NASA Contractor Report 172262

NASA-CR-172262
19840008753

An Interval Logic for Higher-Level Temporal Reasoning

R. L. Schwartz
P. M. Melliar-Smith
F. H. Vogt
D. A. Plaisted

SRI International
Menlo Park, California 94025

Contract NAS1-17067

September 1983

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

NASA Contractor Report 172262

An Interval Logic for Higher-Level Temporal Reasoning

R. L. Schwartz
P. M. Melliar-Smith
F. H. Vogt
D. A. Plaisted

SRI International
Menlo Park, California 94025

Contract NAS1-17067

September 1983

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

Contents

Page

Chapter 1. Introduction. 2

Chapter 2. An Interval Logic 5
2.1 The Interval Operators ⇒ and ⇐. 7
2.2 Parameterized Operations. 12

Chapter 3. A Formal Model. 14

Chapter 4. A Sampling of Valid Formulas. 18

Chapter 5. Queue Specifications 22

Chapter 6. A Self-Timed Systems Specification 25
6.1 Request-Acknowledgment Protocol 25
6.2 Arbiter 28

Chapter 7. Protocol Specification 30
7.1 Introduction 30
7.2 The AB Protocol Used for Illustration 31
7.3 Specification of the Operations 33
7.4 Specification of the Service Used and the Service Provided 33
7.5 AB Protocol Specification 34

Chapter 8. A Specification of Distributed Mutual Exclusion. 40

Chapter 9. Analysis and Conclusions 44

Acknowledgments 48

References 49

Appendix A.
Reduction of Formulas Containing * Modifier 51

Appendix B.
A Decision Procedure for Combinations of Propositional Temporal Logic and Other Specialized Theories 53

1 Introduction 54
2 Extralogical Variables 56
3 The Tableau Method 57
4 Algorithm A 58
5 Algorithm B 58
6 Implementation 62
7 Extensions 62
8 Acknowledgements 63
9 References 64

Appendix C.
A Low Level Language for Obtaining Decision Procedures for Classes of Temporal Logics 65

1 Introduction 65
1.1 Sets of Computations 65
2 Syntax 66
3 Semantics 67
3.1 Restrictions on the Quantifiers 69
4 A Decision Procedure 70
4.1 Graph Construction 71
4.2 Semantics of Graphs 75
4.3 Example 75
4.4 Iteration Method 77
4.5 Complexity 77
5 Interval Logic 78
6 A PSPACE Sublanguage 79

7 Other Temporal Logics 79
7.1 Branching Time Syntax 79
7.2 Branching Time Semantics 80
7.3 Regular Expressions 81
8 Executable Specifications 82
9 Acknowledgements 82
10 References 82

Chapter 1

Introduction

Under NASA Contract NAS1-15528, SRI developed techniques for the formal specification and verification of reliable operating systems for flight control applications. Such operating systems necessarily involve parallel processing activities, both within each processor and between multiple processors. The techniques available for the specification and verification of such parallel activities were very crude and expensive to use. This report describes a new technique for the specification of asynchronous parallel activities.

In previous research, supported by NSF, SRI explored temporal logic as a framework for specifying and reasoning about concurrent programs, distributed systems, and communications protocols. Previous papers[Schwartz/Melliar-Smith 81,82, Vogt82a,b] report on efforts to use temporal reasoning primitives to express very high-level abstract requirements that a program or system must satisfy. Based on experience with those primitives, SRI has developed an interval logic more suitable for expressing higher-level temporal properties.

---

*This research has received additional support from National Science Foundation Grant MCS-8104459.

†Dr. Vogt was on leave from the Hahn-Meitner-Institut, Berlin, Federal Republic of Germany.

The survey paper[Schwartz/Melliar-Smith82] examines how several different temporal logic approaches express conceptual requirements for a simple protocol. The conclusions were both disappointing and encouraging. On one hand, the very abstract temporal requirements provided an elegant statement of minimal behavior for implementation conformance. It was possible to distill a set of requirements expressing the essence of the desired behavior; stating only requirements without implementation-constraining expedients. Our intention was to specify only the minimum required externally visible behavior, leaving all other aspects to lower levels of description. We have argued that only by doing so can one gain the necessary measure of confidence that a specification reflects the intuitive requirements. Implementation-oriented details, while facilitating verification of *like* implementations, lead to overly detailed and complicated specifications and bias implementation strategies.

While the level of *conceptualization* of the specifications was satisfactory, their expression in temporal logic was rather complex and difficult to understand. Because of the relatively low level of the linear-time temporal logic operators ($\Box$, $\Diamond$, **Until**, **Latches-Until**, etc.), many higher-level concepts had to be "encoded". To characterize these intervals and any desired properties in temporal logic becomes quite difficult and unwieldy. Intervals in temporal logic are "tail sequence" intervals, always extending from the present state through the remainder of the computation. Temporal logic operators are always interpreted on the *entire* tail sequence. For this reason, unary $\Box$ and $\Diamond$ operators cannot be used to specify invariance and eventuality properties in bounded intervals. The **Until** operator, which does allow one to identify a future point in the computation, must be composed to encode indirectly such properties. This quickly leads to a morass of embedded **Until** formulas.

The impoverished set of temporal abstractions forced the inclusion of state components that were not properly part of the specification. These additional state components were needed to establish the amount of context necessary to express the requirements. Without these components, context could only have been achieved by complex nestings of temporal **Until** constructs to establish a

sequence of prior states. The survey paper highlighted how the introduction of state simplifies the temporal logic formulas at the expense of increasing the amount of "mechanism" in the specification.

For our goal of minimal specification of internal behavior, the parameterized event-sequence temporal specification was the most satisfying, and least readable. The difficulty of establishing context by temporal constraint rather than by state function led us to include supplementary state and a slightly lower-level specification.

In this research, we have investigated an interval logic to provide a higher-level framework for expressing temporal relationships. A higher-level temporal concept that pervades almost all temporal specifications is that of a property being true for an interval. The concept of intervals and interval composition forms the basic structure of our specification and verification method. This allows conceptual requirements to be stated rather directly and intuitively within the logic. For the examples considered, this new logic has provided concise and workable specifications of the intended semantic requirements.

An informal introduction of the language and logic follows in Section 2. A formal model for the interval logic is given in Section 3, with a selection of valid formulas appearing in Section 4. The remainder of the paper contains sample specifications and a small proof example. Section 5,6,7, and 8 explore the application of interval logic to queues, a hardware arbiter, a simple communications protocol, and a distributed mutual-exclusion algorithm, respectively. Section 9 concludes with a discussion of the current status of the research.

Appended to this report are two papers by David Plaisted, a consultant to this project, describing a decision procedure for the Interval Logic.

Chapter 2

An Interval Logic

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

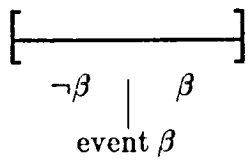
$$[\, I \,][\, J \,]\alpha$$

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. If the interval is found, the $\neg$ inverts the False to True, while if the interval is not found, the interval term is vacuously satisfied and then inverted by the $\neg$ to False. The interval language defines the formula $*I$ to mean exactly this.

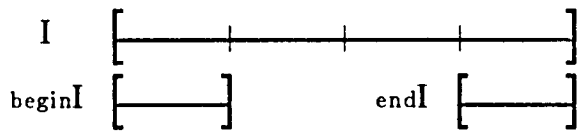
Thus far, we have described how to compose properties of intervals without discussing how intervals are formed. At the heart of a very general mechanism for defining and combining intervals is the notion of an *event.* An event, defined by an interval formula β, occurs when β changes from False to True, 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, begin and end, operate on intervals to extract unit intervals. For interval term I, begin$I$ 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

†at Dq means that control is at the entry point to the operation Dq.



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

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

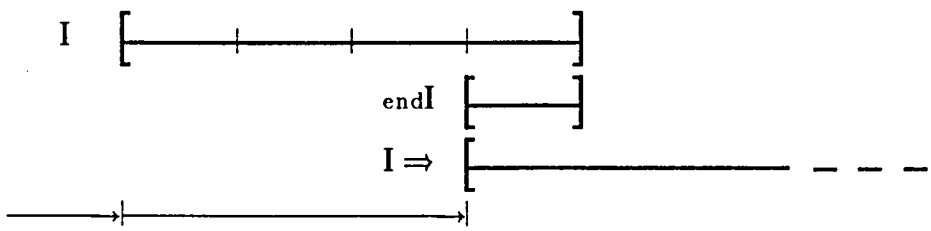
$[\,\text{begin}\,P\,]\neg P$

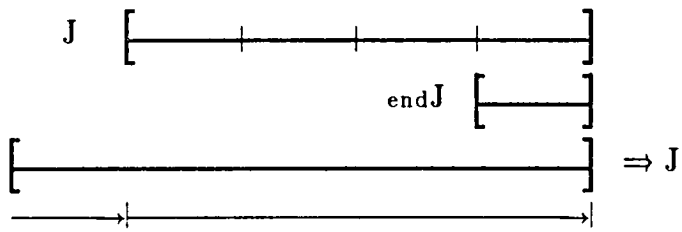
$[\,P\,]\neg P$

2.1 The Interval Operators ⇒ and ⇐

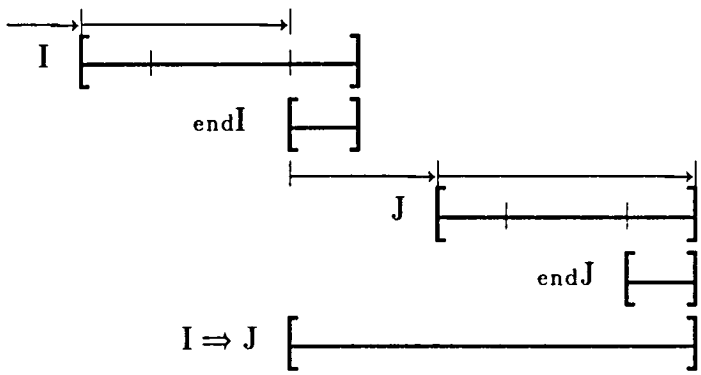
Two generic operators exist to derive intervals from interval arguments. We take the liberty of overloading these operators to allow zero, one or two interval-value arguments. Intuitively, the direction of the operator indicates in which direction and in which order the interval endpoints are located. 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,



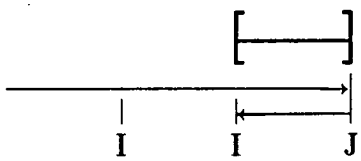


The term $I \Rightarrow J$, with two interval arguments, represents the composition of the two definitions. This constructs the interval starting at the end of interval I and extending to the end of the *next* interval J located in the interval $I \Rightarrow$. Given this definition, the interval formula $[\, I \Rightarrow J \,]\alpha$ 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.

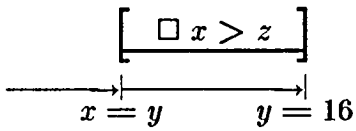
The left arrow operator $\Leftarrow$ is defined analogously. For interval term $I \Leftarrow J$, the first J interval in context is located. From the end of this J interval, the *most recent* I interval is located. The derived interval $I \Leftarrow J$ begins with end$I$ and ends with end$J$. Thus,



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.

$$[\, x = y \;\Rightarrow\; y = 16 \,]\, \Box\; x > z \tag{1}$$



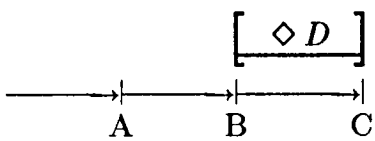
For the interval beginning with the next event of the variable x becoming equal to y and ending with y changing to the value 16, the value of x is asserted to remain greater than z. The first state of the interval is thus the state in which x is equal to y and the last state is that in which y is next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

$$[\, x = y \;\Rightarrow\; \text{begin}(y = 16)\,]\, \Box\; x > z \tag{2}$$

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

$$[\, (A \Rightarrow B) \Rightarrow C \,]\, \Diamond D \tag{3}$$



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

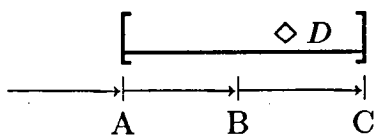
$$[\,(A \Rightarrow\ *B) \Rightarrow C\,]\ \Diamond D \tag{4}$$

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

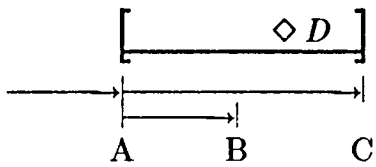
$$[\,A \Rightarrow (B \Rightarrow C)\,]\ \Diamond D \tag{5}$$



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.,

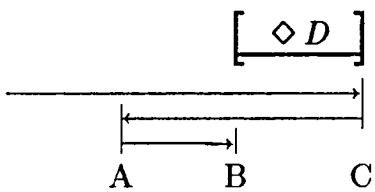
$$[\ \text{begin}(A \Rightarrow B) \Rightarrow C\ ]\ \Diamond D \tag{6}$$



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

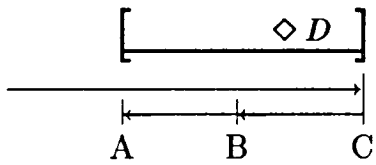
$$[\ (A \Rightarrow B) \Leftarrow C\ ]\ \Diamond D \tag{7}$$



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

$$[\ \text{begin}(A \Leftarrow B) \Leftarrow C\ ]\ \Diamond D \tag{8}$$



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 $\text{at}O$, $\text{in}O$, and $\text{after}O$ 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. $[\, \text{at}O \Rightarrow \text{begin after}O \,] \,\Box\, \text{in}O$
2. $[\, \text{after}O \Rightarrow \text{begin at}O \,] \,\Box\, \neg\text{in}O$
3. $[\, \neg\text{at}O \Rightarrow \text{after}O \,] \,\Box\, \neg\text{at}O$
4. $[\, \neg\text{after}O \Rightarrow \text{at}O \,] \,\Box\, \neg\text{after}O$

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

$$[\,\mathrm{at}O \Rightarrow\ *\,\mathrm{after}O\,]\mathrm{True}$$

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. $\mathrm{at}O(v_1, \ldots, v_n)$ is true in any state in which $\mathrm{at}O$ 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

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

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

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

Chapter 3

A Formal Model

In this section we give the syntax and model-theoretic semantics for the language of interval logic.

In the following, we will use α, β, γ as logical variables ranging over interval formulas and use I, J, K ranging over interval terms. We use P to range over atomic predicates and A to range over event terms.

Summarizing the language of our logic, we have defined the following syntactic constructs:

<interval formula> α ::

$P \mid \neg\beta \mid \beta$ <propositional connective> $\gamma \mid$

$\Diamond\beta \mid \Box\beta \mid *I \mid [\, I \,]\beta$

<interval term> I ::

$A \mid \text{begin}J \mid \text{end}J \mid$

$J \Rightarrow K$ (*with possible omission of one or both arguments*) |

$J \Leftarrow K$ (*with possible omission of one or both arguments*)

<event term> $A :: \alpha$

As we mentioned earlier, the $*$ interval term modifier is considered as a syntactic abbreviation. Rules for its elimination appear in the Appendix.

For a finite or infinite computation state sequence s, we now define satisfaction of an interval formula α by s. In defining the model, we use the notation $s_{<i,j>}$ to denote the subsequence of s beginning with the i^{th} element of the sequence, and ending with the j^{th} element of the sequence. As a representation for an infinite sequence, we use ∞ as the right endpoint value, as in the subsequence $s_{<i,\infty>}$. For a finite computation, we extend the last state to form an infinite sequence.

The following model defines, for sequence s and interval formula α, the satisfaction relation $s_{<i,j>} \models \alpha$. We say that a sequence s satisfies formula α if $s_{<1,|s|>} \models \alpha$. Since our definition of the satisfaction relation will always be referring to portions of the same s sequence, we will refer to s using only its subsequence denotation, i.e., as $< i, j > \models \alpha$.

The relation $< i, j > \models \alpha$ is defined recursively, based on the structure of the formula, as follows:

$< i, j > \models P \quad \equiv \quad s_i \models P$ (i.e. P is true of the first state of the interval.)

$< i, j > \models \neg\alpha \quad \equiv \quad$ not $< i, j > \models \alpha$

$< i, j > \models \alpha \wedge \beta \quad \equiv \quad < i, j > \models \alpha$ and $< i, j > \models \beta$

$< i, j > \models \Box\, \alpha \quad \equiv \quad \forall k \in < i, j > \; < k, j > \models \alpha$

$< i, j > \models \Diamond\, \alpha \quad \equiv \quad \exists k \in < i, j > \; < k, j > \models \alpha$

$< i, j > \models [\, I \,]\alpha \quad \equiv \quad \mathcal{F}(I, < i, j >, \mathrm{F}) \models \alpha$

$\perp \models \alpha$

The $\mathcal{F}$ function appearing in the definition of $[\ I\ ]\,\alpha$ is a interval-valued

function from an interval term, an interval, and a direction of search. The direction of search is denoted by F for forward or B for backward – logical variable d ranges over F and B. The function $\mathcal{F}$ denotes the interval I found in the $< i, j >$ context looking in the direction of search. The function is defined to return the null interval value $\perp$ when the interval cannot be constructed. All functions on intervals are strict on $\perp$. By the last clause in the above definition, any formula α is satisfied for such a null interval. This serves as a device to define our partial correctness semantics for interval formulas.

