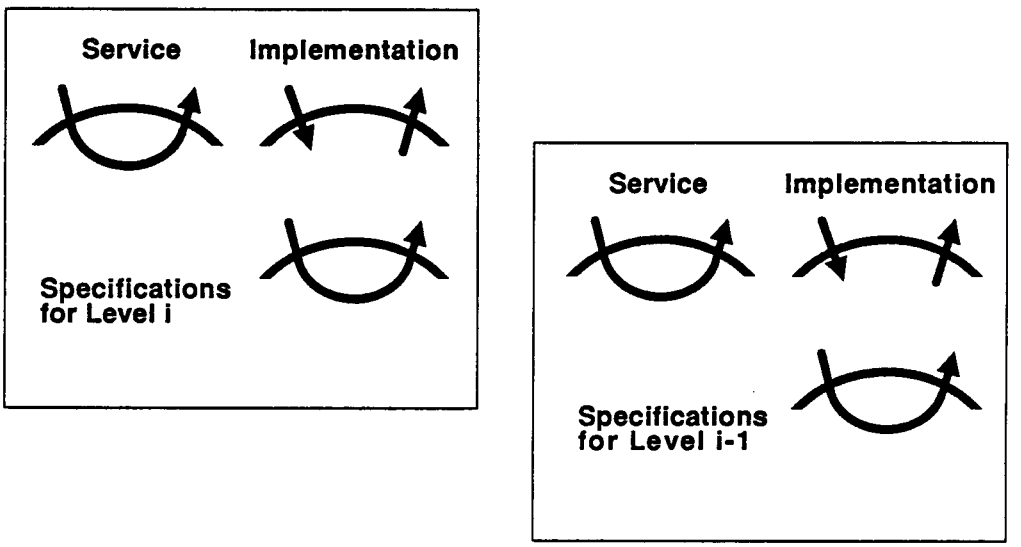


Figure 4.1: Specification of a Level in the Protocol Hierarchy.

If there are to be several independent implementations of some of the components, in the future even if not immediately, it is important that the implementation specification describe only how the components interact with each other without unnecessarily constraining the internal implementation of any component. The ideal specification is one in which

- any component, that satisfies its specification, will operate satisfactorily in the system, and
- any component, that operates satisfactorily in the system, will satisfy the specification.

If both a service specification and an implementation specification have been constructed for a distributed system, it is possible to validate the implementation specification by confirming that it satisfies the service specification. This ability is very valuable for the implementation specification is often quite complex and prone to error, while the service specification is much shorter and simpler. Unfortunately, the current state of the art, and particularly of tools, has not yet advanced to the point at which such a validation is feasible for typical distributed systems.

4.2 Service and Implementation Specifications

The service specification is intended to describe only the service provided to the user that he can depend on obtaining from all implementations, that is the externally observable behavior required of all implementations. The service specifications should allow as much flexibility as possible in the internal implementation of the system, and thus should avoid describing internal structure so far as is possible in a specification that is easy to understand.

Because the service specification describes a distributed system with several external interfaces that operate asynchronously, it is to be expected that the specification will have to describe asynchronous operation of those interfaces. But the service specification is intended to conceal rather than to reveal the internal component structure of the implementation, and the internal process structure should not be exposed. If these internal structures are visible through the service specification, the opportunities for hierarchical composition are reduced. Minor changes in the design at one level, changes that should have been purely local, may be propagated up the hierarchy and may affect the specification and validation of higher levels.

In contrast, an implementation specification describes the implementation of a distributed system as a set of interacting components. Since the

system will be implemented as a distributed system, each component must be defined separately without reference to the specification of any other component. The interactions between components must be provided by a communications mechanism, for which a specification must also be provided. The implementation specification aims to describe the interactions between components rather than their internal construction, and it is desirable that the specification leave as much flexibility as possible for the internal design of those components.

Consequently, the appropriate form for an implementation specification consists of a set of service specifications describing each of the components, and of service specifications describing the communications mechanisms by which they interact. There may be multiple instances of some components, and the implementation specification should therefore be able to instantiate multiply a single service specification to represent those instances. The communication service will typically have multiple equivalent interfaces at which the service is provided. By specifying each service specification as a module of the specification language, the required flexibility can easily be provided by module instantiation mechanisms, which are very powerful in modern specification languages.

The implementation specification should distinguish between the specifications of components that are intended to be localized, and the specifications of communications mechanisms that are distributed over the entire system. Qualifiers `local` and `distributed` should be provided for module declarations. For example (omitting any module parameterization for clarity)

```
producer:  local module is sensorscanner
consumer:  local module is correlator
shareddata:  distributed module is filesystem
```

When defining a service specification, the interfaces take the form of operations, presumably implemented by procedure calls of the programming language. Some of these interfaces are provided by operations mechanized within the component being specified. Such operations are intended to be

invoked by a process from outside the component. Other interfaces are operations mechanized outside the component, and invoked by a processes from within the component. The two cases are symmetrical and, on each side of the interface, the same information must be provided, i.e. type declaration, semantic effect, etc. It is, of course, important that the distinction between the two be clear in the specifications. Thus, it is necessary that the specification language, when defining an operation, should be able to designate the operation to be external, a feature provided quite frequently by programming languages but seldom by specification languages. An example of such a declaration might be

```
put:  external operation[ data, file[data] -> statuscode ]
```

Other interfaces may be provided by means of shared data structures, and here again the same distinction is required.

The implementation specification must link the interfaces of the components to the interfaces of the communication mechanisms. The linkage can be expressed by equating the two interfaces, thus:

```
a1:  linkage producer.put = filesystem.writerecord[2]
a2:  linkage consumer.get = filesystem.readrecord[3]
```

It is evident that linkage of interfaces within an implementation specification should be between an internal and an external construct, and only when they are type compatible. Direct linkage of the interfaces of two local components precludes a distributed implementation.

4.3 Processes

The concept of a process was created to describe the flow of control within concurrent processing contexts, and processes, or related constructs, are provided by programming languages that aim to support real time programming. In contrast, specification languages have generally avoided the concept of process, since flow of control is an implementation oriented con-

cept, distinct from the externally observable behavior defined by a specification. But specification languages must support not only pure specification but also the reasoning that is used to justify the implementation.

At a design level, when establishing that an implementation specification does indeed satisfy its service specification, it is necessary to be able to relate, and thus to be able to distinguish, the processing of the various components. Processes are a convenient mechanism for expressing the progress of each component through its computation. To validate the implementation of a distributed system, expressed in terms of processes, the specification language must also support the concept of processes.

Programming languages provide processes whose declaration introduces not only an (implicit) locus of control but also a relative complex program structuring unit, including the declaration of local variables and parameters. In the purer context of a specification language, it is desirable to define a process solely to denote the locus of control. That declaration may be included within a module structure that also includes other declarations as required.

The extensions to the specification language are, therefore, straightforward. In the specification language prelude, `process` is declared to be a type, as in:

```
process:type
```

Processes are then declared by conventional declarations, as in:

```
p:  process
```

As for other objects, processes may be declared to be external, representing a process that is external to a component and that invokes operations declared within the component.