For event term α and interval $< i, j >$ we define

$$\text{changeset}(\alpha, < i, j >) = \left\{ \begin{array}{l} < k-1, k > \mid k \in < i+1, j > \\ \qquad\qquad \wedge < k-1, j > \models \neg\alpha \\ \qquad\qquad \wedge < k, j > \models \alpha \end{array} \right\}$$

to define the set of events α occuring in the interval, each event being the interval of change $< k-1, k >$ in which α changes from false to true. With this we next define

$$\mathcal{F}(\alpha, < i, j >, \text{F}) = \min(\text{changeset}(\alpha, < i, j >)$$

$$\mathcal{F}(\alpha, < i, j >, \text{B}) = \max(\text{changeset}(\alpha, < i, j >)$$

We assume min and max functions on sets of (interval-valued) pairs are defined in the standard manner (the represented intervals are disjoint). Both min and max return $\perp$ if the set is empty, and max returns $\perp$ for an infinite set. Thus $\mathcal{F}$ returns the interval of change for the first or last event α in the interval $< i, j >$, and returns $\perp$ if that interval cannot be found.

Next we define the interpretation of the interval functions begin and end

$$\mathcal{F}(\text{begin}I, < i, j >, d) = < \text{first}(\mathcal{F}(I, < i, j >, d)), \text{first}(\mathcal{F}(I, < i, j >, d)) >$$

$$\mathcal{F}(\text{end}I, < i, j >, d) = < \text{last}(\mathcal{F}(I, < i, j >, d)), \text{last}(\mathcal{F}(I, < i, j >, d)) >$$

where $\text{first}(< i, j >) = i$, $\text{last}(< i, j >) = j$
and $\text{last}(< i, \infty >)$ is defined to return $\perp$.

We now define our forward and backward interval construction functions through a recursive interpretation for $\mathcal{F}$ based on the structure of the interval-term argument.

$$\mathcal{F}(\Rightarrow, < i,j >, d) \quad = \quad \mathcal{F}(\Leftarrow, < i,j >, d) \quad = \quad < i,j >$$

$$\mathcal{F}(I\Rightarrow, < i,j >, d) \quad = \quad < \text{last}(\mathcal{F}(I, < i,j >, d)),\ j >$$

$$\mathcal{F}(I\Leftarrow, < i,j >, d) \quad = \quad < \text{last}(\mathcal{F}(I, < i,j >, \text{B})),\ j >$$

$$\mathcal{F}(\Rightarrow J, < i,j >, d) \quad = \quad < i,\ \text{last}(\mathcal{F}(J, < i,j >, \text{F})) >$$

$$\mathcal{F}(\Leftarrow J, < i,j >, d) \quad = \quad < i,\ \text{last}(\mathcal{F}(J, < i,j >, d)) >$$

We now derive the semantics of the two argument arrow operators as the composition of those above.

$$\mathcal{F}(I\Rightarrow J, < i,j >, d) \quad = \quad \mathcal{F}(\Rightarrow J,\ \mathcal{F}(I\Rightarrow, < i,j >, d), \text{F})$$

$$\mathcal{F}(I\Leftarrow J, < i,j >, d) \quad = \quad \mathcal{F}(I\Leftarrow,\ \mathcal{F}(\Leftarrow J, < i,j >, d), \text{F})$$

This completes our model for interval logic formulas.

Interval logic specifications are divided into two parts: Init and Axioms. An *Init* portion states properties to be satisifed at (from) the beginning of a computation, assuming a distinguished starting state. Formally, using distinguished (uninterpreted) state predicate *start*, each interval formula α within the Init clause is interpreted as an axiom of the form $start \supset \alpha$. The interpretation of *start* is a a methodological concern: the predicate will be mapped to the beginning state of the computation sequence when proving that a program satisfies the specification. The assumption of a distinguished starting state will allow us to more completely characterize correct system or program behavior.

Chapter 4

A Sampling of Valid Formulas

In this section we present a selection of valid formulas. Our intention here is simply to illustrate a style of expression and deduction rather than a more comprehensive list of valid formulas or a complete axiomatization. We are currently incorporating a decision procedure for interval logic[Plaisted83] into our STP deduction system[Shostak/Schwartz/Melliar-Smith82]. We are therefore more concerned about the *style of expression* than an axiomatization of the language or rules of deduction.

As in the previous section, we use α, β, γ as logical variables ranging over interval formulas, and I, J, K ranging over interval terms. Additionally, we use variables ρ, σ to range over state predicates (not containing any temporal operators).

Interval formulas distribute across intervals, as indicated by the following formulas.

V1. $[\,I\,]\alpha \;\wedge\; [\,I\,]\beta \;\equiv\; [\,I\,](\alpha \wedge \beta)$

V2. $[\,I\,]\alpha \;\supset\; [\,I\,]\beta \;\equiv\; [\,I\,](\alpha \supset \beta)$

Expressing the fundamental case split in interpreting interval formulas, we have

V3. $[\, I \,]\alpha \quad \equiv \quad \neg *I \quad \vee \quad [\, *I \,]\alpha$

defining the formula to be true if either the interval cannot be constructed, or if α holds for the constructed interval. Associated with this, we also have

V4. $*I \quad \equiv \quad \neg[\, I \,]\mathrm{False}$

V5. $*\alpha \quad \equiv \quad \Diamond(\neg\alpha \wedge \Diamond\, \alpha)$

V6. $\neg[\, I \,]\alpha \quad \equiv \quad [\, *I \,]\neg\alpha$

Formula V4 derives the meaning of our interval-eventuality operator in terms of an interval formula, while V5 re-expresses this in terms of nested $\Diamond$ eventuality. Formula V6 defines "pushing" interval formula negation into the interval.

For an arbitrary interval α, we have the following formulas illustrating the "promotion" of noninterval properties to interval properties.

V7. $\alpha \quad \equiv \quad [\Rightarrow]\alpha$

V8. $\Box\, \alpha \quad \supset \quad \Box[\, I \Rightarrow \,]\alpha$

Formula V7 expresses the fact that the interval ($\Rightarrow$) selects the complete outer context, while V8 expresses the fact that any invariant α of the outer context will apply in any "tail interval" of the context. A consequence of our basic definition of event terms is

V9. $[\, \alpha \Rightarrow \mathrm{begin}\neg\alpha \,]\, \Box\, \alpha$

That is, for the interval beginning with α becoming true and extending until just prior to α becoming false, α will remain true.

As properties of how intervals are constructed, we have

V10. $[\, \mathrm{begin}\alpha \Rightarrow \,]*\beta \quad \vee \quad [\, \mathrm{begin}\beta \Rightarrow \,]*\alpha$

V11. $[\, \alpha \Leftarrow \beta \,]\gamma \quad \equiv \quad [\Rightarrow \beta\,][\, \neg *\alpha \Rightarrow \,]\gamma$

V12. $[\Rightarrow I\,]\neg\, \Box\, *J$

Formula V10 expresses a fundamental event-ordering property. For two events

designated by α and β, either (1) one or the other event does not occur, (2) α occurs before β, (3) β occurs before α, or (4) both occur at the same time. This case split is often used to prove properties relating multiple events.

For *nonnested* interval terms, formula V11 reduces the semantics of our backward $\Leftarrow$ operator to an equivalent expression using the forward $\Rightarrow$ operator. In doing this reduction, we employ a nested interval event formula. The embedded $(\neg * \alpha)$ thus begins when the $\neg * \alpha$ formula changes to become true. This will becomes true in the first state when one can no longer find another α event – precisely in the first α state of the last change to α. Of course this kind of "tricky encoding" should be avoided; the backward operator was included in the language to provide a higher-level construct to express this!

Formula V12 expresses the fact that no interval with an upper end point, and therefore finite, can contain an unbounded number of J intervals. This follows from the fact that the occurrence of an event requires a change in predicate value – and thus at least two states. Note that the formula $* \Diamond \alpha$ is satisfiable in a bounded interval. This would be satisfied by any interval state sequence in which α is true in the last state. Thus, the interpretation of $\Box \Diamond$ as "infinitely often" only applies over infinite intervals.

As basic properties of interval partitioning, we have

V13. $[\Rightarrow I]\,\Box\rho \quad \wedge \quad [I \Rightarrow]\,\Box\rho \quad \supset \quad \Box\rho$

V14. $\Box\rho \quad \supset \quad [\Rightarrow I]\,\Box\rho \quad \vee \quad [I \Rightarrow]\,\Box\rho$

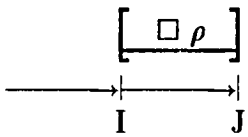
By V13, for any interval term I, if a simple property ρ is true up to I and is true from I onward within the outer context, then ρ is true throughout the context. Typical use of this would be to establish invariance or eventuality properties for an interval by showing the properties to hold for portions of the interval. Formula V14 expresses the dual of this.

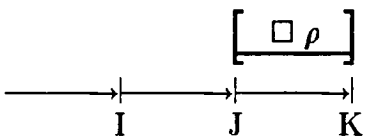
Finally, the following formulas express interval composition.

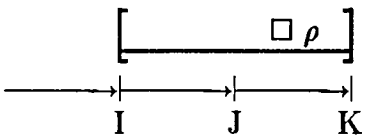
V15. $[\, I \Rightarrow J \,] \Box \rho \quad \wedge \quad [\, (I \Rightarrow J) \Rightarrow K \,] \Box \rho$
$\supset \quad [\, I \Rightarrow (J \Rightarrow K) \,] \Box \rho$

V16. $[\, \Rightarrow (J \Rightarrow K) \,] \alpha \quad \wedge \quad [\, \Rightarrow * J \,] \neg * K$
$\supset \quad [\, \Rightarrow K \,] \alpha$

Formula V15 defines the composition of two intervals $(I \Rightarrow J)$ and $(\,(I \Rightarrow J) \Rightarrow K)$ to form the interval $(I \Rightarrow (J \Rightarrow K))$. Pictorially, we have







A nonembedded interval property $\Box \rho$ is thus derived for the interval from I to the first K that follows the first J by proving it for the associated I to J and J to K intervals. For the case where one can prove that the first K following I also follows J, formula V16 allows the simplification of $(\Leftarrow (J \Rightarrow K))$ to $(\Rightarrow K)$.

Chapter 5

Queue Specifications

In this section, we illustrate two specifications of queues with asynchronous enqueuing and dequeuing operations. We first consider a reliable (normal) queue, followed by an unreliable queue. Our queue has only 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. This avoids a more explicit process-naming convention.

The formula

Queue.

$$[\, \Leftarrow \text{afterDq}(b)\,](*_{\text{afterDq}}(a) \quad \equiv \quad *(\text{atEnq}(a) \Leftarrow \text{atEnq}(b)\,))$$

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.

By exchanging $\text{atEnq}(a)$ and $\text{atEnq}(b)$ terms in the queue axiom above, yielding
Stack.

$$\left[\Leftarrow \text{afterDq}(b) \right] (*\text{afterDq}(a) \quad \equiv \quad *(\text{atEnq}(b) \Leftarrow \text{atEnq}(a)))$$

one obtains a last-in first-out queue (i.e., stack).

In preparation for specifying the services of a communication transmission medium in Section 7, consider a modification to the queue semantics to allow it to be intermittently unreliable. Individual values can be lost from the queue, provided that any value enqueued a sufficient number of times will eventually be available for dequeuing. This specification allows repeated Enq operations for the same value, to permit the value to be reenqueued until it is dequeued. The specification is shown in Figure 5-1.

Init:

I1. $\left[*\left(\text{atEnq}(a) \Rightarrow \text{atEnq}(b)\right) \quad \Leftarrow \quad \left(\text{afterDq}(a) \Rightarrow \text{afterDq}(b)\right) \right] \text{True}$

I2. $\left[\Rightarrow \text{afterDq}(a) \right] *\text{atEnq}(a)$

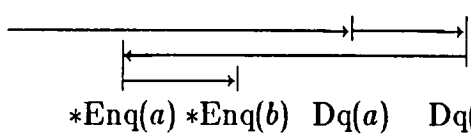
I3. $\left[\text{atEnq}(c) \Rightarrow \text{atEnq}(c) \right] d \neq c \supset \neg *\text{atEnq}(d)$

A1. $\Box *\text{atEnq} \wedge *\text{atDq} \quad \supset \quad *\text{afterDq}$

A2. $\left[\text{atEnq} \Rightarrow \right] *\text{afterEnq}$

Figure 5-1: Specification of an Unreliable Queue, with distinct enqueued items.

Clause I1 requires that, for all a and b, if we dequeue a before dequeuing b then we must have previously enqueued those two items in that same order.



Note that a and b do not have to be successive items; the clause applies to any

pair of items. If the values of either *a* or *b*, or both, are such that the value is never dequeued then it will not be possible to construct the interval between their dequeuings. Note that the clause is vacuously satisfied for any pair of values for which this dequeuing interval cannot be found. Clause I2 contributes the requirement that values must be enqueued prior to being dequeued. These clauses are both predicated on items being dequeued and state that items dequeued must have been enqueued in the same order. These two clauses place no constraints on items lost and thus never dequeued. I3 here expresses the distinct item constraint: repeated Enqs must be consecutive; once some other value is enqueued, it is not permissible to return to any prior value.

Axiom A1 now expresses the weak constraint that infinitely repeated Enqs will ensure that the Dq operation returns. Items can thus be lost from the queue as long as, eventually, an item is retained to be dequeued. Axiom A2 requires only that the Enq operation terminate.

Chapter 6

A Self-Timed Systems Specification

Self-timed logic [Seitz80] was introduced as a means to reduce complexity of asynchronous connections between hardware modules. The method is based on a request-acknowledgment protocol which guarantees that a module remains inactive until it is requested, and that the request remains in place as long as the module is required. The correctness of such systems, if properly constructed, is independent of the speed of its components.

In this section, we use interval logic to describe a simple request-acknowledgment protocol. Based on these specifications, we define an arbiter module (adapted from [Seitz80] and [Bochmann82]), that determines the order in which two user modules obtain access to a shared resource.

6.1 Request-Acknowledgment Protocol

The interaction between self-timed modules takes place by a pair of circuits. One circuit, indicated by "R" carries the *request* from the *requesting* module to the *responding* module (see Figure 6-1). The second circuit indicated by "A" carries the acknowledgments in the opposite direction (from the *responding* module to the *requesting* module).

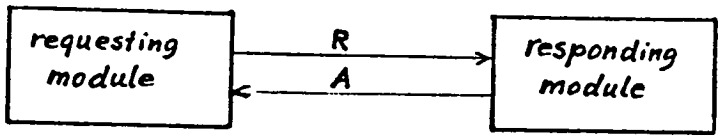
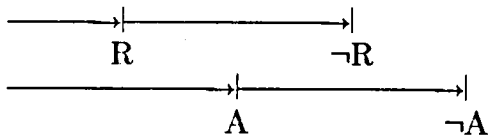


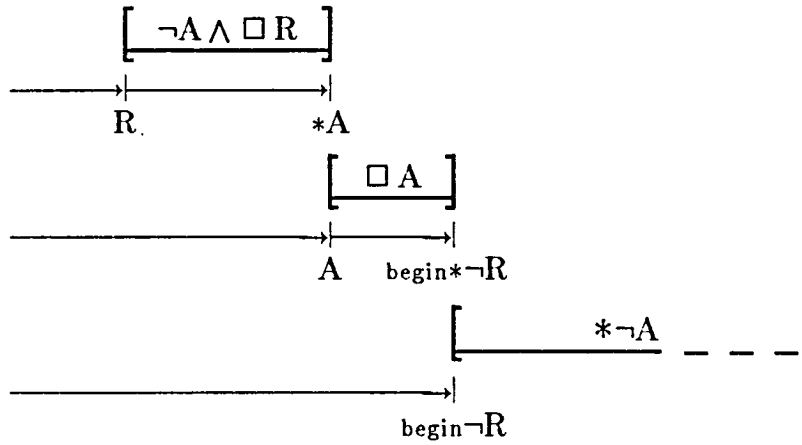
Figure 6-1: Interaction Scheme Between Two Modules

The request-acknowledgment protocol determines how requests and acknowledgements are exchanged between two interacting modules. Using state predicate R to indicate that the request signal is up and A that the acknowledgment signal is up, the following figure illustrates the flow of signals in the request-acknowledgment protocol.



Note that *events* R and A then designate signal raising, while events ¬R and ¬A designate signal lowering.

As the figure indicates, after R is set, an acknowledgment signal must occur before R can become False again. Note the causality between R and A, requiring that the R signal is raised before A. Similarly the acknowledgment signal must be False before a request can be initiated, and the A signal cannot be lowered until the request has ended. A consequence of these requirements is that a "new" request on the same circuit can occur only after the previous acknowledgment has ended. Graphically, these specifications of the order of these signal-changes are:



A precise specification of these properties in interval logic is given in Figure 6-2.