In Interval Logic, as in other temporal logics, the location of control within the program is expressed by three state predicates, at, in and after. These three predicates relate the location of control to an operation. The operation may be primitive or composite, a primitive operation at one

level of abstraction being, of course, composite at a more detailed level. The predicates carry the intuitive meanings of the location of conrol being "at the beginning of," "within" and "immediately after" the operation. Typically, in the past, these predicates have not defined which process is at that point of the computation.

Extending these predicates to accept processes is direct. $\texttt{at}(p, O)$ denotes that process p is at the beginning of operation O, and similarly for `in` and `after`.

It is now appropriate to consider the meaning that was intended when `at`, `in` and `after` were used without an explicit process. Generally, the intended meaning is intuitively clear, but the implicit quantification over processes required to achieve that meaning is not the same in all examples. Consider one of the axioms that has traditionally defined the meaning of `at`, `in`and `after`.

$$[\,\texttt{at}O \Rightarrow_{\text{before}} \texttt{after}O\,]\,\Box\,\texttt{in}O$$

It is clear that the intent is that the axiom be enclosed within a universal quantification over processes, thus:

$$\forall p, O : [\,\texttt{at}(p, O) \Rightarrow_{\text{before}} \texttt{after}(p, O)\,]\,\Box\,\texttt{in}(p, O)$$

But the useful statement about queues expressed as:

$$\Box\Diamond\texttt{at}\mathit{Enqueue} \;\wedge\; \Box\Diamond\texttt{at}\mathit{Dequeue} \supset\; \Box\Diamond\texttt{after}\mathit{Dequeue}$$

requires local existential quantification, thus:

$$\Box\Diamond\exists p : \texttt{at}(p, \mathit{Enqueue}) \;\wedge\; \Box\Diamond\exists q : \texttt{at}(q, \mathit{Dequeue}) \supset\; \Box\Diamond\exists r : \texttt{after}(r, \mathit{Dequeue})$$

There are other examples where various mixtures of universal and existential quantifications are required, variously bound. Note also that, unlike pure predicate calculus, quantifiers cannot all be brought to the outer levels. Universal quantifiers can be promoted through $\Box$ but not through $\Diamond$, and existential quantifiers through $\Diamond$ but not through $\Box$.

Even though the intended quantification over processes is intuitively obvious, the intuition is based on understanding of the semantics. We have found no systematic method by which it is possible to impute an appropriate quantification to an expression in which the processes are implicit rather than explicit. Consequently, we prefer to reject implicit quantification and to require that all uses of `at`, `in` and `after` define explicitly the process and the operation.

Formally, the three state predicates are defined by following temporal axiomatization.

1. $[\,\mathtt{at}(p,O) \Rightarrow \text{before}\ \mathtt{after}(p,O)\,]\,\Box\,\mathtt{in}(p,O)$
2. $[\,\mathtt{after}(p,O) \Rightarrow \text{before}\ \mathtt{at}(p,O)\,]\,\Box\neg\mathtt{in}(p,O)$
3. $[\,\neg\mathtt{at}(p,O) \Rightarrow \mathtt{after}(p,O)\,]\,\Box\neg\mathtt{at}(p,O)$
4. $[\,\neg\mathtt{after}(p,O) \Rightarrow \mathtt{at}(p,O)\,]\,\Box\neg\mathtt{after}(p,O)$

Axioms 1 and 2 together define $\mathtt{in}(p,O)$ to be true exactly from $\mathtt{at}(p,O)$ to the state immediately preceding $\mathtt{after}(p,O)$. Axiom 3 allows $\mathtt{at}(p,O)$ to be true only at the beginning of the operation, and axiom 4 requires that $\mathtt{after}(p,O)$ be true only immediately following an operation.

The axioms do not imply any specific granularity, duration or mapping of the operation symbol to an implementation. Any interpretation of these state predicate symbols satisfying the above axioms is allowed. In addition, no assumption of operation termination is made.

A further complication arises with the operation argument to `at`, `in` and `after`. Consider a process p and an operation op, declared to be of type `operation[ t, t -> t ]`, with variables a and b also declared to be of type t.

In the expression $\mathtt{at}(p, op(a,b))$ the operation term $op(a,b)$ has type `operation[ -> ]`, and thus we can set the signature of the function `at` to be `state[ function[ process, operation[ ->] -> boolean ]]`. But the Interval Logic also allows other forms for the `at` operator that are not

so easy to render type consistent. The three forms provided in Interval Logic are:

- $\texttt{at}(p, op(a, b))$ which becomes true at the next state in which *op* has been called with parameters *a* and *b*
- $\texttt{at}(p, op)$ which becomes true at the next state in which *op* is called regardless of the values of the parameters
- $\texttt{at}(p, op : (a, b))$ which becomes true at the next state in which *op* is called regardless of the values of the parameters, and which binds the free variables *a* and *b* to the values of the parameters at that time.

All three forms are very useful in constructing specifications.

The second form $\texttt{at}(p, op)$ requires that `at` accept as argument operations of many different types, depending on their parameterizations, and thus `at` be declared within a polyadic generic module, similar to those in the prelude that declare `function` and `operation`. Polyadic generic modules are supported by Revised Special only for predefined theories built into the system; they cannot be defined by the user. Thus it is not possible to introduce `at` as a user defined extension to Revised Special. Indeed, it is clear that the appropriate place for `at` to be declared is within the `operations` module that declares operations that accept parameters.

This form of *at* involves an implicit existential quantification. The term $\texttt{at}(p, op)$ is interpreted to imply a term of the form

$$\exists a, b : [- > *\texttt{at}(p, op(a, b)]\forall c, d : \texttt{at}(p, op(c, d) \supset a = c \wedge b = d$$

placed in the context from which the term $\texttt{at}(p, op)$ is searched for. The construction is essentially similar to that for $*E$, which introduces the additional term $\neg[\Rightarrow E]$False in the context where $*E$ indicates that event E must be found. We should note that, whatever the formal semantics of $\texttt{at}(p, op)$, or in deed of $*E$, the decision procedure can process the terms much more efficiently in their original form than in the expanded form.

The third form of at, $\texttt{at}(p, op : a, b)$ not only requires a polyadic generic declaration, but also makes visible the implicit quantification inherent in $\texttt{at}(p, op)$. Additional mechanism is required to bind the cited parameter values to the implicit quantification.

It is necessary to note that the term $\texttt{at}(p, op : a, b)$ introduces a risk of ambiguity into the language if recursive operations are permitted. If control is within operation *op* nested two or more levels of recursion deep, there are alternative choices for the values of a and b. The logic defines the meaning to be an arbitrary choice as to which set of parameters is bound to a and b. Fortunately, recursive operations are very rare in the design of real time systems, though reentrant operations are not. Thus we are careful to avoid in the language any form such as $\texttt{at}(: p, op)$, which would be often ambiguous.

in must be handled similarly to at, but after introduces additional considerations because Interval Logic permits reference to the result of the operation. Prior versions of Interval Logic used $\texttt{after}(p, op(a))$ to constrain the result of the operation to the value of a. This was clearly inappropriate, confusing the parameters of the operation with its result, and introducing context sensitivity, typechecking problems, and lack of expressivity.