Init. $\neg R \wedge \neg A$

A1. $[\, R \Rightarrow *A \,]\neg A \wedge \Box R$

A2. $[\, A \Rightarrow \text{begin} * \neg R \,]R \wedge \Box A$

A3. $[\, \text{begin}\neg R \Rightarrow \,]*\neg A$

Figure 6-2: Request Acknowledgement Protocol Axioms.

Axiom 1 expresses a *requester* requirement that a request signal, only initiatable when the acknowledgment signal is down, remains up at least until the acknowledgment signal is raised.

For the *responder*, A2 states that the acknowledgment signal, once raised, remains up as long as the request stays up (safety). Axiom A3 requires that, after lowering the request signal, the acknowledgment must also be lowered at some later time.

The initial condition indicates that the axioms are implied from a point at which a request has been reset.

6.2 Arbiter

We now give a specification of an arbiter module. The arbiter, adapted from [Seitz80] and [Bochmann82], determines the order in which two user modules obtain access to a shared resource module. The arbiter *AR* interacts with the user modules *U1/U2*, the transfer modules *T1/T2*, and the resource module *RM* (see Figure 6-3) by the request-acknowledgment protocol described in the previous section.

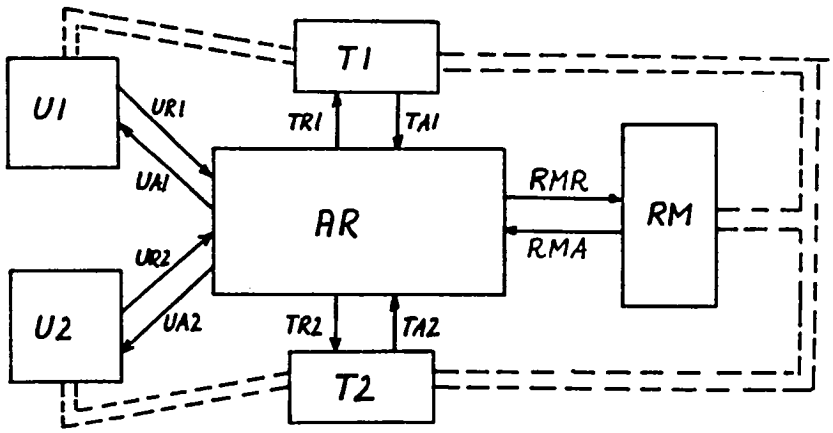
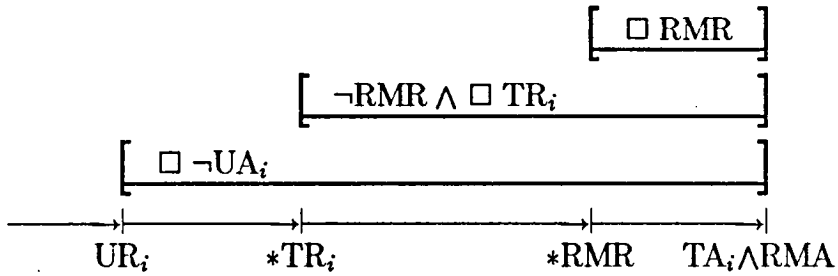


Figure 6-3: The Arbiter Module and its Interacting Modules

Assume that a user module, *U1*, requests access to the resource *RM* by raising UR1. The arbiter grants this access by requesting first the transfer module, *T1*, and then the resource module – provided it is not currently servicing any other user module. Until the arbiter receives acknowledgments from *both* the transfer module and the resource module, it maintains its requests for each of those modules and refrains from sending an acknowledgment to the user. The use of the request-acknowledgment protocol ensures that pairs of requests and acknowledgments be well-behaved – i.e., that both safety and liveness properties expressed in the previous subsection will be obeyed.

The requirements on the signalling order are graphically specified in the

following figure:



Init. $\forall k \neg UR_k$ $k \in \{1,2\}$

A1. $[\, UR_i \Rightarrow TA_i \wedge RMA \,] \Box \neg UA_i$ $i,j \in \{1,2\}$
$\wedge [\, *TR_i \Rightarrow \,] \Box TR_i$
$\wedge \neg RMR$
$\wedge [\, *RMR \Rightarrow \,] \Box RMR$

A2. $TR_i \supset \neg TR_j$ $i \neq j \quad i,j \in \{1,2\}$

Figure 6-4: Arbiter Axioms

A precise specification of the arbiter module in interval logic is given in Figure 6-4.

Axiom 1 establishes three nested intervals, all ending at the first moment at which *both* TA_i and RMA are true. For the outer interval, from UR_i until TA_i and RMA, UA must be False throughout the interval and TR_i must be found. For the contained interval from TR_i, TR_i must remain true throughout the interval, and RMR must be False initially but occur later within the interval. For the inner interval, once RMR becomes true it must remain true.

Similar to the initial condition of the request-acknowledgment protocol, all user request signals must start low.

Chapter 7

Protocol Specification

This section outlines the use of interval logic in the communication protocol and service area. When dealing with a communication system, it is of particular importance to state the conceptual requirements directly and intuitively. A great advantage of the interval-logic approach is its inherent flexibility and the relatively high degree of selectivity it offers when choosing suitable state information. To illustrate the use and the appropriateness of this language the Alternating Bit protocol is selected as an example. 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].)

7.1 Introduction

Protocols are defined as a set of rules that determine the required communication behavior of communicating entities with respect to their functions. Communication services are defined as capabilities of communicating entities at the user's service access points [ISO82].

A service specification defines the services provided by a layer, describing only that behavior visible to the users at the layer above. The protocol specification

refines the service specification in order to define the requirements of each entity supporting the service on one layer through interaction with the service of the next lower layer. (This principle is illustrated in Figure 7-1.)

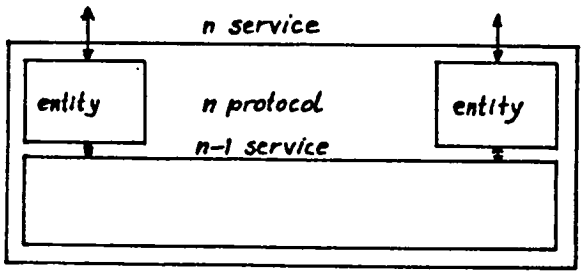


Figure 7-1: Principle of the OSI-Architecture

As such, a protocol standard imposes (or should impose) sufficient constraints to ensure that any implementation that satisfies the standard will uphold continued communication between entities. The standard should also be sufficiently liberal to allow any implementation that would uphold continued communication with other implementations, thus satisfying the standard. Therefore a protocol specification should serve as a formal contract between the overall protocol layer and each distributed component; any component satisfying its local specification should be capable of successfully joining the network.

The objective of the Data Link layer is to detect and possibly correct errors that may occur in the underlying Physical Link layer. For the purpose of this paper, only one direction of data transmission of the Alternating Bit (AB) protocol is considered.

7.2 The AB Protocol Used for Illustration

The AB 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 acknowledgement that its current message has been received correctly. The mes-

sage 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. We consider only the half-duplex protocol providing unidirectional message transfer.

A refinement of the Data Link layer entities for the AB protocol may consist of an input queue and a Sender process as well as a Receiver process and an output queue. The structure of such AB protocol entities as illustrated in Figure 7-2.

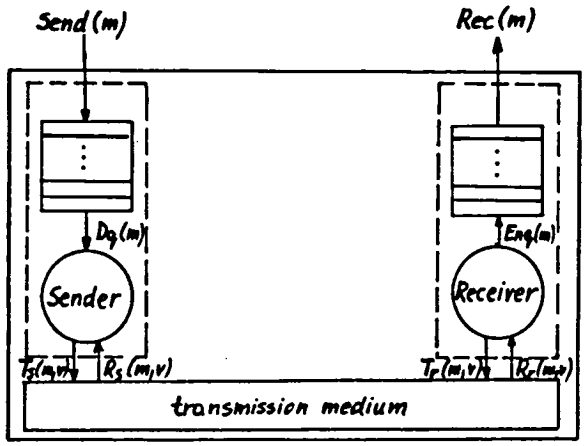


Figure 7-2: Structure of AB protocol entities

The scenario of sending one message can be described by assuming that the Sender entity gets a message m by means of Send(m) from the sending user. It will be placed in the Sender queue. The Sender process dequeues (through Dq(m)) the message and transmits it together with the Sender's current sequence number v as a packet through $T_s(m,v)$. The Receiver entity gets packets by means of $R_r(m,v)$. Acknowledgements are sent from the Receiver entity through $T_r(m,v)$ and received by the Sender entity by means of $R_s(m,v)$. Messages from the Receiver process can be stored by means of Enq(m) in the Receiver queue from which the receiving user can dequeue it by using Rec(m).

7.3 Specification of the Operations

The abstract operations of the Sender process and the Receiver process are:

- Dq(m) to obtain the next message to be sent.
- $T_s(m,v)$ to transmit a packet consisting of a message and a sequence number.
- $R_s(m,v)$ to receive an acknowledgement with a sequence number.
- $R_r(m,v)$ to receive a packet with a message and a sequence number.
- $T_r(m,v)$ to transmit an acknowledgement containing a sequence number.
- Enq(m) to add a message in the Receiver queue.

7.4 Specification of the Service Used and the Service Provided

The service used defines a service of an unreliable medium and therefore subject to loss or corruption of the sent data, but not subject to a reordering of the sequence of submitted packets. It is also assumed that, by repeated retransmission of a packet, it will be delivered uncorrupted at some time. This characteristic is equivalent to the properties of the unreliable queue specified in Section 6. Therefore the specification of the service used consists of the mapping of T_s to Enq and of R_r to Dq , in order to get the unreliable transmission service for the packet transmission. The unreliable transmissions of acknowledgements can be specified by an analogous mapping of T_r and R_s to Enq and Dq, respectively. Two unreliable queues, one for the packet flow and one for the acknowledgement flow, represent the service through which the AB Sender and Receiver processes are communicating with each other.

Similar to the approach taken to specify the unreliable medium, the reliable message exchange between two users in the one-way exchange mode has the same characteristic as the reliable queue. Therefore a similar mapping as above, associating Send with Enq and Rec with Dq , provides the specification for the service provided.

One may also be interested in the service provided by the sublayer consisting only of the two processes (based on the same characteristics of the underlying medium). In this case only the dequeue and enqueue operations have to be considered. It will turn out (with respect to the imposed behavior of the AB Sender process and the AB Receiver process described in the following subsections) that this service is just the service that could be represented by a one element (maximum) queue. For this reason, of course, the AB protocol is not efficient for long delay links.

7.5 AB Protocol Specification

The protocol specification focuses on the Sender process and the Receiver process.

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. Sucessive 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 acknowledegement with the transmitted sequence number is received.
4. Having initiated transmission of a message, continued retransmission must occur at least until an acknowledgement is received.
5. If acknowledgements for the last transmitted packet are repeatedly received, they must lead to a call to dequeue another message. Any finite number of acknowledgements may be ignored.

6. No packet may be transmitted during a dequeue. (By (5), the acknowledgement 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 as follows:

1. Until the next packet is received, acknowledgements may be transmitted only for the last packet received.

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 sucessive messages be transmitted in packets with alternating sequence numbers, the Receiver can deliver successive messages 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 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 acknowledgement of a packet, the contained message must eventually be delivered.

For the Sender process:

Figure 7-3 illustrates the initial property and the three axioms corresponding to the above informal requirements. The initial property *Init* states 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 a distinguished *initial* value.

Init. $[\Rightarrow \text{atDq}] \neg *\text{atT}_s \quad \wedge \quad [* \text{atDq} \Rightarrow] exp = initial$

A1. $[\text{afterDq}(m) \Rightarrow] \quad exp = v \supset [\text{end afterDq}] exp = \overline{v}$
$\wedge [\Rightarrow \text{atDq}] *\text{afterR}_s(m, \overline{v})$
$\wedge \Box [\text{end at T}_s] \text{atT}_s(m, \overline{v})$

A2. $[\text{afterDq}(m) \Rightarrow] \quad exp = v \supset (\Box *\text{afterR}_s(m, \overline{v}) \supset *\text{atDq})$
$\wedge \neg *\text{atDq} \supset \Box *\text{atT}_s(m, \overline{v})$

A3. $\text{inDq} \supset \neg \text{inT}_s$

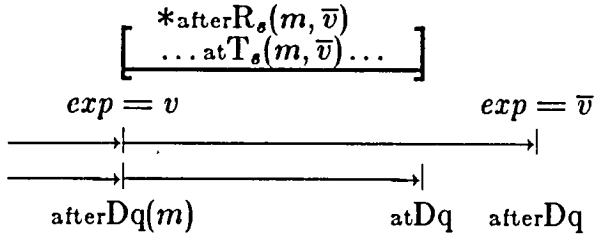
Figure 7-3: Specification of Sender for AB Protocol.

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

The three clauses in Axiom A1 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 v, the expected sequence number will be $\overline{v}$ (i.e., incremented modulo 2) at the time of the next dequeue.
- At least one uncorrupted acknowledgement with the expected sequence number $\overline{v}$ must be received before the next message can be dequeued.
- Until the next message is dequeued, only $< m, \overline{v} >$ packets may be transmitted.

Graphically:



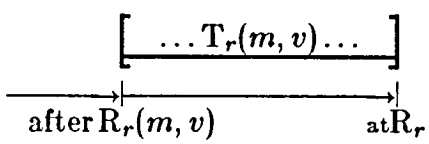
The two clauses of Axiom A2 express Sender liveness requirements. After returning from dequeuing a message m, with current sequence number v, repeated acknowledgents for sequence number $\overline{v}$ must lead to a request for another message from the queue. Furthermore, that the Sender *never* attempts to dequeue another message implies continual retransmission of the current packet $< m, \overline{v} >$.

Axiom A3 expresses a further safety requirement: while the Sender is dequeuing another message, no packet can be transmitted.

For the Receiver process:

Figure 7-4 illustrates the Receiver specification. The initial property is that, until receipt of an initial packet, there will be no prior delivery of messages or transmission of acknowledgements, and from that receipt onward, transmission of the first acknowledgement leads to delivery of the message. Again, we introduce a state component *exp*, defining the current sequence number only at the time of a call on Enq.

Axiom A1 expresses a safety property about acknowledgments: Between receiving a packet $< m, v >$ and the next packet receipt, acknowledgements will be sent only for sequence number v .



Init. $[\Rightarrow \text{atR}_r]\neg *\text{atEnq} \wedge \neg *\text{atT}_r(a,b)$
$\wedge [\text{begin atR}_r \Rightarrow] \quad [\text{end atT}_r]\text{atT}_r(l,w) \quad \supset \quad [* \text{end atEnq}]\text{atEnq}(l)$
$\wedge\ exp = w$

A1. $[\text{afterR}_r(m,v) \Rightarrow \text{afterR}_r] \Box [\text{end atT}_r]\text{atT}_r(m,v)$

A2. $\Box *\text{afterR}_r(m,v) \quad \supset \quad *\text{atT}_r(m,v)$

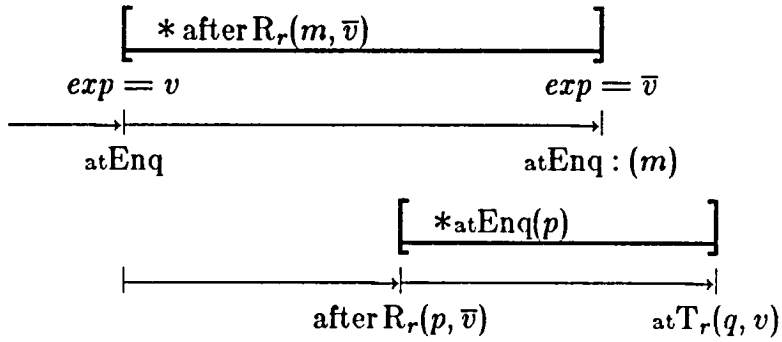
A3. $[\text{atEnq} \Rightarrow] \quad exp = v \supset [\text{atEnq} \Rightarrow] exp = \overline{v}$
$\wedge [\Rightarrow \text{atEnq} : (m)] *\text{afterR}_r(m,\overline{v})$
$\wedge [\text{afterR}_r(p,\overline{v}) \Rightarrow \text{atT}_r(q,v)] *\text{atEnq}(p)$
$\wedge *\text{atT}_r(n,\overline{v}) \supset *\text{atEnq}(n)$

Figure 7-4: Specification of Receiver for AB Protocol.

Axiom A2 expresses a liveness property about acknowledgments: If packets are received continually, they must eventually be acknowledged.

Axiom A3 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.

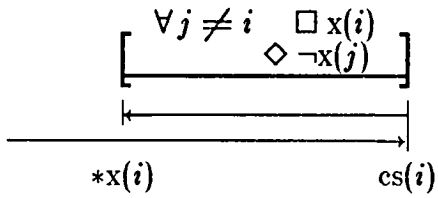
- Delivery of sucessive messages must result from packets with alternating sequence numbers v and $\overline{v}$.
- Delivery of a message must be preceded by its receipt.
- Having received a packet, the contained message must be delivered before an acknowledgement for a packet with a different sequence number is transmitted.
- Acknowledging a packet n must ensure delivery of its message, but that acknowledgement may be transmitted before or after the delivery occurs.


* after $R_r(m, \overline{v})$
$exp = v$
$exp = \overline{v}$
atEnq
atEnq : (m)
$*_{atEnq}(p)$
after $R_r(p, \overline{v})$
at $T_r(q, v)$

Chapter 8

A Specification of Distributed Mutual Exclusion

The intent of this specification is 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.

Figure 8-1 gives the interval logic specification. 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 x(i), each x(j) must be found to be false. Axiom A2 requires that 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 x(i) prior to the entry. Valid formula V5 of section 4 can be used to deduce this from the initial assumption and A2. Similarly we can deduce that x(i) remains true through that interval.

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

Init. $\forall m \neg x(m)$

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

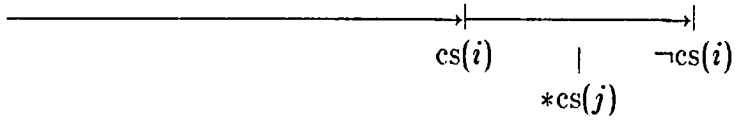
A2. $cs(i) \supset x(i)$

Figure 8-1: Specification of Distributed Mutual Exclusion Algorithm

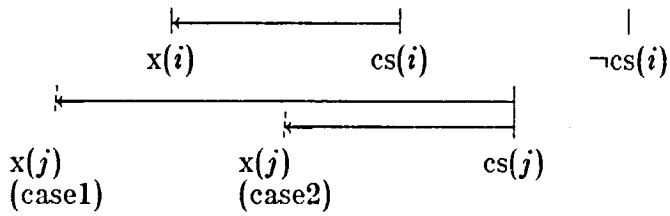
$$\forall m \neg x(m) \wedge i \neq j \supset \Box \neg(cs(i) \wedge cs(j))$$

Pictorially, we show that a violation of mutual exclusion, with both processes in the critical section, requires that one process enter while the other is already in,

or just entering, the critical section.



From the axioms, we know that each entry must be preceded by setting of the coresponding x. Two situations arise. Either setting x(j) precedes setting x(i), or x(j) is set at the same time or after x(i).



In the first case, since the interval x(i) $\Leftarrow$ cs(i) is fully contained in the interval x(j) $\Leftarrow$ cs(j), process i could not have found the required false x(j) in that interval. Similarly, in the second case j could not have found x(i) false. Since neither of these two situations can arise, the postulated violation of mutual exclusion could not occur.

In interval logic, our proof is given in Figure 8-2. With mechanized decision-procedure support in the style of [Shostak/Schwartz/Melliar-Smith82, Plaisted83], the only user input necessary, in principle, is instantiation of the free variable m in our initial assumption, and of I in step L2. More realistically, the proof would likely be decomposed into user-provided steps L2 and L5. The other steps, including the major case split expressed in L1, would follow automatically as part of the complete theory.

Lemma L1 expresses the case split illustrated above, elaborated to include a third case in which a process enters and never exits the critical section. To avoid considering the symmetric argument of which process enters the critical section first, the

L1. $$\forall m\ \neg \mathrm{x}(m)$$
$$\wedge\ \forall k,l\quad k \neq l\ \supset\ \Box \begin{cases} \left[\ \mathrm{begin}(\ \mathrm{x}(k) \Leftarrow \mathrm{cs}(k)\ ) \Rightarrow \right]\left[\Rightarrow \mathrm{begin}(\ \neg\mathrm{cs}(k)\ \right] \neg *(\mathrm{x}(l) \Leftarrow \mathrm{cs}(l)) \\ \qquad\qquad\qquad\qquad \wedge\ \neg *\neg \mathrm{cs}(k)\ \supset\ \neg *(\mathrm{x}(l) \Leftarrow \mathrm{cs}(l)) \\ \wedge \left[\ \mathrm{begin}(\ \mathrm{x}(l)) \Leftarrow \mathrm{cs}(l)\ \right] \neg *(\mathrm{x}(k) \Leftarrow \mathrm{cs}(k)) \end{cases}$$
$$\supset\quad \Box \neg(\ \mathrm{cs}(i)\ \wedge\ \mathrm{cs}(j))$$

L2. $$\neg \mathrm{x}(j)\ \wedge\ i \neq j\quad \supset\quad \Box\left[\ \mathrm{I}\ \right](\ \Box\ \mathrm{x}(i)\ \supset\ \neg *(\mathrm{x}(j) \Leftarrow \mathrm{cs}(j))\ )$$

L3. $$\left[\ \mathrm{x}(m) \Leftarrow \mathrm{cs}(m)\ \right]\ \Box\ \mathrm{x}(m)$$

L4. $$\left[\ \mathrm{cs}(m) \Rightarrow \right]\left[\Rightarrow \mathrm{begin}(\neg \mathrm{cs}(m))\ \right]\ \Box\ \mathrm{x}(m)$$
$$\wedge\ \neg *\neg \mathrm{cs}(m)\ \supset\ \Box\ \mathrm{x}(m)$$

L5. $$\left[\ \mathrm{begin}(\ \mathrm{x}(m) \Leftarrow \mathrm{cs}(m)) \Rightarrow \right]\left[\Rightarrow [\mathrm{begin}(\neg \mathrm{cs}(m))\ \right]\ \Box\ \mathrm{x}(m)$$
$$\wedge\ \neg *\neg \mathrm{cs}(k)\ \supset\ \Box\ \mathrm{x}(m)$$

Figure 8-2: Proof of the Mutual Exclusion Property.

antecedent is expressed in terms of quantified k and l. This lemma is valid within the interval theory.

Lemma L2 states that, if $\mathrm{x}(i)$ is true throughout an interval I, then it is not possible to find the $\mathrm{x}(j) \Leftarrow \mathrm{cs}(j)$ interval. By axiom A1, if the interval were found, there would be within it a $\neg \mathrm{x}(i)$ state, contradicting the antecedent.

Lemmas L3 and L4 state intervals throughout which $\mathrm{x}(m)$ is true. Both lemmas follow directly from axiom A2. Combining L3 and L4, we obtain lemma L5 for the composed interval, from the $\mathrm{x}(m)$ preceding entry until the exit if any, otherwise indefinitely.

Instantiating the free interval variable I in L2 with the intervals of L5, we use the invariant $\Box\ \mathrm{x}(m)$ of L5 to establish the antecedent of the implication in L2. We then use the consequent of L2 to establish each of the three cases of L1, thereby establishing the conclusion and completing the proof.

Chapter 9

Analysis and Conclusions

This report presents a preliminary version of the Interval Logic and illustrates its application to several different problem domains. We are reasonably satisfied with its success, although we expect further honing of the language as we gain more experience with specification and verification attempts. But much remains to be done before Interval Logic can be used for the specification and verification of operating systems or asynchronous applications programs. The next steps required are:

- The current logic does not distinguish the various concurrent processes of a multiprocess system, and does not attribute operations to processes. A notation is required to identify processes and to associate operations and state variables with processes.
- A method must be devised for composing together the specifications of individual processes, or of small multi process systems, so as to form the specification of a larger multiprocess system.
- Theory and techniques must be developed to allow the hierarchical development of specifications in Interval Logic, with temporal mappings between levels of abstraction.

- Using these techniques, the semantics of a concurrent programming language must be defined.
- Methods must be developed to use Interval Logic, with the language definition and the specification of the required behavior, to develop a concurrent program verification method. Taking advantage of the power of Interval Logic to improve on the efficiency of the current methods will be important if the method is to be effective in practical use.
- Although an initial version of a decision procedure for the Interval Logic has been constructed, further investigation of the theory of deciding the logic. Work is still required to improve its performance and also to integrate it into the Specification and Verification Environment currently under development.
- Interval Logic lends itself to graphical representation, and we feel that such graphical representations can greatly assist in human comprehension of concurrent specifications, which are otherwise difficult to understand. The mechanical support for such graphical representation, both input and output, requires investigation.

At the heart of the interval logic design is the decision to support a behavioral style of specification and reasoning. A *cause/effect* style pervades our specifications – always of the form "given a particular context, some future behavior of the system must occur". As discussed in [Schwartz/Melliar-Smith82], this form of specification is closer to the intuitive operational understanding of requirements, while still managing to avoid details of operational implementation. More history-related specifications, capturing a static view of necessary relationships between different input/output histories, don't seem to provide the same degree of intuition crucial to understanding and reasoning about a system from its specification.

The decision to base interval formation on "state-change events" was motivated by the observation that establishing context almost always required seeing a change in state. Without "anchoring" requirements on properties *becoming* true, one often cannot guarantee that the proper interval has been identified. This is

particularly true for eventuality properties.

Two language decisions related to this notion of context establishment are the decisions (1) to make interval formulas vacuously true whenever the context cannot be established, and (2) to interpret interval formulas as properties of the *next time* the context occurs. Both these decisions support an abstract form of operational thinking. Having sufficient expressive power to conveniently establish context requirements either temporally or through the use of state components proved to be an important method of directing the level of abstraction of the specification.

Based on previous experience with formal specification methods, we do not think *any* specification method for distributed and concurrent systems can be successful without mechanical verification support. The level of process interaction makes it only too easy to make incorrect or incomplete analysis of specifications, regardless of the amount of human care that is taken. Experience with informal proof techniques and unverified specifications have led us to include mechanical verification support as a *crucial* part of any specification language design effort. The emphasis in designing the interval logic was to retain decidability in order to provide a complete decision procedure. Although interval logic has a complete axiomatization, through a reduction to linear-time temporal logic, we do not expect anyone to attempt to use the axiomatization in doing a proof. For this reason, we chose features on the basis of utility rather than mathematical elegance.

One direction for further work that may prove extremely fruitful is development of a formal graphical representation of specifications and proofs. The ability to represent specifications and proof arguments pictorially could greatly enhance intuitive understanding of temporal properties.

Preliminary analysis of the computational complexity of the logic indicates it is P-space complete – the same order of complexity as for linear-time temporal logic. We, with David Plaisted playing the primary role, have developed an experimental decision procedure for interval logic, described in the attached papers by Dr Plaisted.

Several other higher-order temporal languages have appeared in the literature. Lamport introduced a Timeset language[Lamport80] for defining properties of intervals. At the heart of the language proposal are terms of the form $[P \Rightarrow Q)$, denoting the set of *all* time intervals starting with a state in which property P is true and extending to *all* points such that Q has remained false. Such all-inclusive terms make it difficult to avoid capturing unexpected and unwanted contexts, and, we believe, result in nonelementary computational complexity.

Wolper[Wolper82] introduced the concept of a regular-expression grammar operator into his Extended Temporal Logic (ETL). These grammar operators are used to define constraints, in the form of regular expressions, on allowable sequences of parameterized operations. This produces very abstract specifications, in much the same style as Hailpern's[Hailpern80] history-based, linear-time temporal logic. Wolper's extension preserves P-space complexity.

With a somewhat different focus, Moszkowski[Moszkowski82] uses a related notion of interval logic to define and prove properties of hardware circuits. Moszkowski integrates specification of quantitative bounds into his hardware description language. While our interval logic is oriented toward identifying properties true of specified contexts, Mozkowski's logic provides interval abstraction, that is, a method to refer to all intervals having a certain property or decomposition. A semicolon operator, similar in spirit to the dynamic logic[Harel79] "chop" operator, allows formulas such as $[\ P\ ;\ Q\ ]$ to refer to all intervals composed from subintervals having properties P and Q. This very powerful concept again leads to nonelementary computational complexity.

Acknowledgments

Discussions with Joe Halpern and David Plaisted have been extremely valuable in helping us to explore the complexity of our logic and to investigate decision procedure support for the logic. Joe Halpern, David Plaisted, Moshe Vardi, Ronald Peikert, Ed Ashcroft, Leslie Lamport, Jan Vitopil, and Rob Shostak have also contributed many valuable suggestions for improving the interval logic and its presentation.

References

[1] Bochmann, G.V., "Hardware Specification with Temporal Logic: An Example", *IEEE Transactions on Computers*, Vol C-31, No. 3, March 1982.

[2] Hailpern, B., "Verifying Concurrent Processes Using Temporal Logic", Technical Report 195, Computer Systems Laboratory, Stanford Univ., August 1980.

[3] Harel, D., "First-Order Dynamic Logic", Springer Verlag Lecture Notes, No. 68, 1979.

[4] International Standards Organization, "Data Processing - Open Systems Interconnection - Basic Reference Model", ISO/DIS 7498, April 1982.

[5] Lamport, L., "Timesets: A New Method for Temporal Reasoning about Programs", *Logics of Programs Conference*, Springer Verlag, Vol. 131, Sept. 1981.

[6] Moszkowski, B., "A Temporal Logic for Multi-Level Reasoning about Hardware", Technical Report STAN-CS-82-952, Computer Science Dept., Stanford Univ., Dec. 1982.

[7] Plaisted, D., "An Intermediate-Level Language for Obtaining Decision Procedures for a Class of Temporal Logics", Appendix C of this report.

[8] Schwartz, R., P.M. Melliar-Smith, "Temporal Logic Specification of Distributed Systems", *Proceedings of the IEEE Conference on Distributed Systems,* April 1981.

[9] Schwartz, R., P.M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards", *IEEE Transactions on Communications,* Dec. 1982.

[10] Seitz, C., "Ideas about Arbiters", *Lambda,* pp. 10-14, First Quarter, 1980.

[11] Shostak, R., R. Schwartz, P.M. Melliar-Smith, "STP: A Mechanized Logic for Specification and Verification", *6th Conference on Automated Deduction,* Springer Verlag Lecture Notes, Vol. 138, June 1982.

[12] Vogt, F., "Entwurf eines Ereignisorientierten Modells zur Spezifikation von Verteilten Systemen Mittels Temporaler Logik", Ph.D. Dissertation, Technische Universität Wien, Austria, Feb. 1982.

[13] Vogt, F., "Event-Based Temporal Logic Specification of Services and Protocols", *Protocol Specification, Testing and Verification,* North-Holland Publishing, 1982.

[14] Wolper, P., "Synthesis of Communicating Processes from Temporal Logic Specifications", Report No. STAN-CS-82-925, Dept of Computer Science, Stanford University, August 1982.

Appendix A.

Reduction of Formulas Containing $*$ Modifier

The $*$ modifier in the interval language is regarded as a linguistic convenience. Below, we give axioms to reduce a formula containing the $*$ modifier to an equivalent formula without the modifier. In this section we denote interval terms possibly containing the $*$ modifier by $\hat{I}$ and $\hat{J}$.

We base the reduction on the following equivalence

$$[\hat{I}]\alpha \equiv [I']\alpha \wedge [\hat{I}]\text{true}$$

where I' is derived from $\hat{I}$ by omitting throughout the $*$ modifiers. We also use the definition of $*\hat{I}$ to reduce the eventuality on intervals to an interval formula

$$*\hat{I} \equiv \neg[\hat{I}]\text{false}$$

For the outer level of interval structure, we use:

$$[*\hat{I}]\text{true} \equiv [\Rightarrow *\hat{I}]\text{true} \equiv *\hat{I}$$

$$[*\hat{I}\Leftarrow]\text{true} \equiv *(\hat{I}\Leftarrow)$$

$$[\text{begin} * \hat{I}]\text{true} \equiv [*\,\text{begin}\hat{I}]\text{true}$$

$$[\text{end} * \hat{I}]\text{true} \equiv [*\,\text{end}\hat{I}]\text{true}$$

and for splitting composite intervals we use:

$$[\hat{I}\Rightarrow\hat{J}]\text{true} \equiv [\hat{I}\Rightarrow][\Rightarrow\hat{J}]\text{true}$$
$$[\hat{I}\Leftarrow\hat{J}]\text{true} \equiv [\Leftarrow\hat{J}][\hat{I}\Rightarrow]\text{true}$$

Finally we give reduction rules for the four composite intervals that cannot be reduced by simple splitting of an interval.

$$[\Rightarrow(\hat{I}\Rightarrow\hat{J})]\text{true} \equiv [\hat{I}\Rightarrow\hat{J}]\text{true}$$
$$[\Rightarrow(\hat{I}\Leftarrow\hat{J})]\text{true} \equiv [\hat{I}\Leftarrow\hat{J}]\text{true}$$
$$[(\hat{I}\Rightarrow\hat{J})\Leftarrow]\text{true} \equiv [\text{begin}(\hat{I}\Leftarrow)\Rightarrow\hat{J}]\text{true}$$
$$[(\hat{I}\Leftarrow\hat{J})\Leftarrow]\text{true} \equiv [\hat{J}\Leftarrow\text{begin}(\hat{I}\Leftarrow)]\text{true}$$

Appendix B.

A Decision Procedure for Combinations of Propositional Temporal Logic and Other Specialized Theories

David A. Plaisted
SRI International, University of Illinois

Abstract

We present two decision procedures for formulae of discrete linear time propositional temporal logic whose propositional part may include assertions in a specialized theory. The combined decision procedures may be viewed as extensions of known decision procedures for quantifier-free theories to theories including temporal logic connectives. The first runs in polynomial space relative to an oracle for the underlying theory. The second is more modular but requires the computation of least and greatest fixed points and may have a worse asymptotic running time. However, the second procedure can handle assertions containing arbitrary mixtures of extralogical variables, whose values cannot change with time, and state variables, whose values can change with time. The second procedure has been implemented efficiently enough to be practical. The same techniques appear to apply to logics other than temporal logic which have tableau-like decision procedures.