We therefore propose the form $\texttt{after}(p, op!a)$ to become true in the next state in which control is after *op* and the result of the operation is equal to a. Indeed $\texttt{after}(p, op!a)$ can be replaced by $\texttt{after}(p, op) \wedge \texttt{result} = a$.

Operations, when they return a result, usually return only a single value as their result, unlike parameters where multiple parameters are common. Consequently, it appears appropriate to orient the specification mechanisms to a single result, as above. When a compound result must be specified, it can be represented as a tuple, as in $\texttt{after}(p, op! < a, b >)$.

It must also be possible to search for the next state in which control is after *op*, and to bind a variable to the result in that state. This can be expressed as $\texttt{after}(p, op! : a)$, which is replaced by $\texttt{after}(p, op)$ and introduces into that context an additional term $[\texttt{after}(p, op) \Rightarrow] \texttt{result} =$

a.

It is also possible to constrain both the parameters and the result of an operation, as in $\texttt{after}(p, op(a, b)!c)$.

Section 5

Extending Revised Special to Include Interval Logic

Some, indeed many, parts of the Interval Logic can be introduced as user defined modules using the standard theory definition mechanisms of Revised Special. But there are aspects of Interval Logic that are not well handled by the standard mechanisms made available to users, and must really be provided through internal modification of the specification and verification system if Interval Logic is to be effective. Three of these aspects are:

- The binding of parameters and results to operations for the `at`, `in`, and `after` predicates, as discussed above.
- The decision procedure for Interval Logic should be integral to the internal prover if it is to be effective. No axiomatization of Interval Logic has been developed, and if an axiomatization existed then it would be excessively complex. Thus, attempting to prove theorems in Interval logic by use of some predefined set of axioms and theorems would be tedious and frustrating to the user.
- While it is possible to define Interval Logic solely in terms of prefix functions that can be declared in user defined modules, the result

lacks most of the intuitive clarity that is a major part of the value of Interval Logic. The infix syntax of Interval Logic complicates the parsing of temporal expressions, but is important to the utility of the logic.

Consequently, we believe that a realistic extension of Revised Special and of the Enhanced HDM specification and verification system must depend on extensions to the code of the parser, typechecker, and prover.

5.1 Extensions to the Syntax

The required extensions to the syntax are quite straight forward and are marked below by the rule in the left margin. They are, in order:

- Modules can be optionally qualified as local or distributed.
- Axioms can be defined to be linkages or as assumptions. The existing assuming clause can be used to define assumptions about module parameters, but is unsatisfactory for stating assumptions about the behavior of constants defined to be external. The handling of assumptions is equivalent to that for formulae in the assuming clause.
- Constants can be optionally declared to be external.
- Temporal expressions are introduced as boolean expressions, comprising expressions prefixed by the operators `henceforth`, `eventually` and *, and the interval expressions denoted by `[ ]` .
- Intervals are defined by the operators `->` and `<-`, or by events. The arguments to the `->` and `<-` operators are optional, and intervals can be prefixed by a single *.
- An event is a boolean expression, optionally prefixed by an *.

- The unary prefix operators are extended to include the unary relational operators < and >.
- Operation application is extended to include the : and ! qualifiers.

5.1.1 Revised User Syntax

The conventions of the syntactic description are:

- italics are used to indicate syntactic classes and metavariables that range over syntactic classes.
- a `printer font` is used to indicate language keywords,
- choice is indicated by vertical bar $\mid$ in the normal fashion.
- optional inclusion of a clause A is indicated by square parentheses $[\, A \,]$.
- iteration of a clause B one or more times is written with braces and a plus sign $\{\, B \,\}^{+}$.
- repetition zero or more times is written with an asterisk $\{\, B \,\}^{*}$.
- a symbol below the plus or asterisk indicates a clause separator, i.e. to indicate separation of each clause by a comma $\{\, B \,\}^{*}_{,}$.

module_decl ::= [*comment_list*]
ident : [`local` | `distributed`]
`module` [*formal_params*]
{ *module_synonym* | *module_body* }

module_synonym ::= `is` [`new`] *module_name* [*comment_list*]

module_body ::= [[*comment_list*] *using_clause*]
[[*comment_list*] *exporting_clause*]
[[*comment_list*] *assuming_clause*]
[*theory_clause*]
[*proof_clause*]
`end` [*ident*] [*comment_list*]

using_clause ::= `using` { *module_name* }*, |
`mapping` { *module_name* }*, `onto` { *module_name* }*,

exporting_clause ::= `exporting` [*entity_name_list*]
[`with` { *module_name* }*,]

assuming_clause ::= `assuming` { *assuming_decl* }*

assuming_decl ::= *module_decl* | *const_decl* | *var_decl* | *type_decl* |
formula_decl | *comment_list*

theory_clause ::= `theory` { *theory_decl* }*

theory_decl ::= *module_decl* | *const_decl* | *var_decl* | *type_decl* |
formula_decl | *axiom_decl* | *comment_list*

proof_clause ::= `proof` [*using_clause*] { *proof_decl* }*

proof_decl ::= *module_decl* | *const_decl* | *var_decl* | *type_decl* |
formula_decl | *prove_decl* | *comment_list*

prove_decl ::= *ident* : {`prove` | `verify` }
instance_list [`from` *instance_list*]

instance_list ::= { *instance* }$^{+}$,

instance ::= *entity_name* [*substitutuion*]

substitutuion ::= { { *entity_name* <- *expr* }*, }

formula_decl ::= *ident* :{`formula` | `theorem` | `lemma` }
[*synonym* | *formula_expr*]

formula_expr ::= *sc_expr* [*changes_clause*]

axiom_decl ::= *ident* :{`axiom` | `assumption` | `linkage` }
[*synonym* | *formula_expr*]

var_decl ::= *ident_list* :{`variable` | `var` }*type_name*
[*synonym*]

const_decl ::= *entity_name_list* : *type_name*
[*synonym* | *const_expr* | *where_clause*]

synonym ::= `is` *entity_name*

const_expr ::= {= | == }*sc_expr*

where_clause ::= `where` *expr changes_clause*

changes_clause ::= `changes` {*entity_name_list* | `nothing` }

type_decl ::= *entity_name_list* : `type` [*type_syn* | *type_deriv*]

type_syn ::= `is` *type_name*

type_deriv ::= `from` *type_name*

type_name ::= *entity_name*

formal_params ::= [{ *formal_param_elt* },$^{+}$]

formal_param_elt ::= *ident_list* : *formal_type*

formal_type ::= `type` | *type_name*

entity_name_list ::= { *entity_name* },$^{+}$

entity_name ::= *number* | { *ident* [*actual_param*] }.$^{+}$

module_name ::= { *ident* [*actual_param*] }.$^{+}$

actual_param ::= [[*type_or_expr_list*] [-> [*type_name*]]]