This research was supported in part by National Science Foundation Grant MCS-81-09831.

1. Introduction

Various temporal logics have been proposed recently for reasoning about concurrent programs. The advantage of temporal logic is that primitives are available for expressing time relationships concisely. The application of temporal logic to concurrent programs is discussed in [6] and [2] and [3]. In [1], temporal logic specifications are used to guide the synthesis of programs having the desired behavior. This paper makes use of a tableau-like satisfiability algorithm for propositional temporal logic. An extension of propositional temporal logic is given in [10] for specifying and synthesizing programs written in the language of communicating sequential processes developed by Hoare. The complexity of deciding satisfiability of propositional temporal logic formulae is discussed in [9] for several variants of temporal logic.

In practice, one is often interested in deciding validity or satisfiability of temporal formulae involving theories for which specialized decision procedures are available. For example, to verify that "Henceforth $a > 1$ implies eventually $a \geq 0$" requires reasoning not only about time but also about inequalities and integers. We develop a method for deciding the satisfiability (or validity) of such formulae. The method we give applies in general to logics having tableau like decision procedures similar to that for temporal logic.

We consider discrete linear time temporal logic similar to that described in [3]. The formulae of this logic are composed of predicate symbols P_i, Q_i, R_i, atoms (predicate symbols followed by a list of arguments which may contain variables, constants, and function symbols), the usual Boolean connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), and the temporal connectives $\Box$ (henceforth), $\Diamond$ (eventually), U (until), and $\circ$ (next time). Predicate symbols by themselves are considered as special cases of atoms. Of the temporal connectives, $\Box$, $\Diamond$, and $\circ$ are unary operators and U is binary. Our semantics is similar to that given in [3] except that U does not imply an eventuality; that is, U(p, q) is true if p is henceforth true and q never becomes true. The decision procedure could be adapted to either version of U. For a detailed description of the semantics of these formulae, see [3]. We give a brief description here.

An interpretation consists of an infinite sequence of states, representing the world at successive instants of time. Each predicate symbol is given a Boolean value in each state (at each instant of time); the variables and function symbols in atoms are also interpreted so that atoms may be given Boolean values in the usual way. The interpretations of variables may differ depending on whether the variable is an

extralogical variable, whose value does not change with time, or a *state variable*, whose value may change with time. From these values, the interpretation of an arbitrary formula in a state is defined. A formula is *valid* if it is true in all states in all interpretations, and it is *consistent* if it is true in some state in some interpretation. The Boolean connectives are interpreted as usual, so that for example $P \wedge Q$ is true in a state if P is true in the state and Q is true in the state. A formula $\Box A$ is true at time t if A is true at time t and at all successive times; a formula $\Diamond A$ is true at time t if A is true at time t or at some later time; a formula $\circ A$ is true at time t if A is true at time $t+1$, and $U(P,Q)$ is true at time t if either $\Box P$ is true at time t or there exists time u, $u \geq t$ such that Q is true at time u and P is true at all times v, $t \leq v < u$. For example, the formula $\Diamond \Box P \supset \Box \Diamond P$ is valid since if at some future time, P is henceforth true, then at all future times, P is eventually true. However, the formula $\Diamond P \supset \Box P$ is satisfiable but not valid, since P may be eventually true without being henceforth true. We write $TL \models A$ if A is a valid temporal logic formula.

Suppose only a subset of the models are considered. For example, the atoms may actually be quantifier-free formulae involving integers, addition, and inequalities, and we are only interested in interpretations consistent with the theory of linear inequalities of integers. The formula $\Box(y = x + x) \supset \Box(y = 2x)$ is true in all such interpretations, but not valid in the uninterpreted case. In general, we assume a theory $\mathcal{T}$ which is time-independent and is a subset of the interpretations of the predicate symbols and atoms at each instant of time. Thus the same interpretations of the predicate symbols are permitted at all time instants. We write $\mathcal{T} \models A$ to indicate that A is a Boolean combination of atoms which is valid in $\mathcal{T}$, that is, A is true in all interpretations in $\mathcal{T}$. We write $TL(\mathcal{T}) \models A$ to indicate that A is a temporal logic formula which is true in all interpretations allowed by $\mathcal{T}$; that is, the interpretation of the atoms at any time instant must be a member of $\mathcal{T}$. We say informally that A is valid in $\mathcal{T}$ in this case. Satisfiability is defined as usual.

The complexity of the satisfiability problem for such formulae without specialized theories is PSPACE complete [9]. In the presence of specialized theories, the complexity can be much greater; it will be at least as high as the complexity of the theory $\mathcal{T}$ being used. We will show that the complexity can never be much higher than this. In particular, the decision procedure is of PSPACE complexity relative to an oracle for deciding the specialized theory. The PSPACE upper bound can be realized by an algorithm which we shall call Algorithm A. However, another decision procedure (Algorithm B) for the combined theory has a more modular structure, requiring no

interaction between the tableau method and the specialized theory, but may have a worse asymptotic behavior. Algorithm B has the advantage, however, that it can be used for formulae containing variables whose values do not change with time (which we shall call *extralogical variables* below, to distinguish them from *state variables* whose values can change with time).

2. Extralogical variables

The variables in atoms can be of two types, *state variables* and *extralogical variables*. State variables have values that may change from one time instant to the next; extralogical variables have the same values at all times. Thus the formula $x = 1 \supset \circ(x \neq 2)$ is valid if x is an extralogical variable but not if x is a state variable. We are interested in deciding the validity of formulae containing both kinds of variables, in the presence of specialized theories. For example, suppose $A(x, y, z, u, v, w)$ is a temporal formula in which x, y, and z are extralogical variables and u, v, and w are state variables. We give a method for constructing a formula $T_A(x, y, z, u, v, w)$ not containing temporal connectives, such that $TL(\mathcal{T}) \models \forall x \forall y \forall z A(x, y, z, u, v, w)$ iff $\mathcal{T} \models \forall x \forall y \forall z T_A(x, y, z, u, v, w)$. This method uses Algorithm B, and is therefore apparently of a higher complexity than Algorithm A in some cases.

3. The tableau method

Without going into details, we give enough of the tableau method to describe the workings of Algorithms A and B. Given a temporal logic formula A, we decide if $TL \models A$ by negating A and constructing a graph $G = Graph(\neg A)$ which represents the set of models of $\neg A$. The nodes of G represent states and are labeled with formulae which must be true in the state. A node may be labeled with several formulae; in that case, all of the formulae labeling the node must be true in the state. One of the nodes is distinguished as the *initial node* of G and is labeled with $\neg A$. The edges are labeled with conjunctions of literals, where a *literal* is an atom or the negation of an atom. The edges may also be labeled with *eventualities*, which represent temporal formulae which must eventually be satisfied in any model of $\neg A$. Nodes with no outgoing edges may be deleted from G; similarly, edges are deleted if their terminal node is deleted, if the conjunction of literals labeling the edge is a contradiction, or if the edge is labeled with an eventuality which cannot be satisfied. An eventuality A on edge E can be satisfied iff there is a path in the graph from the terminal node of E to some node N having A as one of its labels. Let $Iter(G)$ represent the graph that results from iterating all such deletions on G until no more deletions are possible. It turns out that A is valid ($TL \models A$) iff the initial node of $Graph(\neg A)$ is deleted in $Iter(Graph(\neg A))$. If a specialized theory $\mathcal{T}$ is specified, the graph or the iteration must be modified in a manner to be described below to determine whether $TL(\mathcal{T}) \models A$.

The tableau method works because of the following chain of reasoning: A is non-valid iff there is an interpretation in which $\neg A$ is true, iff there is an infinite path through $Graph(\neg A)$ starting at the initial node, such that all eventualities are satisfied, iff the initial node is not deleted from $Iter(Graph(\neg A))$. We now explain what is meant by "all eventualities are satisfied" on an infinite path. Suppose that the infinite path is $\{e_1, e_2, ..., e_n, ...\}$ where the e_i are edges and e_i is an edge from node N_i to node N_{i+1}. (There may be more than one edge between two nodes in the graph.) Suppose A is a temporal formula. Then we say A is *reachable from* node N_i if there exists j, $j \geq i$, such that A is one of the labels of node N_j. If A is an eventuality labeling edge e_i, then we say A is *satisfied at* edge e_i if A is reachable from node N_{i+1}. Finally, all eventualities are satisfied on this infinite path if for all edges e_i in the path and all eventualities A labeling e_i, A is satisfied at edge e_i. If there is a theory $\mathcal{T}$ specified, then $TL(\mathcal{T}) \not\models A$ iff there is an infinite path as above in which for all i, the conjunction of literals labeling e_i is satisfiable in $\mathcal{T}$.

4. Algorithm A

The first algorithm is quite simple. Before iterating to obtain $Iter(G)$, we first delete from G all edges E labeled with a conjunction of literals which is unsatisfiable in $\mathcal{T}$. Other than this, the algorithm is exactly as in the general tableau method. One can easily verify that this is correct by reasoning similar to that used above to justify the general tableau method; one disadvantage is that this method requires a closer interaction between the decision procedure for $\mathcal{T}$ and the tableau method than Algorithm B. However, Algorithm A has the advantage that edges can be deleted using the specialized theory as the graph is constructed, so it may be that whole sections of the graph need not be constructed at all. It is clear that this method can be performed in polynomial space relative to an oracle for deciding $\mathcal{T}$, by nondeterministically guessing a long enough path through the graph.

5. Algorithm B

The second algorithm requires a much more complicated iteration method on the graph G. Given the graph $Graph(\neg A)$, the method constructs a formula C representing conditions guaranteeing the validity of A. To be precise, C is a maximal formula $\bigvee_i \Box C_i$ where the C_i are Boolean combination of literals of A, such that $TL \models (C \supset A)$. We mean "maximal" in the sense of being true the most often possible. Note that C does not depend on $\mathcal{T}$.

Theorem 1. *$TL(\mathcal{T}) \models A$ iff $TL(\mathcal{T}) \models C$ iff for some i, $\mathcal{T} \models C_i$.*

Proof: If for some i, $\mathcal{T} \models C_i$, then $TL(\mathcal{T}) \models C$ (and conversely). However, $TL(\mathcal{T}) \models (C \supset A)$ and so $TL(\mathcal{T}) \models A$ by modus ponens. Conversely, suppose $TL(\mathcal{T}) \models A$. Let D be a Boolean combination of atoms in A such that for all interpretations I of the atoms in A, $I \models D$ iff I can be extended to an element of $\mathcal{T}$ by assigning Boolean values to other atoms. Thus, for all Boolean combinations D_1 of atoms of A, $D \supset D_1$ iff $\mathcal{T} \models D_1$, and D represents the set of interpretations of atoms in A consistent with $\mathcal{T}$. Since D specifies all permissible assignments of Boolean values to atoms of A consistent with $\mathcal{T}$, and $TL(\mathcal{T}) \models A$, it follows that $TL \models \Box D \supset A$. Since C is maximal, $TL \models \Box D \supset C$. By properties of temporal logic, $D \supset C_i$ for some i (in propositional logic). Since $\mathcal{T} \models D$, $\mathcal{T} \models C_i$ for some i. ∎

In method B, the formula C is found and then each C_i is given to a decision procedure for $\mathcal{T}$. If some C_i is valid in $\mathcal{T}$, then A is valid in the combined theory $TL(\mathcal{T})$; otherwise, A is non-valid in this combined theory. Note that algorithm B is more modular than algorithm A since there is little interaction between the tableau method and the decision procedure for $\mathcal{T}$. The decision procedure is called only when the graph construction and iteration are completed. Also, algorithm B may require fewer calls to the decision procedure than algorithm A. For example, if the formula A is valid in pure temporal logic, then algorithm B will not use the decision procedure for $\mathcal{T}$ at all, but algorithm A may. We do not know if algorithm B can be done in polynomial space relative to an oracle for $\mathcal{T}$. One might try to guess a condition C_i such that $\mathcal{T} \models C_i$ and then verify that $TL \models \Box C_i \supset A$; however, such a C_i may itself be of size exponential in the size of A. Furthermore, extralogical variables require that C be computed as a whole, and a nondeterministic guess of C_i is not sufficient.

If A has universally quantified extralogical variables, then these are included in C. Thus if A is $\forall x \forall y \forall z A(x,y,z)$ where x, y, and z are extralogical variables, then C is $\forall x \forall y \forall z C(x,y,z)$ where C(x, y, z) is obtained from A(x, y, z) the same way C is obtained from A above.

Corollary 2. $TL(\mathcal{T}) \models \forall x_1 ... \forall x_n A(x_1 ... x_n)$ *iff* $TL(\mathcal{T}) \models \forall x_1 ... \forall x_n C(x_1 ... x_n)$ *where the* x_i *are extralogical variables.*

Proof: Similar to the theorem. ∎

To apply this result, we need a way of reducing the decision procedure for formulae $\forall x_1 ... \forall x_n C(x_1 ... x_n)$, which are quantified temporal logic formulae, to formulae in the theory $\mathcal{T}$. Note that $TL(\mathcal{T}) \models \forall x_1 ... \forall x_n C(x_1 ... x_n)$ iff

$$\forall x_1 ... \forall x_n \exists i \mathcal{T} \models C_i(x_1 ... x_n) \tag{1}$$

where C_i are as before except with extralogical variables included. In order to use specialized decision procedures for the theory $\mathcal{T}$, it is necessary to make statement (1) into a statement in the theory $\mathcal{T}$. There are ways of doing this for theories having certain common properties. For example, suppose $\mathcal{T}$ has the property that for all closed formulae B, either $\mathcal{T} \models B$ or $\mathcal{T} \models \neg B$. (A formula is *closed* if it has no free variables.) Then statement (1) is true iff

$$\mathcal{T} \models \forall x_1 ... \forall x_n \vee_i C_i^*(x_1 ... x_n). \tag{2}$$

where C_i^* is C_i with all state variables universally quantified. Thus the difference between extralogical variables and state variables is one of scope; extralogical variables have the whole formula (2) as scope, whereas state variables have only a singel C_i^* as scope. If $\mathcal{T}$ has uninterpreted function symbols, it will be necessary to rename these in each C_i^* so that no two formulae C_i^* have common uninterpreted function symbols.

5.1. Example

Suppose C is $\Box(x > 0) \vee \Box(x < 1)$. If x is a state variable, this formula is valid in $TL(\mathcal{T})$ iff $\mathcal{T} \models \forall y(y > 0) \vee \forall z(z < 1)$. Thus C would not be valid for ordinary arithmetic. If x is an extralogical variable, then C is valid iff $\mathcal{T} \models \forall x(x > 0 \vee x < 1)$, hence C would be valid for ordinary arithmetic.

5.2. Existential quantifiers

There are problems with extending the method to existentially quantified extralogical variables. In fact, we have not even been able to give an upper bound in the arithmetic hierarchy [11] for the decision problem for formulae of the form $\exists x A(x)$ where x is an extralogical variable and A is a temporal formula. For example, consider the formula

$$\Box \forall y(z = y \supset \circ(z = y - 1)) \supset \Diamond(z < 0)$$

where y is an extralogical variable and z is a state variable. This formula is valid in the usual interpretation of arithmetic, but to show this requires an inductive argument. This formula becomes of the form $\exists y A(y)$ when the universal quantifier is moved to the outside of the formula.

5.3. Iterating to obtain C

We now discuss the method of computing C, which involves a double iteration on the graph G. We compute a set of conditions delete(N) for nodes N of G and fail(A, N) for nodes N of G and eventualities A of edges of G. These conditions are defined in terms of one another by a set of equations. It will turn out that for delete(N) we want the minimal solution of these equations and for fail(A, N) we want the maximal solution, where FALSE is minimum and TRUE is maximum as usual. To compute the solution requires a double iteration.

The condition delete(N) gives the condition under which node N will be deleted from the graph G. The condition fail(A, N) gives the condition under which the

eventuality A will not be reachable by a path from the node N. Intuitively, since nodes will not be deleted unless they are forced to be deleted, we find the minimal solution for delete(N); however, an eventuality is not reachable unless it is forced to be reachable, hence we want the maximal solution for fail(A, N).

Given edge e of G, let fin(e) be the final node of e, event(e) be the set of eventualities labeling e, and prop(e) be the conjunction of literals labeling e. Thus prop(e) is the "propositional part" of e. Given a node N of G, let edges(N) be the set of edges whose initial node is N. We have the following equations for delete(N) and fail(A, N):

$$delete(N) = \wedge_{e \in edges(N)}(\Box \neg prop(e) \vee delete(fin(e)) \vee \vee_{A \in event(e)} fail(A, fin(e))) \quad (3)$$

$$fail(A, N) = \wedge_{e \in edges(N)}(\Box \neg prop(e) \vee delete(fin(e)) \vee (A \in event(e) \wedge fail(A, fin(e)))) \quad (4)$$

Let Delete be a vector of deletion conditions for the nodes of G, and let Fail be a vector of fail conditions for edges and eventualities of G. We compute Delete and Fail by iteration using the functionals $\mathcal{F}_D$ and $\mathcal{F}_F$ where $\mathcal{F}_D$ uses equation (3) to compute new values of Delete from old values of Delete and Fail, and $\mathcal{F}_F$ uses equation (4) to compute new values of Fail from old values of Delete and Fail. The iteration proceeds as follows:

1. Set all elements of Delete to False.
2. Set all elements of Fail to True.
3. Repeat 4, 5, and 6 until both Delete and Fail are unchanged:
4. Iterate Fail := $\mathcal{F}_F(Delete, Fail)$ until no change.
5. Iterate Delete := $\mathcal{F}_D(Delete, Fail)$ until no change.
6. Set all elements of Fail to True.
7. Return Delete of initial node as the condition C.