type_or_expr_list ::= { *type_name* | *expr* },$^{+}$

sc_expr ::= { *expr* };$^{+}$

expr ::= *imp_expr* [`iff` *imp_expr*]

imp_expr ::= *or_expr* [`implies` *imp_expr*]

or_expr ::= *and_expr* [`or` *or_expr*]

and_expr ::= *temporal_expr* [`and` *and_expr*]

temporal_expr ::= `henceforth` *temporal_expr* |
`eventually` *temporal_expr* |
* *temporal_expr* |
[*interval*] *temporal_expr* |
rel_op_expr

interval ::= [*] (*int_term*) | *int_term*

int_term ::= [*interval*] -> [*interval*] |
[*interval*] <- [*interval*] |
event

event ::= [*] *expr*

rel_op_expr ::= *plus_minus_expr* {= | ~= | < | <= | > | >= } *plus_minus_expr*

plus_minus_expr ::= [*plus_minus_expr* {+ | - }] *times_or_div_expr*

times_or_div_expr ::= [*times_or_div_expr* {* | / }] *unary_minus_expr*

unary_minus_expr ::= *op_expr* | - *plus_minus_expr* | {< | > } *plus_minus_expr*

op_expr ::= *not_expr* | *op_construct*

not_expr ::= [not] *misc_expr*

misc_expr ::= *simple_expr* | *if_expr* | *case_expr* |
opspec_expr | *hoare_sentence*

simple_expr ::= {*entity_name* |
paren_expr |
lambda_expr |
quan_expr } [']
{*with_expr* ['] |
({ *expr* },* [: (*expr*)] [! [:] *expr*]) ['] }*

paren_expr ::= (*sc_expr*)

lambda_expr ::= (lambda [*entity_name_list*] -> *type_name* : *expr*)

quan_expr ::= ({exists | forall } *entity_name_list* : *expr*)

with_expr ::= with [{ *assignments* }+]

assignments ::= { *arg_list* ['] }+ := *expr*

if_expr ::= if *expr* then *expr* else *expr* end

case_expr ::= case { *case_part* },*
else *expr* [*comment_list*] end

case_part ::= {*expr* : *expr* } | [*comment_list*]

opspec_expr ::= opspec *op_expr*
case { *case_part* }$^{*}_{,}$
else *expr* [*comment_list*] end

hoare_sentence ::= {*expr* } *sc_expr* {*expr* }

op_construct ::= *compound_op* | *repeat_op* | *while_op* | *assign_op*

compound_op ::= begin *sc_expr* end

repeat_op ::= repeat *sc_expr* until *expr*

while_op ::= while *expr* do *op_expr*

assign_op ::= *entity_name* := *expr*

ident_list ::= { *ident* }$^{+}_{,}$

comment_list ::= { *comment* }*

comment ::= (* { *Any sequence of characters except* (* *and* *) | *comment* }* *)

5.2 Extensions to the Typechecking

Most of the extensions to the typechecker are direct consequences of the extensions to the syntax, making use of an extended prelude.

Additional special handling is required for `at`, `in` and `after`. These operators must be polyadic, as are operations. The ; and ! qualifiers also require special processing.

The optionally omitted interval arguments to the -> and <- operators must be inserted, using the now and then interval constants.

It is necessary for the typechecker to distinguish between two uses of the * monadic prefix operator. If the term occurs where an interval term is expected, the operator is replaced by an application of the `find` function; if a boolean term is expected, the `occurs` function is used.

The type checker makes use of a extensions to the prelude:

- the `operations` module includes polyadic declarations for `at`, `in` and `after`
- the `intervallogic` module introduces the remaining interval logic functions.

Much of the complexity of this module declaration comes from the requirement that the -> and <- operators accept either intervals or events as arguments, thus requiring extensive overloading of the declarations.

5.2.1 The Revised Prelude

The revised modules of the prelude are:

```
operation: module[ characteristic: type ]

exporting operation, noop, compose, assignops, whileop, repeatop, hoare

theory

    operation: type

    noop: operation

    (* diadic infix oparator   ;   *)
    compose: function[ operation, operation -> operation ]

    assignops: module[ t: type ]
        exporting assign
        theory
            (* diadic infix operator := *)
            assign: function[ state[t], t -> operation ]
        end assignops

    (* diadic mixfix operator   while _ do _   *)
    whileop: function[ boolean, operation -> operation ]

    (* diadic mixfix operator   repeat _ until _   *)
    repeatop: function[ operation, boolean -> operation ]

    (* triadic mixfix operator   { _ } _ { _ }   *)
    hoare: function[ boolean, operation, boolean -> boolean ]

    at, in, after: state[function[ process, operation -> boolean ]]

end operations
```

```
intervallogic: module

using operations
exporting
   process, interval, event, now, then, Int,
   moduleee.forward, moduleee.backward,
   moduleie.forward, moduleie.backward,
   moduleei.forward, moduleei.backward,
   moduleii.forward, moduleii.backward,
   modulene.forward, modulene.backward,
   moduleet.forward, moduleet.backward,
   moduleni.forward, moduleni.backward,
   moduleit.forward, moduleit.backward,
   henceforth, eventually, occurs, find,
   before, atend

theory

    process: type
    interval: type
    event: type is state[boolean]
    now,then: const event
    vacuous,failed: const interval

    p: process
    e: event
    I: interval
    op: operation
    b: state[boolean]

    (*  diadic mixfix operator   [ _ ] _   *)
    Int: state[function(interval,state[boolean]->boolean)]

    before: function(interval->interval)
    atend: function(interval->interval)

    (*  monadic prefix operator   eventually   *)
    eventually: state[function(state[boolean]->boolean)]
    (*  monadic prefix operator   henceforth   *)
    henceforth: state[function(state[boolean]->boolean)]

    (*  monadic prefix operator   *   *)
    occurs: state[function(state[boolean]->boolean)]
    (*  monadic prefix operator   *   *)
    find: function(state[boolean]->state[boolean])
```

```
moduleee:module
    exporting forward, backward
    theory
    (*  diadic infix operator  ->  *)
    forward: state[function(event,event->interval)]
    (*  diadic infix operator  <-  *)
    backward: state[function(event,event->interval)]
end moduleee

moduleie: module
    exporting forward, backward
    theory
    (*  diadic infix operator  ->  *)
    forward: state[function(interval,event->interval)]
    (*  diadic infix operator  <-  *)
    backward: state[function(interval,event->interval)]
end moduleie

moduleei:module
    exporting forward, backward
    theory
    (*  diadic infix operator  ->  *)
    forward: state[function(event,interval->interval)]
    (*  diadic infix operator  <-  *)
    backward: state[function(event,interval->interval)]
end moduleei

moduleii:module
    exporting forward, backward
    theory
    (*  diadic infix operator  ->  *)
    forward: state[function(interval,interval->interval)]
    (*  diadic infix operator  <-  *)
    backward: state[function(interval,interval->interval)]
 end moduleii
```

```
modulene:module
    using moduleee
    exporting forward, backward
    theory
    (*  monadic prefix operator  ->  *)
    forward: state[function(event->interval)]
             is (lambda e -> interval: moduleie.forward(now,e))
    (*  monadic prefix operator  <-  *)
    backward: state[function(event->interval)]
              is (lambda e -> interval: moduleie.forward(now,e))
end modulene

moduleet:module
    using moduleee
    exporting forward, backward
    theory
    (*  monadic postfix operator  ->  *)
    forward: state[function(event->interval)]
             is (lambda e -> interval: moduleie.forward(e,then))
    (*  monadic postfix operator  <-  *)
    backward: state[function(event->interval)]
              is (lambda e -> interval: moduleie.backward(e,then))
end moduleet

moduleni:module
    using moduleei
    exporting forward, backward
    theory
    (*  monadic prefix operator  ->  *)
    forward: state[function(interval->interval)]
             is (lambda I -> interval: moduleie.forward(now, I))
   (*  monadic prefix operator  <-  *)
    backward: state[function(interval->interval)]
              is (lambda I -> interval: moduleie.backward(now,I))
end moduleni

moduleit:module
    using moduleie
    exporting forward, backward
    theory
    (*  monadic postfix operator  ->  *)
    forward: state[function(interval->interval)]
             is (lambda I -> interval: moduleie.forward(I,then))
    (*  monadic postfix operator  <-  *)
    backward: state[function(interval->interval)]
              is (lambda I -> interval: moduleie.backward(I,then))
end moduleit
```

```
    atin1: axiom
           henceforth(at(p,op) implies in(p,op))

    atin2: axiom
           [ at(p,op) -> before(after(p,op)) ] henceforth(in(p,op)))

    atin3: axiom
           henceforth(after(p,op) implies not in(p,op))

    atin4: axiom
           [ after(p,op) -> before(at(p,op)) ] henceforth( not in(p,op)))



    vacaxiom1: axiom  Int(vacuous,b) iff true

    vacaxiom2: axiom  I ~= failed implies
                      forward(e,vacuous) = forward(vacuous,e)
                      = forward(I,vacuous) = forward(vacuous,I)
                      = backward(e,vacuous) = backward(vacuous,e)
                      = backward(I,vacuous) = backward(vacuous,I)
                      = vacuous

    failedaxiom1: axiom  Int(failed,b) iff false

    failedaxiom2: axiom  forward(e,failed) = forward(failed,e)
                         = forward(I,failed) = forward(failed,I)
                         = backward(e,failed) = backward(failed,e)
                         = backward(I,failed) = backward(failed,I)
                         = failed

end intervallogic
```

5.3 Extensions to the Prover

The necessary extensions to the prover are described in a subsequent section of the report.

Section 6

An Example: Specification of the Alternating Bit Protocol

This section demonstrates the use of processes in interval logic, with an example from the area of communication protocols, the Alternating Bit protocol. This protocol can be considered as a rather simple, but not trivial, example of a Data Link layer protocol. (The concept of layering is specified in the ISO OSI Basic Reference Model [ISO82].) The objective of the Data Link layer protocol is to detect and possibly correct errors that may occur in the underlying Physical Link layer.

6.1 The Alternating Bit Protocol

The Alternating Bit protocol is used to provide a reliable message communication over an unreliable transmission line through repeated transmission. It considers messages one at a time and cannot proceed to the next message until it receives acknowledgment that its current message has been received correctly. The message is placed in a packet with a one-bit sequence number (hence the name of the protocol), and an acknowledgment is assumed to consist of the return of the same packet (although only the sequence

number is really required). Several packets may be in transit simultaneously. The protocol recovers successfully from packets lost, duplicated, or delayed by the transmission line, as long as no packets arrive out of order.

The scenario of sending one message can be described by assuming that a user process sends a message m by invoking the operation `send(i,m)`. The message will be placed in the Sender queue. The Sender process dequeues the message (through `dq!<i,m>`) and transmits it together with the Sender's current sequence number v as a packet through `transmit(i,<message,m,v>)`. The Receiver entity gets packets by means of `receive!<i,<message,m,v>>`. Acknowledgments are sent from the Receiver entity through `transmit(i,<ack,m,v>)` and received by the Sender entity by means of `receive!<i,<ack,m,v>>`. Messages from the Receiver process can be stored by `enq(<i,m>)` in the Receiver queue from which the receiving user can dequeue it by the operation `rec!<i,m>`.

We have described here the protocol as a communication between a single Sender and a single Receiver. To demonstrate the process mechanisms better, the example specifies communication between many senders and receivers. Thus every user invoking the `send` operation must specify not only the message but also its destination, as must the `transit` operation. Similarly, the `receive` and `rec` operations supply a source as a part of their result. We assume that the sender still only sends packets one at a time, and does not proceed to the next message until it has received an acknowledgment for the previous packet, but the receiver must, of course, be able to receive packets from several sources simultaneously.

6.2 Specification of the Service Used and the Service Provided

The Alternating Bit protocol converts the unreliable queue service of the physical transmission medium into a reliable queue. The service specifications for both the Alternating Bit protocol and of the transmission medium

are based on specifications for queues. But before proceeding to those relatively complex queue specifications, we first consider a simple queue specification that will be used to provide local input and output queues for the Senders and Receivers of the communication mechanism.

The simple queue specification was introduced above, where axiom a1 was cited to require that messages be dequeued in the correct order. Axiom a2 also requires that messages be enqueued before being dequeued. Axioms a3 and a4 are liveness requirements; a3 ensures that messages will be dequeued, while a4 requires that the enqueue operation always return. Axiom a5 is an assumption about the behavior of the user, that every message enqueued is distinct, an assumption without which it is not possible to write the specification using temporal logic.

```
simplequeue: module[ data: type ]
    exporting  enq, dq, pa, pb

theory
    pa, pb: external process
    a, b: var data

    enq: operation[ data -> ]
    dq:  operation[ -> data ]

a1: axiom
    [ *(at(pa,enq(a)) -> at(pa,enq(b)))
      <- (after(pb,dq!a) -> after(pb,dq!b)) ] true

a2: axiom
    [ *at(pa,enq(a)) <- after(pb,dq!a) ] true

a3: axiom
    *at(pa,enq(a)) and henceforth *at(pb,dq) implies *after(pb,dq!a)

a4: axiom
    henceforth [ *at(pa,enq) <- after(pa,enq) ] true

    (* assumption about external behavior *)
a5: assumption
    [ at(pa,enq(a)) -> ] b~=a implies  not *at(at(pa,enq(b))

end simplequeue
```

We next define a reliable queue serving many inputs and outputs. The extensions to the simple queue are direct and obvious, except for axiom q4, which is new and is a fairness requirement that messages from all sources have a chance of being dequeued.