Since Fail is set to True before each iteration, the maximal fixpoint of Fail will be computed. Since Delete is initially set to False, the minimal fixpoint of Delete will be computed. Note that extralogical variables of A must be universally quantified.

6. Implementation

Method B has been implemented in Interlisp on the F2 computer and appears to be of reasonable complexity. For formulae of moderate size, the graph construction and iteration typically take about a minute of compute time or less; the graph construction usually takes longer than the iteration. The iteration was greatly sped up by finding the strongly connected components of G and iterating on them in order. Thus if G1 is a strongly connected component of G having no edges leading out of G1, then the Fail and Delete conditions can be iterated to a fixpoint in G1 before iterating on the rest of G. This can be extended component by component to the whole graph and avoids much repeated computation.

As examples of formulae run on the program, we give the following formulae R3, R4, and R5:

R3: $\Box\, LUA(A,X) \wedge \Box\, LUA(A,Y) \supset \Box\, LUA(A, X \wedge Y)$

R4: $\Box\, LUA(A, B \wedge C) \wedge \Box\, LUA(B, A \wedge \neg C) \supset \Box\, LUA(A \vee B, FALSE)$

R5: $LUA(A,B) \wedge LUA(B,C) \supset LUA(A \vee B, C)$

Here LU(X, Y) is defined to be $U(\neg P, U(P \wedge \neg Q, Q))$ and LUA(X, Y) is defined to be $LU(A, A \wedge B)$. The times to construct the graph and iterate and the number of nodes and edges in the graph are given in the following table. These formulae were all shown to be valid in pure temporal logic.

	Graph Construction (*Seconds*)	*Iteration* (*Seconds*)	*Nodes*	*Edges*
*R*3	67	14	13	108
*R*4	105	22	16	166
*R*5	13.8	5	8	34

7. Extensions

We are studying an interval based temporal logic developed by Schwartz, Melliar-Smith, and Vogt [7] which also has a PSPACE complete decision problem in the absence of specialized theories. The above methods can be extended to this logic, and probably to any temporal or modal logic having a tableau like decision procedure. Since the decision procedures for specialized theories developed by Nelson, Oppen, and

others [4], [5], [8] have proven to be of considerable practical value in the verification of non-concurrent programs, it is possible that the methods presented here will also be of much value in the verification of concurrent programs using temporal logic.

8. Acknowledgements

This study grew out of discussions with Richard Schwartz, Michael Melliar-Smith, Robert Shostak, and Fritz Vogt. The facilities provided by SRI International provided the environment for this work.

REFERENCES

[1] Manna, Z. and Wolper, P. Synthesis of Communicating Processes from Temporal Logic Specifications, 253-281 in *Proceedings of the Workshop on Logics of Programs*, , 1981.

[2] Manna, Z. and Pnueli, A. Verification of Concurrent Programs: the Temporal Framework, 215-273 in *The Correctness Problem in Computer Science*, R. S. Boyer and J. S. Moore,Ed.,Academic Press , 1981.

[3] Manna, Z. and Pnueli, A. Verification of Concurrent Programs, Part II: Temporal Proof Principles, Department of Computer Science, Stanford University Tech. Report STAN-CS-81-843 (Sep. 1981).

[4] Nelson, G. and Oppen, D. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems 1*,2 (Oct. 1979).

[5] Nelson, G. and Oppen, D. Fast decision procedures based on congruence closure. *J. ACM 27*,2 (1980).

[6] Pnueli, A. The Temporal Logic of Programs, 46-57 in *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, IEEE, Nov. 1977.

[7] Schwartz, R., Melliar-Smith, P. M., and Vogt, F. An Interval Logic for Higher Level Temporal Reasoning. NASA Contractor Report 172262.

[8] Shostak, R. Deciding Combinations of Theories, 209-222 in *Proceedings of the 6th Conference on Automated Deduction*, , 1982.

[9] Sistla, A. P. and Clarke, E. M. The Complexity of Propositional Linear Temporal Logics, 159-168 in *Proceedings of the 14th ACM Symposium on Theory of Computing*, , May 1982.

[10] Wolper, P. Synthesis of Communicating Processes from Temporal Logic Specifications, Ph.D. Thesis Stanford University, (Aug. 1982).

[11] Yasuhara, A. Recursive Function Theory and Logic, Academic Press , 1971.

Appendix C.

A Low Level Language for Obtaining Decision Procedures for Classes of Temporal Logics

David A. Plaisted
SRI International, University of Illinois

1. Introduction

We present a low level language which has been found convenient for obtaining a decision procedure for the interval logic of Schwartz, Melliar-Smith, and Vogt [7]. This language is a generalization of regular expressions, and is expressive enough so that there are easy translations of other temporal logics into the low level language. We give a non-elementary decision procedure for the language with a certain syntactic restriction. This procedure requires that eventualities be treated in a nonstandard way; the reason seems to be that this language deals with concatenation of sequences as well as with the usual temporal connectives. The low level language is convenient for expressing synchronization constraints such as mutual exclusion and thus may have applications to automatic generation of concurrent programs as described in Manna and Wolper[3]. It would also be interesting to investigate relationships of this language to the path expressions of [1].

1.1 Sets of computations

The most natural way to view the language is that each expression represents a set of computation sequence constraints. A *computation sequence constraint* is a sequence of sets of permitted and forbidden events, specifying which events may or may not occur at various instants of time. For example, we specify that event x is permitted and events y and z are

forbidden at a given instant of time by the conjunction $x \wedge \bar{y} \wedge \bar{z}$, where x, y, and z are propositional variables. Thus a computation sequence constraint may be represented by a sequence of conjunctions of propositional variables and negations of propositional variables. A *computation sequence* over a given set X of events is a sequence of conjunctions C in which for each x in X, either x occurs in C or $\bar{x}$ occurs in C but not both. This represents the computation in which event x occurs at time i if x is in the i^{th} conjunction, and event x does not occur at time i if $\bar{x}$ is in the i^{th} conjunction. Such a computation sequence satisfies a constraint if permitted events occur when specified by the constraint and forbidden events do not occur when forbidden by the constraint. Sets of such constraints represent the disjunction of their elements; that is, a computation sequence satisfies a set S of constraints if the computation sequence satisfies some element of S. The language has connectives for expressing concurrency, nondeterministic choice, iteration, concatenation, "hiding" of events, and "exceptional events" which are false unless specified to be true, or true unless specified to be false. Note that this language differs from dynamic logic[6] in that we consider computation sequences rather than just input-output relations of programs.

2. Syntax

The language consists of well-formed expressions built up from propositional variables and their negations, the following constants:

T (True), F(False), T^*,

the following unary operations:

infloop, $\exists x$, Fx, Tx (for propositional variable x),

the following binary connectives:

$\wedge$ (conjunction), $\vee$ (disjunction), as, concatenation, ";", iter*, iter(*)

Expressions in the language are denoted by α, β, γ, δ. The concatenation of α and β is written as $\alpha\beta$. Also, infloop(α) is sometimes written α^∞. Thus $(\exists x)[y \wedge (Fx)(T^*x)]$ is an example of a formula. The quantifier $\exists x$ binds the variable x according to the usual scope rules; Fx and Tx do not bind x, although they can also be viewed as quantifiers. Thus in the formula $(Fx)(x \wedge y)$, both x and y are free variables; in the formula $(\exists x)(x \wedge y)$, y is free but x is not free. Therefore in the formula $(\exists x)(y \wedge (Fx)x)$, the same x is referred to by $(\exists x)$ and by (Fx). Negation can only be applied to propositional variables; this restriction seems natural for the examples we have considered.

3. Semantics

Our method of defining semantics is nonstandard, but seems most convenient for this language. With each formula α we associate a set $\Psi(\alpha)$ of *partial interpretations,* where a partial interpretation is a finite or infinite sequence of conjunctions of propositional variables and negations of propositional variables. These are the same as the "computation sequence constraints" introduced in section 1.1. Thus a formula represents a set of constraints; later we introduce another semantics in which a formula represents the set of computations satisfying at least one of these constraints. This is an example of a partial interpretation:

$$P,\ \bar{P} \wedge Q,\ \bar{R},\ F,\ T,\ R$$

If I is a partial interpretation then |I| is the length of I (so the length of the above example is 6). The letters I and J will be used for partial interpretations. A formula α is *satisfiable* if there exists I in $\Psi(\alpha)$ such that no conjunction of I is contradictory.

Intuitively, propositional variables z represent computation sequences consisting of the single event z, negations $\bar{z}$ of propositional variables represent computation sequences consisting of a single time instant in which z does not occur, T represents any computation sequence of length one (that is, consisting of one instant of time), F represents no computation sequence, T^* represents any finite or infinite computation sequence, $\alpha \vee \beta$ represents the nondeterministic choice of α or β, $\alpha \wedge \beta$ represents concurrent execution of α and β, with the longer computation extended past the shorter one, $\alpha\beta$ represents concurrent execution for sequences of the same length, $\alpha;\beta$ represents serial composition of α and β, $\alpha\beta$ represents serial composition of α and β in which the last state of α is concurrent with the first state of β, and $(\exists z)\alpha$ represents the computation of α with the events z "hidden;" this permits "local events" not visible outside of $(\exists z)\alpha$. Such local events can be used for message passing or synchronization within a subcomputation, for example. Also, $(Fz)\alpha$ represents computations of α in which the event x is made false everywhere except where it is specified to be true, and $(Tz)\alpha$ represents computations of α in which the event x is made true everywhere except where it is specified to be false. In addition, α^∞ represents computation sequences in which a copy of α is begun at each successive time instant from now on, $iter(*)(\alpha, \beta)$ represents computation sequences in which copies of α are begun at successive time instants until possibly some future time, at which β is begun; and $iter*(\alpha, \beta)$ is the same except that β must eventually be started, and up to that time, copies of α are begun. Furthermore, these last three "iteration" operators require that all relevant α and β computations end at the same time. Possibly this simultaniety requirement can be dropped. We could add a constant ϵ to the language, representing a sequence of length zero, but this has not been necessary.

We give an example to show how the language can express synchronization

constraints. Let α and β be formulae of the language in which neither of the propositional variables x or y occur free. Consider the expression

$$(Fx)(T'x\alpha) \ \wedge \ (Fy)(T'y\beta) \ \wedge \ (Fx)(Fy)(T'xT'y).$$

The first part of the formula $(Fx)(T'x\alpha)$ specifies x as an event that occurs at the beginning of the α computation, but nowhere else until α ends. The second part of the formula specifies that y is an event that occurs at the beginning of the β computation, but nowhere else until β ends. The third part $(Fx)(Fy)(T'xT'y)$ specifies that the first time x becomes true is no later than the first time y becomes true. The whole formula therefore specifies that α begins no later than β begins. The formula

$$(\exists x)(\exists y)[(Fx)(T'x\alpha) \ \wedge \ (Fy)(T'y\beta) \ \wedge \ (Fx)(Fy)(T'xT'y)].$$

is the same except that the events x and y used to communicate between α and β have been hidden, and are no longer part of the computation sequences.

It is useful to define some operations on partial interpretations in order to give a formal semantics of the language.

$I \ \wedge \ J$ is defined by

1. $|I \ \wedge \ J| = max(|I|, \ |J|)$ and
2. if $i \leq |I|, \ i \leq |J|$ then $|I \ \wedge \ J|_i = I_i \ \wedge \ J_i$;
   if $i \leq |I|, \ i > |J|$ then $|I \ \wedge \ J|_i = I_i$;
   if $i > |I|, \ i \leq |J|$ then $|I \ \wedge \ J|_i = J_i$.

IJ (the concatenation of I and J) is defined by

1. $|IJ| = |I| + |J| - 1$ where $\infty + x = x + \infty = \infty$, and
2. $IJ_i = I_i$ if $i < |I|$,
   $IJ_i = I_i \ \wedge \ J_1$ if $i = |I|$,
   $IJ_i = J_{i+1-|I|}$ if $i > |I|$.

Thus there is a one element overlap between I and J.

$I;J$ is concatenation without overlap, and is defined by

1. $|I;J| = |I| + |J|$ and
2. $(I;J)_i = I_i$ if $i \leq |I|$,
   $(I;J)_i = J_{i-|I|}$ if $i > |I|$.

$(\exists x)I$ is I with x and $\overline{x}$ deleted from all conjunctions.

$(Fx)I$ is I with $\bar{x}$ added to all conjunctions not containing x or $\bar{x}$. Thus x is made false except where a value for x is already specified.

$(Tx)I$ is I with x added to all conjunctions not containing x or $\bar{x}$. Thus x is made true except where a value for x is already specified.

The semantics of formulae are defined as follows:

$\Psi(p) = \{\mathrm{p}\}$ for propositional variable p
$\Psi(\bar{p}) = \{\bar{p}\}$
$\Psi(T) = \{\mathrm{T}\}$
$\Psi(F) = \{\mathrm{F}\}$
$\Psi(T^*) = \{T,\ T;T,\ T;T;T,\ \cdots,\ T^\infty\}$
$\Psi(\alpha \bigvee \beta) = \Psi(\alpha) \cup \Psi(\beta)$
$\Psi(\alpha \bigwedge \beta) = \{I \bigwedge J : I \in \Psi(\alpha),\ J \in \Psi(\beta)\}$
$\Psi(\alpha\ as\ \beta) = \{I \bigwedge J : I \in \Psi(\alpha),\ J \in \Psi(\beta),\ |I| = |J|\}$
$\Psi(\alpha;\beta) = \{I;J : I \in \Psi(\alpha),\ J \in \Psi(\beta)\}$
$\Psi(\alpha\beta) = \{IJ : I \in \Psi(\alpha),\ J \in \Psi(\beta)\}$
$\alpha^\infty \equiv \alpha \bigwedge (T;\alpha) \bigwedge (T;T;\alpha) \bigwedge (T;T;T;\alpha) \bigwedge \cdots$
$iter^*(\alpha,\ \beta) \equiv \bigvee_{k \geq 0} [\alpha\ as\ (T;\alpha)\ as\ (T^2;\alpha)\ as \cdots as\ (T^k;\alpha)\ as\ (T^{k+1};\beta)]$
where T^2 is $T;T$ and T^3 is $T;T;T$, et cetera.
$iter(*)(\alpha,\ \beta) \equiv \alpha^\infty \bigvee iter^*(\alpha,\ \beta)$
$\Psi(\exists x\alpha) = \{\exists xI : I \in \Psi(\alpha)\}$
$\Psi(Fx\alpha) = \{FxI : I \in \Psi(\alpha)\}$
$\Psi(Tx\alpha) = \{TxI : I \in \Psi(\alpha)\}$

3.1 Restrictions on the Quantifiers

Note that Fx and Tx are non-monotone. They must therefore be used with care. Let L be the language defined above. Let L_1 be L with the following restriction added:

The quantifiers Tx and Fx may only be applied to a formula α which is composed of

a) formulae in which x does not occur free
b) x
c) the connectives concatenation, ";", $\bigwedge$, as, $\exists y$, Fy, Ty for $y \neq x$.

If these restrictions are relaxed, then one can construct formulae which can count arbitrarily high, and the tableau like decision procedure does not work correctly. In fact, satisfiability of formulae in L may even be undecidable.

4. A Decision Procedure

The decision procedure for L_1 is complicated by the fact that eventualities do not behave in the usual way. The connective iter* is the only connective introducing an eventuality: iter*(α, β) implies that eventually β will be true (considering the interpretations as representing sequences of formulae which must be true at successive instants of time). Also, the formula $iter^*(\alpha, \beta);\gamma$ implies that eventually $\beta;\gamma$ will be true. We express an eventuality $\Diamond\delta$ as $\bigvee_k(T^k;\delta)$. We would like to find some eventuality δ such that

$$iter^*(\alpha, \beta);\gamma \equiv [iter(*)(\alpha, \beta);\gamma] \wedge \Diamond\delta$$

or such that

$$iter^*(\alpha, \beta);\gamma \equiv [iter(*)(\alpha, \beta);\gamma] \text{ as } \Diamond\delta$$

Now, letting δ be $\beta;\gamma$ will not work because we need to know that the β in $\beta;\gamma$ ends the same time $iter(*)(\alpha, \beta)$ ends. In fact, we have the following result:

Proposition 4.1. There does not exist a formula δ depending on α, β, γ such that for all α, β, γ,

$$iter^*(\alpha, \beta);\gamma \equiv [iter(*)(\alpha, \beta);\gamma] \wedge \Diamond\delta$$

or such that for all α, β, γ,

$$iter^*(\alpha, \beta);\gamma \equiv [iter(*)(\alpha, \beta);\gamma] \text{ as } \Diamond\delta$$

Proof. Let α be $PT^\infty \vee \bar{P}T^*$, let β be $\bar{P}$, and let γ be $\bar{P}^\infty$. Then $P;\bar{P}^\infty$ is a model of iter(*)(α, β);γ but not a model of iter*(α, β);γ. Therefore if such a δ exists, $\Diamond\delta$ must be false in the interpretation $P;\bar{P}^\infty$. However, $\bar{P}^\infty$ is a model of iter*(α, β) so $\Diamond\delta$ must be true in the interpretation $\bar{P}^\infty$. But if $\bar{P}^\infty \models \Diamond\delta$ then $T;\bar{P}^\infty \models \Diamond\delta$ hence $P;\bar{P}^\infty \models \Diamond\delta$, contradiction.