The unreliable queue specification is derived from that of the reliable queue by modification of axioms q3 and q4. In both cases, because of the unreliability, a message must be sent repeatedly to ensure that it will be received.

```
reliablequeue: module[ id, data: type ]
    exporting interface

theory
    i, j: var id
    a, b: var data

    interface: module[ i: id ]
    exporting  pa, pb, enq, dq
    theory
        pa, pb: external process
        enq: operation[ id, data -> ]
        dq:  operation[ -> tuple[id,data]]
    end interface

q1: axiom
    [ *(at(interface[i].pa,enq[i](j,a))
        -> at(interface[i].pa,enq[i](j,b)))
      <- (after(interface[j].pb,dq[j]!<i,a>)
          -> after(interface[j].pb,dq[j]!<i,b>)) ] true

q2: axiom
    [ *at(interface[i].pa,enq[i](j,a))
      <- after(interface[j].pb,dq[j]!<i,a>) ] true

q3: axiom
    [ *at(interface[i].pa,enq[i](j,a)) and *at(interface[j].pb,dq[j])
      ->  *after(interface[j].pb,dq[j]) ] true

q4: axiom
    *at(interface[i].pa,enq[i](j,a))
       and henceforth *at(interface[j].pb,dq[j])
    implies *after(interface[j].pb,dq[j]!<i,a>)

q5: axiom
    henceforth [ *at(interface[i].pa,enq[i])
                 <- after(interface[i].pa,enq[i]) ] true

    (* assumption about external behavior *)
q6: assumption
    [ at(interface[i].pa,enq[i](j,a))
      -> at(interface[i].pa,enq[i](j,a)) ] false

end reliablequeue
```

```
unreliablequeue: module[ id, data: type ]
    exporting interface

theory
    i, j: var id
    a, b: var data

    interface: module[ i: id ]
    exporting  pa, pb, enq, dq
    theory
        pa, pb: external process
        enq: operation[ id, data -> ]
        dq:  operation[ -> tuple[id,data]]
    end interface

u1: axiom
    [ *(at(interface[i].pa,enq[i](j,a))
        -> at(interface[i].pa,enq[i](j,b)))
      <- (after(interface[j].pb,dq[j]!<i,a>)
          -> after(interface[j].pb,dq[j]!<i,b>)) ] true

u2: axiom
    [ *at(interface[i].pa,enq[i](j,a))
      <- after(interface[j].pb,dq[j]!<i,a>) ] true

u3: axiom
    henceforth *at(interface[i].pa,enq[i](j,a))
        and *at(interface[j].pb,dq[j])
    implies *after(interface[j].pb,dq[j])

u4: axiom
    henceforth *at(interface[i].pa,enq[i](j,a))
      and henceforth *at(interface[j].pb,dq[j])
    implies *after(interface[j].pb,dq[j]!<i,a>)

u5: axiom
    henceforth [ *at(interface[i].pa,enq[i])
                 <- after(interface[i].pa,enq[i]) ] true

    (* assumption about external behavior *)
u6: assumption
    [ at(interface[i].pa,enq[i](j,a))
      -> at(interface[i].pa,enq[i](j,a)) ]
      b~=a implies not *at(interface[i].pa,enq[i](j,b))

end unreliablequeue
```

It is now possible to make use of the specification of the reliable queue to define the service provided by the Alternating Bit protocol.

```
datatypes: module
    exporting id,data,message,packet
    theory
        id: type
        data: type
        packettype: type
        message, ack: packettype
        message: type is tuple[id,data]
        packet: type is tuple[packettype,data,boolean]
end datatypes

abservice: distributed module
    using reliablequeue, datatypes
    exporting serviceinterface

theory
    serviceinterface: module[ i: id ]
        exporting pa, pb, send, rec
    theory
        pa, pb: external process
        send: operation[ id, data -> ]
        rec:  operation[ -> message ]
    end serviceinterface

    rq: module is new reliablequeue[id,message]
    j: var id

    i1: linkage  serviceinterface[j].pa = rq.interface[j].pa
                 and serviceinterface[j].pb = rq.interface[j].pb

    i2: linkage  serviceinterface[j].send = rq.interface[j].enq

    i3: linkage  serviceinterface[j].rec = rq.interface[j].dq

end abservice
```

The service interface is provided by an inner module `serviceinterface` that is parameterized by the identity of the user to whom service is provided. The semantics of `serviceinterface` are provided by linking it to a copy of `reliablequeue` for which semantics have already been defined. A private copy of `reliablequeue`, indicated by the keyword `new` is necessary, since we intend only to equate the operation `send` to `enq` and not to some

other operation that might have been elsewhere also defined in terms of `reliablequeue`.

Similarly, we define the service specification of the transmission medium using the specification of the unreliable queue.

```
transmissionservice: distributed module[ packettype: type ]
    using unreliablequeue, datatypes
    exporting transmissioninterface

theory
    transmissioninterface: module[ i: id ]
        exporting pa, pb, send, rec
    theory
        pa, pb: external process
        send: operation[ id, packettype -> ]
        rec:  operation[ -> tuple[id,packettype] ]
    end transmissioninterface

    unrq: module is new unreliablequeue[id,packettype]
    j: var id

    i1: linkage  transmissioninterface[j].pa = unrq[j].pa
                 and transmissioninterface[j].pb = unrq[j].pb

    i2: linkage  transmissioninterface[j].transmit = unrq[j].enq

    i3: linkage  transmissioninterface[j].receive = unrq[j].dq

end transmissionservice
```

6.3 The Protocol Specification

We must now define the protocol itself, in the form of local specifications for the Sender and Receiver modules. Consider what requirements one would like to impose on the visible behavior of the Sender process as part of a protocol standard. We will assume the following requirements are desired:

1. Successive messages must be transmitted in packets having alternating sequence numbers.
2. The sequence of distinct packets transmitted must follow the sequence of messages dequeued.
3. Having initiated transmission of a packet containing a new message, only that message may be transmitted until the first uncorrupted acknowledgment with the transmitted sequence number is received.
4. Having initiated transmission of a message, continued retransmission must occur at least until an acknowledgment is received.
5. If acknowledgments for the last transmitted packet are repeatedly received, they must lead to a call to dequeue another message. Any finite number of acknowledgments may be ignored.
6. No packet may be transmitted during a dequeue. (By (5), the acknowledgment for the last packet must have been noted, prior to the call of dequeue, with the next message not yet available.)

The requirements we assume for the visible behavior of the Receiver process are complicated by the need to handle simultaneously messages from many sources. In effect, messages from distinct sources are handled independently, as follows:

1. Until the next packet from a source is received, acknowledgments may be transmitted to that source only for the last packet received from it.

2. If packets are received repeatedly, they must eventually be acknowledged. Any finite number of packets may be ignored.
3. In accordance with the Sender requirement that successive messages be transmitted in packets with alternating sequence numbers, the Receiver can deliver successive messages from a source only from packets with alternating sequence numbers.
4. Only messages from received packets are allowed to be delivered.
5. The message contained in a packet must be delivered before a packet from the same source with a different sequence number can be acknowledged. Note that this allows the Receiver process to store the packets temporarily, since the delivery can occur after the reception of a new packet.
6. Having initiated acknowledgment of a packet, the contained message must eventually be delivered.

In the specification for the Sender module, the first axiom, s1, states that the module will try to dequeue a message, that no transmissions occur before the first dequeue and that, at the time of the first dequeue, the value of the expected sequence number has been set to an initial value.

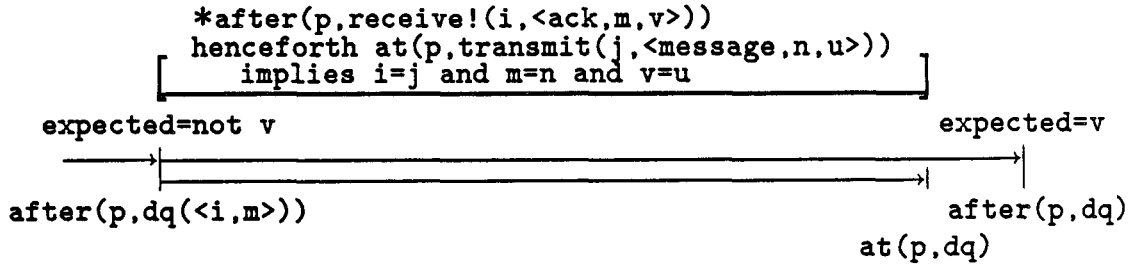
Rather than use interval expressions to establish temporally the alternation of outgoing sequence numbers, we introduce state component `expected`, indicating the expected sequence number. This simplifies our temporal expressions while not overly constraining implementation strategy. Note that the value of `expected` is specified only at the time of returns from Dq.

```
sender: module
    using datatypes
    exporting  p, dq, transmit, receive

theory
    p: process
    dq: external operation[ -> message ]
    transmit: external operation[ id, packet -> ]
    receive:  external operation[ -> tuple[id,packet] ]

    i, j: var id
    m, n: var data
    u, v: var boolean

    initial:  boolean
    expected: state[boolean]

s1: axiom
    [ -> *at(p,dq) ] not *at(p,transmit)
    and  [ after(p,dq) -> ] expected = initial

s2: axiom
    [ after(p,dq!<i,m>) -> ]
    expected = not v
    implies  [ after(p,dq) -> ] expected = v
             and [ -> at(p,dq) ] *after(p,receive!(i,<ack,m,v>))
                                 and henceforth
                                       at(p,transmit(j,<message,n,u>))
                                       implies i=j and m=n and u=v

s3: axiom
    [ after(p,dq!<i,m> -> ]
    expected = not v
    implies  ( henceforth *after(receive(i,<ack,m,v>))
               implies *at(p,dq) )
             and ( not *at(p,dq)
                   implies henceforth *at(p,transmit(i,<message,m,v>)))

s4: axiom
    henceforth not ( in(p,dq) and in(p,transmit(i,<message,m,v>)) )

end sender
```

The three clauses in axiom s2 express the basic safety requirements on the Sender. In clause order, they are:

- After returning from dequeuing a message `m` with the currently expected sequence number `not v`, the expected sequence number will be `v` at the time of the next dequeue.
- At least one uncorrupted acknowledgment with the expected sequence number `not not v` must be received from the destination of the message before the next message can be dequeued.
- Until the next message is dequeued, only `<message, m, v>` packets may be transmitted, and only to the destination of the message.

Graphically:



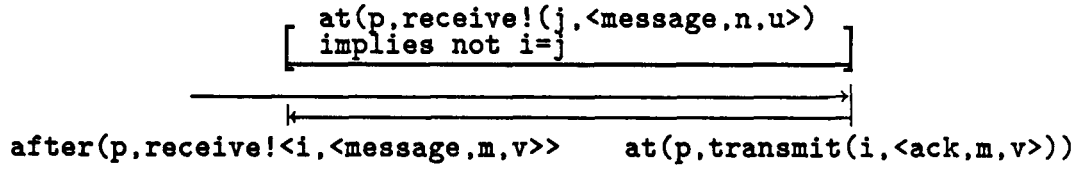
The two clauses of axiom s3 express Sender liveness requirements. After returning from dequeuing a message `m`, with current sequence number `not v`, repeated acknowledgments for sequence number `v` from the destination of the message must lead to a request for another message from the queue. Furthermore, that the Sender *never* attempts to dequeue another message implies continual retransmission of the current packet `<message, m, v>` to the destination of the message.

Axiom s4 expresses a further safety requirement: while the Sender is dequeuing another message, no message packet can be transmitted.

Turning now to the specification of the Receiver module, we again introduce a state component `expected`, now a function of the source of the

message, defining the current sequence number only at the time of a call on Enq. Axiom r1 states that the receiver must attempt to receive packets and that, until receipt of an initial packet, there will be no prior delivery of messages nor transmission of acknowledgments, while axiom r2 requires that, for every message received, transmission of an acknowledgment for that message implies that the message will be delivered.

Axiom r3 expresses a safety property about acknowledgments: acknowledgments will be sent to another destination only for the packet most recently received from that destination