Because of this result, we give a decision procedure in which eventualities are treated in a nonstandard way. The decision procedure is graph oriented and model theoretic in nature; it may be possible to convert it to a syntactic nondeterministic tableau-like decision procedure. We first give another definition of the semantics of a formula of L.

Definition. A *standard temporal interpretation* is an infinite sequence of interpretations of propositional variables in a given set X of propositional variables. This is the same as

the "computation sequence" introduced in section 1.1.

Definition. If I is a partial interpretation c_1, c_2, c_3, $\cdots$, let $\Psi_1(I)$ be the set of standard temporal interpretations I' such that c_i is true in the i^{th} element of I' for $1 \leq i \leq |I|$.

Definition. If α is a formula of L_1 then $\Psi_1(\alpha)=\cup\{\Psi_1(I) : I \in \Psi(\alpha)\}$. This is the set of computation sequences satisfying at least one of the constraints in $\Psi(\alpha)$.

Note that α is consistent iff $\Psi_1(\alpha) \neq \emptyset$. For each formula α of L_1, the decision procedure constructs a graph G_α and provides a semantics $\Psi_1(G_\alpha)$ for G_α such that $\Psi_1(G_\alpha)=\Psi_1(\alpha)$. An iteration procedure applied to G_α decides if $\Psi_1(G_\alpha)=\emptyset$.

4.1 Graph construction

We construct graphs G_α such that G_α represents the set of computation sequences specified by α. The nodes in the graphs represent states, and the edges represent transitions from one state to another. Successive states in a path through the graph represent successive instants of time in a computation sequence. If there is an edge from node m to node n, then this edge specifies the events (propositional variables) that must occur or not occur in state m, if the transition from m to n is taken. Also, this edge may have a set of *eventualities*, representing events that must occur at some future time, and a set of *satisfied eventualities*, representing events that occur at state m and satisfy some previous eventuality. It is necessary to associate eventualities with nodes (actually, node basis elements, see below) in the graph. The reason is that if two processes are running concurrently, and they both require that some eventuality be satisfied, it is sometimes necessary to know for which of the two processes the eventuality has been satisfied. It may not be enough just for an eventuality to be satisfied; it may have to be satisfied at a particular time in the computation. For this reason, eventualities also contain information about which node they are associated with. We let the nodes in a graph be sets of elements of the *node basis*, which is some set disjoint from the set of eventualities. The reason for using subsets of the node basis as nodes of the graph, is that we can represent states s1 and s2 occurring concurrently by a node which is the union of the node basis elements of s1 and s2. However, if s1 and s2 have common elements, the semantics can become confused; eventualities are associated with node basis elements, and it may be necessary to distinguish which node the eventuality came from. Therefore we require that the node basis elements of s1 and s2 be disjoint whenever such a union is done. If this disjointness property does not already hold, then we define a *disjoining* operation and a *separation* property on graphs, which insure that the disjointness property does hold.

We define the graphs as follows. Each node is a subset of the *node basis* NB. One node of the graph is distinguished as the *initial node* of G, written init(G). The edges e have, in addition to an initial node init(e) and a final node fin(e), a set ev(e) of *eventualities* and a set se(e) of *satisfied eventualities*. Each eventuality and satisfied eventuality is an ordered pair <v, n> where n is a subset of NB and v is an *eventuality primitive*. The eventuality primitives are elements of the set EP; we assume the set EP is specified in some way and is disjoint from NB. An edge e also has a *propositional part* prop(e), which is a conjunction of propositional variables and their negations. Associated with each edge e of a graph G there is a *node relation* R_e between subsets of NB and subsets of NB. We consider such a relation R to be the set $\{<x, y> : R(x, y)\}$. Thus $\emptyset$ is the totally undefined relation. Also, for nodes m and n, let $g_{m,n}$ be the relation $\{<m, n>\}$ between m and n. We write the edge e as the tuple <init(e), fin(e), prop(e), ev(e), se(e), R_e>. Let N(G) be the nodes of graph G and E(G) be the edges. Each graph may have a distinguished END node. This indicates the end of the partial interpretation.

The graphs G_α for various α are defined as follows. We give the easy cases first.

If α is T, F, x, or $\bar{x}$ for propositional variable x, then G_α is defined by $N(G_\alpha) = \{m, END\}$, $init(G_\alpha) = m$, and $E(G_\alpha) = \{<m, END, \alpha, \emptyset, \emptyset, \emptyset>\}$. Here m is some singleton subset of NB. Note that f_e is totally undefined for the edge e of G.

If α is T^* then G_α is defined by $N(G_\alpha) = \{m, END\}$, $init(G_\alpha) = m$, and $E(G_\alpha) = \{<m, m, T, \emptyset, \emptyset, g_{m,m}>, <m, END, T, \emptyset, \emptyset, empty>\}$, where m is some singleton subset of NB.

$G_{\exists x\alpha}$ is G_α with x and $\bar{x}$ deleted from the propositional parts of all edges (and node relations unchanged).

$G_{\bar{x}x\alpha}$ is G_α with $\bar{x}$ added to the propositional parts of all edges not containing x or $\bar{x}$ in the propositional part (and node relations unchanged).

$G_{Tx\alpha}$ is G_α with x added to the propositional parts of all edges not containing x or $\bar{x}$ in the propositional part (and node relations unchanged).

Definition. Two graphs G_α and G_β are *separated* if they have no common node basis elements or eventuality primitives. That is, if $m_1 \in Nodes(G_\alpha)$ and $m_2 \in Nodes(G_\beta)$ then $m_1 \cap m_2 = \emptyset$, and if $<v_1, m_1>$ is an eventuality or satisfied eventuality of G_α, and $<v_2, m_2>$ is an eventuality or satisfied eventuality of G_β, then $v_1 \neq v_2$.

In the following definitions of graphs, assume that G_α and G_β are separated. If they are not, then assume node basis elements and eventuality primitives have been systematically renamed so that G_α and G_β are separated. Note that this also requires modifying the node relations in a corresponding way.

$G_\alpha \vee_\beta$ is defined as follows: Let m be a new node not in $N(G_\alpha)$ or $N(G_\beta)$. That is, m is {b} for some node basis element b which does not appear in G_α or G_β. Then

$N(G_\alpha \vee_\beta)=N(G_\alpha)\cup N(G_\beta)\cup\{m\}$,
$init(G_\alpha \vee_\beta)=m$, and
$E(G_\alpha \vee_\beta)=E(G_\alpha)\cup E(G_\beta)\cup$
$\{<m,\ n,\ C,\ ev,\ se,\ g_{m,\ n}> :$
$<init(G_\alpha),\ n,\ C,\ ev,\ se,\ R_1>\in E(G_\alpha)\}\cup$
$\{<m,\ n,\ C,\ ev,\ se,\ g_{m,\ n}> :$
$<init(G_\beta),\ n,\ C,\ ev,\ se,\ R_2>\in E(G_\beta)\}$.

$G_{\alpha;\beta}$ is defined as follows: $N(G_{\alpha;\beta})=N(G_\alpha)\cup N(G_\beta)$, $E(G_{\alpha;\beta})=E(G_\alpha)\cup E(G_\beta)$ except that edges of G_α of the form $<m,\ END,\ C,\ ev,\ se,\ R_e>$ are replaced by $<m,\ init(G_\beta),\ C,\ ev,\ se,\ g_{m,\ init(G_\beta)}>$. Also, $init(G_{\alpha;\beta})=init(G_\alpha)$.

$G_{\alpha\beta}$ is defined as follows: $N(G_{\alpha\beta})=N(G_\alpha)\cup N(G_\beta)$, $init(G_{\alpha\beta})=init(G_\alpha)$, and $E(G_{\alpha\beta})=E(G_\alpha)\cup E(G_\beta)$ except that an edge $<m,\ END,\ C,\ ev,\ se,\ R_e>$ of G_α is replaced by $\{<m,\ n,\ C \wedge D,\ ev',\ se,\ g_{m,\ n}> : <init(G_\beta),\ n,\ D,\ ev',\ se',\ R'>\in E(G_\beta)\}$.

For the remaining cases we need to define operations on edges. Suppose $e_1 \cdots e_k$ are edges, and either $fin(e_i)=END$ for all i or $fin(e_i)\neq END$ for all i. Then $as(e_1, \cdots, e_k)$ is the edge e such that

$init(e)=\cup_i init(e_i)$,
$fin(e)=\cup_i fin(e_i)$ unless $fin(e_i)=END$ for all i,
in which case $fin(e)=END$;
$prop(e)=\wedge_i prop(e_i)$,
$ev(e)=\cup_i ev(e_i)$,
$se(e)=\cup_i se(e_i)$, and
$R_e=\cup_i R_{e_i}$.

Also, $and(e_1, \cdots, e_k)$ is defined similarly except that the condition on $fin(e_i)$ and END need not hold, and if $fin(e_i)=END$ for all i then $fin(and(e_1, \cdots, e_k))=END$, but otherwise $fin(and(e_1, \cdots, e_k))=\cup\{fin(e_i) : fin(e_i)\neq END\}$.

$G_\alpha \wedge \beta$ is defined as follows:
$N(G_\alpha \wedge \beta)=\{m \cup n : m \in N(G_\alpha),\ n \in N(G_\beta)\} \cup N(G_\alpha) \cup N(G_\beta)$,
$E(G_\alpha \wedge \beta)=\{and(e_1, e_2) : e_1 \in E(G_\alpha),\ e_2 \in E(G_\beta)\}$, and $init(G_\alpha \wedge \beta)=init(G_\alpha) \cup init(G_\beta)$.

$G_{\alpha\ as\ \beta}$ is defined as follows: $N(G_{\alpha\ as\ \beta})=\{m \cup n : m \in N(G_\alpha),\ n \in N(G_\beta)\}$,
$E(G_{\alpha\ as\ \beta})=\{as(e_1, e_2) : e_1 \in E(G_\alpha),\ e_2 \in E(G_\beta),\ as(e_1, e_2)\ is\ defined\}$, and
$init(G_{\alpha\ as\ \beta})=init(G_\alpha) \cup init(G_\beta)$.

The remaining connectives are iter*, iter(*), and infloop. For these iteration primitives, it is necessary to require that some of the graphs be node disjoint. We say that a graph G is *node disjoint* if for any two distinct nodes m and n of G, $m \cap n=\emptyset$. We define the operation of *disjoining* a graph G_1 to produce an "equivalent" graph G_2 which is node disjoint. This consists essentially in renaming node basis elements in each node so that distinct nodes will be disjoint, and also adjusting eventualities, satisfied eventualities, and node relations in an appropriate way. Formally, for each node n we find a 1-1 function θ_n whose domain is n and such that for distinct nodes m and n of G_1, $\theta_m(m)$ and $\theta_n(n)$ are disjoint. Note that we are extending θ_m and θ_n to sets of elements in the node basis, in the usual way. Then G_2 is defined by $Nodes(G_2) = \{\theta_n(n) : n \in Nodes(G_1)\}$, $Init(G_2) = \theta_{Init(G_1)}(Init(G_1))$, and $Edges(G_2) = \{<\theta_m(m), \theta_n(n), C, ev', se', R'> : <m, n, C, ev, se, R> \in Edges(G_1)\}$, where $ev' = \{<v, \theta_m(r)> : <v, r> \in ev\}$, $se' = \{<v, \theta_m(r)> : <v, r> \in se\}$, and $R' = \{<\theta_m(z), \theta_n(y)> : <z, y> \in R\}$. It is this operation of disjoining graphs that leads to the nonelementary performance of the satisfiability algorithm. It is not really necessary to do this operation in all cases, but we specify it for all cases for simplicity. When defining graphs for $iter^*(\alpha, \beta)$, $iter(*)(\alpha, \beta)$, and $infloop(\alpha)$, we assume that α and β are separated as before, and also that α is node disjoint.

$G_{iter(*)(\alpha, \beta)}$ is defined using $G_\alpha \vee \beta$ in the following way:
$N(G_{iter(*)(\alpha, \beta)})=\{US : S$ is a subset of $N(G_\alpha \vee \beta)$ not containing END and containing at most one node in $N(G_\beta)\} \cup \{END\}$, $init(G_{iter(*)(\alpha, \beta)})=\{init(G_\alpha \vee \beta)\}$, and $E(G_{iter(*)(\alpha, \beta)})=E1 \cup E2$ where E1 and E2 are as follows:

$E1=\{as(e_1, \cdots, e_k, <init(G_\alpha \vee \beta), init(G_\alpha \vee \beta), T, \emptyset, \emptyset, \theta_{init(G_\alpha \vee \beta)}, init(G_\alpha \vee \beta)>) :$ this is defined and $e_i \in E(G_\alpha \vee \beta)$ all i, no e_i in $E(G_\beta)$, $init(e_1)=init(G_\alpha \vee \beta)$, and $init(e_i)$ are all distinct$\}$

$E2=\{as(e_1, \cdots, e_k) :$ $as(e_1, \cdots, e_k)$ is defined and $e_i \in E(G_\alpha \vee \beta)$ with exactly one e_i in $E(G_\beta)$ and $init(e_i)$ are all distinct$\}$

The edges E1 represent repeated iterations of α and the edges E2 represent the time after β has begun.

$G_{iter*(\alpha,\ \beta)}$ is the same as $G_{iter(*)(\alpha,\ \beta)}$ except that there is a new eventuality $<v,\ init(G_\alpha \vee \beta)>$ added to edges e such that $init(G_\alpha \vee \beta) \in init(e)$ and $init(G_\alpha \vee \beta) \in fin(e)$. Also, edges e in $G_{iter*(\alpha,\ \beta)}$ have satisfied eventuality $<v,\ init(G_\alpha \vee \beta)>$ added if $init(G_\alpha \vee \beta) \in init(e)$ but not $init(G_\alpha \vee \beta) \in fin(e)$. Intuitively, v represents the eventuality that β must eventually be true.

Finally, $G_{inf loop(\alpha)}$ is like $G_{iter(*)(\alpha,\ \beta)}$ except that the edges in E2 and nodes having subsets in $N(G_\beta)$ are omitted.

4.2 Semantics of graphs

With a graph G as above we associate a semantics $\Psi_1(G)$ representing the set of standard interpretations satisfying G. A standard interpretation I is in $\Psi_1(G)$ if there is an infinite sequence $e_1,\ e_2,\ \cdots$ of edges of G such that

a) $init(e_1) = init(G)$
b) $fin(e_i) = init(e_{i+1})$ for all $i \geq 1$
c) $I_i \models prop(e_i)$ for all i, where I_i is the interpretation I specifies at the i^{th} instant of time
d) all eventualities in the path $e_1,\ e_2,\ \cdots$ are satisfied.

The satisfaction of eventualities is defined in a nonstandard way. We extend the node relations R to eventuality relations by $R(<v, m>, <w, n>)$ iff $v = w$ and $R(m, n)$. An eventuality ev in $ev(e_i)$ is satisfied in the path if there exist $ev_i,\ ev_{i+1},\ \cdots,\ ev_{i+k}$ such that $ev = ev_i$ and $ev_{i+k} \in se(e_{i+k})$ and for all j, $0 \leq j < k$, $R_{e_{i+j}}(ev_{i+j},\ ev_{i+j+1})$. Thus the eventualities may be transformed at each edge in the path, and they are satisfied if at some future time, some such transformed eventuality is satisfied. We claim that $\Psi_1(\alpha) = \Psi_1(G_\alpha)$ for all formulae α in L_1. Thus the semantics of graphs agree with those of the formulae of the low level language.

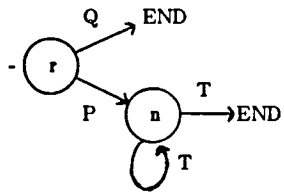
4.3 Example

We now give an example of a formula and the graph constructed from it. First we give an intuitive explanation of the construction for iter*(α, β). Consider the graph $G = G_\alpha \vee \beta$ for $\alpha \vee \beta$. We construct the graph for $G_{iter*(\alpha,\ \beta)}$ from G by permitting the nodes of G to have "markers." These markers can travel along edges of G. The current state of the graph is determined by which nodes have markers on them. At the start, only the initial node of G has a marker. Thereafter, markers travel along edges in one of two ways: a) The marker from the initial node travels to some node in G_α along an edge, and also reproduces a

copy of itself which remains on the initial node. All other markers travel along some edge; if there is an edge e and a marker on node init(e), this marker can travel to node fin(e). This marker is then removed from init(e). A marker can only travel to one other node in one time instant (except that the marker on the initial node also may reproduce a copy of itself on the initial node). If a marker is on a node with no outgoing edges, this marker is deleted; this will happen for markers on the END node, for example. b) The marker from the initial node travels to some node in G_β, but does not reproduce a copy of itself on the initial node. Other markers may travel along edges as in a).