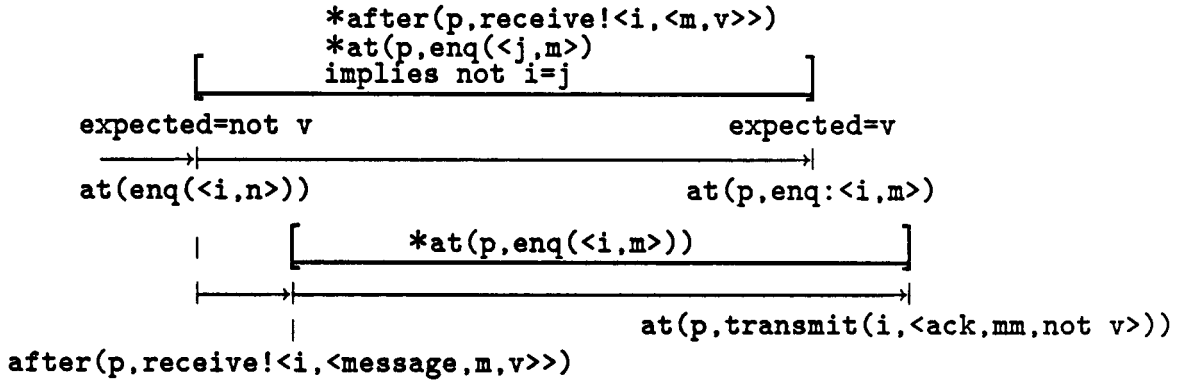


Axiom r4 expresses a liveness property about acknowledgments: If packets are received continually, they must eventually be acknowledged.

```
receiver: module
    using datatypes
    exporting  p, dq, transmit, receive

theory
    p: process
    enq: external operation[ tuple[id,data] -> ]
    transmit: external operation[ id, tuple[data,boolean] -> ]
    receive:  external operation[ -> tuple[ id, tuple[data,boolean] ] ]

    i, j: var id
    m, mm, n: var data
    u, v, w: var boolean

    expected: state[function[id->boolean]]

r1: axiom
    [ -> ( *at(p,receive) -> after(p,receive!<i,<message, m,v>>) ) ]
    not *at(p,transmit(i,<ack,m,v>)
 and
    [ -> after(p,receive!<i,<message,m,not v>>)
         or after(p,receive!<i,<message,m,v>>) ] not *at(p,enq(<i,m>))


r2: axiom
    [ after(p,receive!<i,<message,m,v>>) -> ]
    *at(p,transmit(i,<ack,m,v>) implies *at(p,enq(<i,m>))

r3: axiom
    [ *after(p,receive!<i,<message,m,v>>) <- at(p,transmit(i,<ack,m,v>) ]
    *after(p,receive!<j,<message,n,u>>) implies not i=j

r4: axiom
    henceforth *after(p,receive!<i,<message,m,v>>)
    implies  *at(p,transmit(i,<ack,m,v>))

r5: axiom
    [ at(p,enq(<i,n>)) and expected = not v -> ]
     [ -> at(p,enq:(<i,m>)) ] *after(p,receive!<i,<message,m,v>>)
     and [ after(p,receive!<i,<message,m,v>>)
           -> at(p,transmit(i,<ack,mm,not v>) ] *at(p,enq(<i,n>))
     and [ -> before at(p,enq(<i,m>)) ] *at(p,enq(<j,mm>) implies not i=j
         implies  [ at(p,enq(<i,m>)) -> ] expected = v

end receiver
```

Axiom r5 expresses safety properties related to message receipt. The interval logic formula combines these requirements in order to exhibit their dependence on a common context. In clause order, their contribution is as follows.

- Delivery of a message must be preceded by its receipt.
- Having received a packet, the contained message must be delivered before an acknowledgment for a packet from the same source with a different sequence number is transmitted.
- Delivery of successive messages from the same source must result from packets with alternating sequence numbers v and not v.



We can now put the preceding modules together to form the full implementation specification for the Alternating Bit Protocol. The module `abprotocol` is defined to be a distributed module, since it provides service at many sites. In contrast, each user interface is provided by an instance of the generic module `abinterface` which should be local since it provides service at only a single site.

The `interface` module is built from copies of a simple queue `qa` and the `sender` module to provide the transmission function, and from copies of the `receiver` module and a simple queue `qb` for reception. Linkages make the required connections between their processes and interfaces.

Communication between interface modules is provided by a copy of the `transmissionservice` module which, of course, is a distributed module. Again, linkages make the required connections.

```
abprotocol: distributed module
using datatypes, simplequeue, sender, receiver, transmissionservice
exporting interface

theory

    ts: distributed module is new transmissionservice[packet]

    abinterface: local module[ i: id ]
        using datatypes, simplequeue, sender, receiver, transmissionservice
        exporting pa, pb, send, rec,
                  s.transmit, s.receive, r.transmit, r.receive, s.p, r.p
    theory
        pa, pb: external process
        send: operation[ id, data -> ]
        rec: operation[ -> message ]

        qa: module is new simplequeue[message]
        qb: module is new simplequeue[message]
        s: module is new sender
        r: module is new receiver

        i: var id   d: var data

    i1: linkage  qa.pa = abinterface.pa  and  qa.pb = s.p
                 and  qb.pa = r.p  and  qb.pb = abinterface.pb

    i2: linkage  abinterface.send(i,d) = qa.enq(<i,d>)
                 and abinterface.rec = qb.dq

    i3: linkage  qa.dq = sender.dq  and  receiver.enq = qb.enq

    end abinterface

    j: var id

i4: linkage  s[j].p = ts.transmissioninterface[j].pa
             and  r[j].p = ts.transmissioninterface[j].pb

i5: linkage  s[j].transmit = ts.transmissioninterface[j].send
             and  s[j].receive = ts.transmissioninterface[j].rec

i6: linkage  r[j].transmit = ts.transmissioninterface[j].send
             and  r[j].receive = ts.transmissioninterface[j].rec

end abprotocol
```

6.4 Conclusions

This report has described several extensions to Interval Logic and to the Revised Special specification language that allow them to specify real-time distributed systems.

As demonstrated by the protocol example, the hierarchical composition mechanisms appear to work quite well. However, no hierarchical proof has been performed and such a proof is necessary to demonstrate that it is possible to maintain separation between levels even during the proof.

The introduction of processes into the interval logic is adequate. The mechanism does what is necessary but is not really very elegant. It would have been preferable to be able to rely more on context to define the processes implied by the specifications, but it is unlikely that any purely lexical mechanism will suffice to deduce the intent of the designers unless the processes are cited explicitly.

The extensions to Revised Special appear to be relatively clean and easy to implement. It would be preferable to use the special symbols $\Box$ and $\Diamond$ instead of `henceforth` and `eventually`. But, even on equipment that is capable of displaying the special symbols, their presence in the text of specifications would surely cause great difficulty with the editor.

The extensions necessary to introduce real time constraints into the interval logic specifications appear to work well. The extension is clean and does not complicate the deduction process.

Quite a lot of work, much of it software engineering, is still required to provide a version of Enhanced HDM that can reason with interval logic about distributed systems. Most of that work lies in the area of the decision process and in the area of the skolemization procedures that instantiate existentially quantified formulae and reduce them to the unquantified ground formulae that the decision procedures can solve. In particular, the problem of existential instantiation in quantified temporal logic formulae is still unsolved. For specifications in the first order predicate calculus, the

instantiation for an existentially variable can be regarded as a constant, and it is not difficult for the user to devise a suitable constant. Unfortunately, in a temporal logic formula, the appropriate instantiation is a function of its temporal context and may be hard for the user to define. We are still considering this problem.

Bibliography

(1) R. L. Schwartz and P. M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards," *IEEE Transactions on Communications*, Vol. COM-30, No. 12, December 1982, pp. 2486-2496.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-181804		

4. Title and Subtitle	5. Report Date
Specifying Real-Time Systems with Interval Logic	April 1989
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
John Rushby	
	10. Work Unit No. 505-66-21-01

9. Performing Organization Name and Address	11. Contract or Grant No.
SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	NAS1-17067
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor: Ricky W. Butler
Task 4 Final Report

16. Abstract

Pure temporal logic makes no reference to time. In the first part of this report we describe an interval temporal logic and an extension to that logic which includes real time constraints. We demonstrate the application of this logic by giving a specification for the well-known "lift" (elevator) example. In the second part of this report we show how interval logic can be extended to include a notion of "process" and we describe how the specification language and verification environment of EHDM could be enhanced to support this logic. We conclude by giving a specification of the alternating bit protocol in this extended version of the specification language of EHDM.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Verification Specification Temporal Logic Concurrency	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	87	

NASA FORM 1626 OCT 86