In both cases, if a node ends up with more than one marker on it, all but one of these markers are removed. The collection of marked nodes may be considered as the "current node" of the graph G. A transition as in a) corresponds to the part of the iteration in which α is being repeated; a transition as in b) corresponds to the beginning of the β part of the iteration. Let us call these transitions *a-transitions* and *b-transitions*, respectively. These transitions are the edges of G. The propositional part of such a transition is the conjunction of the propositional parts of the edges of $G_{\alpha \vee \beta}$ traversed during the transition. The a-transitions have a new eventuality associated with them; the b-transitions have this eventuality satisfied. This corresponds to the fact that there must eventually be a b-transition. In the formal definition of G, the nodes of G are unions of the node basis elements in the marked nodes of $G_{\alpha \vee \beta}$, with the END node ignored in such unions. However, if only END is marked, this corresponds to the END node of G. The graph $G_{iter(\alpha), \beta)}$ is similar except that there is no eventuality for a b-transition to occur. The graph G_{α^∞} is similar except that there are no b-transitions and no eventuality for a b-transition to occur.

Consider the formula iter*(P, Q). Since all the partial interpretations P must end at the same time as Q does, this formula is equivalent to Q. To get a nontrivial use of iter*, we need to use the T^* constant. Consider the formula iter*(PT^*, Q). This is equivalent to $\bigvee_i P^i;Q$. To represent graphs pictorially, we draw a node as a circle or oval containing its node basis elements. The edges are drawn as arrows from their initial node to their final node. The propositional parts of edges are drawn next to the edges. The eventualities, satisfied eventualities, and eventuality transforms are not given in the picture but are specified separately for simplicity. The initial node is indicated by a minus sign next to the node; the end node, if any, is indicated by END. This graph, with nodes deleted that are not reachable from the initial node, is as follows. Note that it is also permissible to delete edges whose propositional part is contradictory.



Graph for the formula $PT^* \vee Q$

Now, whenever there is an a-transition, P will be true and markers will remain on nodes {r} and {n} and possibly END; when there is a b-transition, Q will be true. Thus this graph specifies $\bigvee_i P^i;Q$.

4.4 Iteration method

An iteration method is applied to the graph G_α to determine if α is satisfiable. The idea is to repeatedly delete edges having eventualities that cannot be satisfied by any path in the graph, and to delete nodes having no outgoing edges (except for the END node). Also, edges whose propositional part is contradictory may be deleted. The formula α is satisfiable iff the initial node of G_α remains after this iteration is completed. When searching for paths satisfying eventualities, the eventuality transforms have to be considered as indicated above. The techniques described in [5] for obtaining decision procedures for combinations of temporal logic and other specialized theories, can also be applied. Finally, as in [5], it is possible to permit an arbitrary combination of state variables, whose values change with time, and free variables, whose values do not. For a discussion of these concepts see [5].

4.5 Complexity

This decision procedure is of nonelementary complexity since $|N(G_{iter*(\alpha,\ \beta)})|$ is exponential in $|N(G_\alpha)|$, and the node disjoining procedure can then lead to an exponential number of node basis elements in the graph. There may be an arbitrarily deep nesting of the iter* and iter(*) and infloop connectives, leading to nonelementary behavior. The following example may give some syntactic insight as to why the closure of the formulae in L_1 can be so large: Let A_1 be the formula

$$infloop(\exists x(iter(*)(\alpha_1,\ \beta_1)\ as\ \cdots\ as\ iter(*)(\alpha_n,\ \beta_n)))$$

The closure of this formula will include formulae of the form γ_1 *as* γ_2 *as* $\cdots$ *as* γ_k where γ_i is of the form $\exists z(\delta_1$ *as* $\cdots$ *as* $\delta_n)$ and δ_i is in the closure of $iter(*)(\alpha_i, \beta_i)$. If the closure of $iter(*)(\alpha_i, \beta_i)$ has at least two formulae for all i, then there can be 2^n formulae γ_i and the closure of A_1 can contain formulae 2^n times as large as formulae in the closure of $iter(*)(\alpha_i, \beta_i)$. Similarly, let A_2 be

$$infloop((iter(*)(\alpha_1, \beta_1)\ as\ \cdots\ as\ iter(*)(\alpha_n, \beta_n));\gamma)$$

The closure of A_2 includes formulae of a similar form except that γ_i is of the form $(\delta_1$ *as* $\cdots$ *as* $\delta_n);\gamma$. Finally, let A_3 be the formula

$$infloop(\gamma \wedge (iter(*)(\alpha_1, \beta_1)\ as\ \cdots\ as\ iter(*)(\alpha_n, \beta_n)))$$

The closure of A_3 is similar except that γ_i is of the form $\gamma \wedge (\delta_1$ *as* $\cdots$ *as* $\delta_n)$. Intuitively, the closure of a formula A represents the set of formulae B which may be true at future times if A is true now.

5. Interval logic

We now give some examples to illustrate how the interval logic of Schwartz, Melliar-Smith, and Vogt[7] may easily be translated into the low level language. In fact, this translation was the original motivation for developing the low level language, since it seemed much simpler to program a decision procedure for the low level language than for interval logic.

Interval logic was developed to permit convenient reasoning about intervals of time. An interval *formula* is a formula of interval logic and has a Boolean truth value in any interpretation. An interval *term* is an expression of interval logic whose value is a time interval. Without going into details, let Expr(α)(z) be the translation of interval formula α in context z, and let Int(α)(x y z d) be the translation of interval term α in context z, where the interval begins at x and ends at y. Here d is the direction in which you are looking for the interval, and may be F (forward) or B (backward). For our purposes, x, y, and z are propositional variables which intuitively denote the next state in which they are true. We give a few translations; for an explanation of the notation see [7].

$$Expr([I]\alpha)(z) = \exists x\, \exists y Int(I)(x, y, z, F) \wedge Fx(T^* x Expr(\alpha)(y))$$

$$Int(\alpha \rightarrow \beta)(x, y, z, d) = \exists w Int(\alpha)(w, x, z, d) \wedge \exists v Fx(T^* x Int(\beta)(v, y, z, F))$$

$$Expr(p)(z)=\text{if } z=\infty \text{ then } pT^{\infty} \text{ else } p \text{ iter}*(T', z)$$

To decide if an interval formula α is valid, we can convert $\neg\alpha$ to a normal form β and test if $Expr(\beta)(\infty)$ is satisfiable.

6. A PSPACE sublanguage

We originally intended to use the language L_1 to show that interval logic has a PSPACE decision procedure. For this, it is necessary to find a sublanguage of L_1 which can be decided in PSPACE and into which all interval logic expressions may be translated. We have been unable to do this. It seems that preventing α from containing any iteration connectives in expressions of the form $iter*(\alpha, \beta)$, $iter(*)(\alpha, \beta)$, and $infloop(\alpha)$ would help, but this prevents certain interval logic formulae from being expressed. However, this does not mean that interval logic is not in PSPACE.

7. Other temporal logics

It would be interesting to compare the expressive power of L_1 with other temporal and process logics. One can easily encode the usual discrete linear time temporal logic into L_1 by expressing Until(x, y) as iter(*)(x, y) (with no eventuality implied), "next time x" as T;x, "henceforth x" as infloop(x), "eventually x" as iter*(T', x), propositional variables p as pT', $\neg p$ as $\neg pT'$, and Boolean connectives $\wedge$ and $\vee$ as themselves. This requires pushing negations to the bottom, but it is possible to do this; the only slightly hard case is negating "until".

The semicolon operator seems similar to the "chop" operator of dynamic logic [2]; the interval logic of Moszkowski[4] has a slightly similar semicolon operator but is undecidable. We now consider a branching time version of the low level language.

7.1 Branching time syntax

Expressions may be *path expressions* or *state expressions*. Intuitively, the models are trees, and path expressions refer to paths in the tree while state expressions refer to the whole tree. All the previous connectives are still used; they map path expressions to path expressions. Thus if α and β are path expressions, so are $\alpha;\beta$, $\alpha\beta$ et cetera. In addition, if α is a path expression, then $A\alpha$ and $E\alpha$ are state expressions. Also, if α and β are state expressions, then $\alpha \wedge \beta$ and $\alpha \vee \beta$ are state expressions. If α is a state expression, then $\exists x\alpha$, $Fx\alpha$, and

$Tx\alpha$ are state expressions. Finally, if x is a propositional variable or its negation, T, or F, then x may be regarded as a state expression. Thus we are overloading certain operators; for example, $\exists x$ maps path expressions to path expressions and state expressions to state expressions. Finally, any state expression can be viewed as a path expression.

7.2 Branching time semantics

The semantics is defined analogously to that for the linear time logic. A *literal* is a propositional variable or its negation. A *partial path interpretation* is a triple (V, L, P) where V is a tree, L is a labeling function mapping nodes of V to conjunctions of literals, and P is a finite or infinite path of V starting at the root and not crossing any node more than once. A *partial state interpretation* is a pair (V, L) with V and L as above. With each path expression α we associate a set $\Psi(\alpha)$ of partial path interpretations, and with each state expression α we associate a set $\Psi(\alpha)$ of partial state interpretations. The expression α is *consistent* if some member of $\Psi(\alpha)$ is a tree having no contradictory conjunctions. Let us call a path P of V as above a *prefix path* of V. By convention, if L is a labeling function of a tree V, and N is a node not in V, then L(N) = T. The semantics is defined as follows.

$(V, L, P) \in \Psi(x)$ for x a literal, T, or F if L(N) = x where N is the root node of V and L(M) = T (True) for $M \neq N$ and P = {N}.

$(V, L, P) \in \Psi(T^*)$ if L(N) = T for all N and P is any prefix path of V.

$(V, L, P) \in \Psi(\alpha \wedge \beta)$ if there exists L1, L2, and P1 such that P1 is a prefix of P and $L \equiv L1 \wedge L2$ and either $(V, L1, P1) \in \Psi(\alpha)$ and $(V, L2, P) \in \Psi(\beta)$ or $(V, L1, P) \in \Psi(\alpha)$ and $(V, L2, P1) \in \Psi(\beta)$.

$(V, L, P) \in \Psi(\alpha \text{ as } \beta)$ if there exist L1, L2 such that $L \equiv L1 \wedge L2$ and $(V, L1, P) \in \Psi(\alpha)$ and $(V, L2, P) \in \Psi(\beta)$.

$(V, L, P) \in \Psi(\alpha \vee \beta)$ if $(V, L, P) \in \Psi(\alpha)$ or $(V, L, P) \in \Psi(\beta)$.

$(V, L, P) \in \Psi(\alpha;\beta)$ if there exist L1, L2, P1, P2 such that $L \equiv L1 \wedge L2$, P = P1 ; P2, $(V, L1, P1) \in \Psi(\alpha)$, and V has a subtree V1 such that $(V1, L2, P2) \in \Psi(\beta)$.

$(V, L, P) \in \Psi(\alpha\beta)$ if there exist L1, L2, P1, P2 such that $L \equiv L1 \wedge L2$, P = P1 P2 (that is, P1 and P2 have a node in common), $(V, L1, P1) \in \Psi(\alpha)$, and V has a subtree V1 such that $(V1, L2, P2) \in \Psi(\beta)$.

$$infloop(\alpha) \equiv \alpha \wedge T;\alpha \wedge T^2;\alpha \wedge \cdots \wedge T^k;\alpha \wedge \cdots$$

$$iter*(\alpha, \beta) \equiv \bigvee_j [\alpha \wedge T;\alpha \wedge \cdots \wedge T^j;\alpha \wedge T^{j+1};\beta]$$

$$iter(*)(\alpha, \beta) \equiv infloop(\alpha) \vee iter*(\alpha, \beta)$$

$(V, L, P) \in \Psi(\exists x \alpha)$ if there is a function L1 such that $(V, L1, P) \in \Psi(\alpha)$ and L is identical to L1 except that L deletes x and $\bar{x}$ from nodes in P.

$(V, L, P) \in \Psi(Fx\alpha)$ if there is a function L1 such that $(V, L1, P) \in \Psi(\alpha)$ and L is identical to L1 except that L adds $\bar{x}$ to nodes of P not containing x or $\bar{x}$.

$(V, L, P) \in \Psi(Tx\alpha)$ if there is a function L1 such that $(V, L1, P) \in \Psi(\alpha)$ and L is identical to L1 except that L adds x to nodes of P not containing x or $\bar{x}$.

$(V, L, \{N\}) \in \Psi(\alpha)$ if α is a state expression, $(V, L) \in \Psi(\alpha)$, and N is the root of V. (This converts state expressions to path expressions.)

If α is a literal, T, F, or T^*, regarded as a state expression, then $\Psi(\alpha)$ is as above except that the path part of interpretations is omitted.

$(V, L) \in \Psi(\exists x \alpha)$ for state expression α if there is a function L1 such that $(V, L1) \in \Psi(\alpha)$ and L is identical to L1 except that L deletes x and $\bar{x}$ from *all* nodes.

The semantics of $Fx\alpha$ and $Tx\alpha$ for state expressions α are defined similarly, modifying *all* conjunctions, not just those on some path.

$(V, L) \in \Psi(A\alpha)$ if for all infinite prefix paths P of V, P has a prefix P1 such that $(V, L, P1) \in \Psi(\alpha)$.

$(V, L) \in \Psi(E\alpha)$ if for some prefix path P of V, $(V, L, P) \in \Psi(\alpha)$.

We do not have any information about the decidability of this branching time version of the low level language, except that the satisfiability problem is at least as hard as that of L_1 since L_1 is a subset of the language. Also, it appears that L_1 is of nonelementary complexity.

7.3 Regular expressions

We could add the star operator α^* to the linear and branching time logics to get a

formalism including regular expressions as a syntactic subset. However, this was not necessary for our purposes.

8. Executable specifications

In the style of Manna and Wolper [3] and the "path expressions" of Campbell and Habermann[1], we can use the linear time low level language to construct programs having a specified behavior. Given a low level formula α, we construct the graph G_α which represents the set of models of α; this graph can then be regarded as a program. By adding suitable fairness constraints to certain nodes of G_α, we obtain a program which satisfies all eventualities of α and thus behaves as specified by α. In this way we might consider automatically constructing concurrent programs from their specifications.

9. Acknowledgements

This work was initiated by many conversations with Michael Melliar-Smith, Richard Schwartz, and Fritz Vogt at SRI International, relating to the possible implementation of a decision procedure for interval logic. The comments of Richard Schwartz helped in the preparation of this paper.

10. References

1. Campbell, R. and Habermann, A., The specification of process synchronization by path expressions, Lecture Notes in Computer Science, Springer-Verlag, Volume 16, 1974, pp. 89-102.

2. Harel, D, First-order Dynamic Logic, Springer-Verlag Lecture Notes, No. 68, 1979.

3. Manna, Z. and Wolper, P., Synthesis of communicating processes from temporal logic specifications, 253-281 in Proceedings of the Workshop on Logics of Programs, 1981.

4. Moszkowski, B., A temporal logic for multi-level reasoning about hardware, Technical Report STAN-CS-82-952, Computer Science Department, Stanford University, 1982.

5. Plaisted, D., A decision procedure for combinations of propositional temporal logic and other specialized theories, Appendix B of this report.

6. Pratt, V., Semantical considerations on Floyd-Hoare logic, Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science (1976)109-121.

7. Schwartz, R., Melliar-Smith, P. M., and Vogt, F., An interval logic for higher level temporal reasoning. NASA Contractor Report 172262.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
172262		

4. Title and Subtitle	5. Report Date
An Interval Logic for Higher-Level Temporal Reasoning	September 1983
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
R.L. Schwartz, P.M. Melliar-Smith, F.H. Vogt, D.A. Plaisted	SRI Project 4616
	10. Work Unit No. Task 3
9. Performing Organization Name and Address SRI International 333 Ravenswood Avenue Menlo Park, CA 94025	11. Contract or Grant No. NAS1-17067
	13. Type of Report and Period Covered Task Final Report
12. Sponsoring Agency Name and Address NASA Langley Research Center Hampton, VA 23665	14. Sponsoring Agency Code

15. Supplementary Notes

16. Abstract

Prior work explored temporal logics, based on classical modal logics, as a framework for specifying and reasoning about concurrent programs, distributed systems, and communications protocols, and reported on efforts using temporal reasoning primitives to express very high level abstract requirements that a program or system is to satisfy. Based on experience with those primitives, this report describes an Interval Logic that is more suitable for expressing such higher level temporal properties. The report provides a formal semantics for the Interval Logic, and several examples of its use. A description of decision procedures for the logic is also included.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
specification, verification, Asynchronous, temporal logic	

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	83 pages	