This is to certify that the

thesis entitled

TEMPORAL SPECIFICATIONS FOR DISTRIBUTED SYSTEMS

presented by

William Eugene McUmber

has been accepted towards fulfillment
of the requirements for

M.S. degree in Computer Science

Major professor

Date 6/1/93

Temporal Specification Systems for Distributed Systems

By

William E. McUmber

A Thesis

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Master of Science

Department of Computer Science

May, 1993

Betty H. C. Cheng, Advisor

ABSTRACT

Temporal Specification Systems for Distributed Systems

By

William E. McUmber

This work explores specification techniques that have the capability of handling statements about occurrences in time. The work is oriented towards problems found in conjunction with distributed systems. Distributed system problems are characterized by many processes executing concurrently, often with important interoperating timing constraints, and often without the benefit of synchronization from a global clock. The usual approach, state based temporal systems, assume some form of clocking of the system and requires either a synchronous or interleaving state transition model. After giving a survey of current specification methods, this thesis presents two alternative specification systems based on modal-temporal logic. The first approach is based on an extension to the Larch specification language. The model proposed is based on points in time and avoids the use of clocks. The second approach uses two sets of modal operators, one for time and another for possibility, as a means of specifying event sequences without state transitions in either linear or branching time. In this approach, use of the modal *possibility* operator replaces existential quantification found in state-based branching time formulations of temporal logic. The second approach disallows instants in time and is based on intervals.

To my wife, Cheryl and our sons Weston and Robert.

ACKNOWLEDGMENTS

I am indebted to many people without whom I could not have completed this work. First and foremost, I am indebted to my advisor, Betty H. C. Cheng, whose direction has led me to this point. It is Dr. Cheng that convinced me of the importance of predicate logic for program specifications, and it was her suggestion to carefully study Larch that influenced much of my thinking about the nature of a specification language. In addition, her guidance and patience through many edits and her pointed questions were invaluable. I also wish to thank Phillip McKinley who introduced me to the formal aspects of distributed systems. From him I first understood the complexity inherent in problems dealing with mutual exclusion, liveness and distributed systems in general.

I wish to thank the members of my committee, John Geske, Phillip McKinley, and Betty H. C. Cheng who provided many helpful comments.

I wish to extend my thanks to George Stockman and Lora Mae Higbee who lead the way through a maze of procedures and requirements that ultimately permitted the completion of this Thesis.

Finally, the personal and emotional support provided by my loving wife, Cheryl, and our sons, Wes and Robbie can not be measured. The many hours spent on this work was, in many ways, their time, and I am indebted to them for it.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Formal methods focus a software development effort on an accurate and precise specification of *what* a software system or component is to achieve. This type of specification is usually referred to as a *formal specification* and is expressed in a precise mathematical notation defined by a *formal specification language*. Specifications can be written in natural languages, semi-formal notations (analogous to pseudo-code for programs), and in formal languages. We define a *formal language* as one in which the syntax and semantics are pre-defined in an unambiguous way. Syntax is usually defined through a grammar.[1] Semantics are harder to define precisely but, at a minimum, the intention of language constructions can be described through logical formulas accompanied by a small amount of clarifying text. An example can be found in [1] where Larch, both described and used later, is formally defined. The intention of formality in a language is to provide the interpretation of any statement in the language by a set of defined rules. The advantage of a formal language as defined is that its assertions are unambiguous and have the potential of being manipulated by programs [2, 3, 4]. Many formal languages are already in use for software specification. A frequent basis for these languages is formal logic [2, 5, 6].

Software for distributed systems have the same problems of correct and accurate

[1] Grammars can be "ambiguous" in the sense that the same language construct may possibly be derived from two or more different sets of production rules. Since determination of ambiguity in grammars more complex than regular grammars (*i.e.*,context free and context sensitive grammars) is undecidable, we can never be sure, in the general case, that the formal language grammar, and hence the formal language, has unambiguous syntax. The intended meaning of "unambiguous" is that the rules for the formal language are defined as best as possible before use and that the semantics are unambiguous.

behavior as do other forms of software, but it also has the added burden of inter-process timing relations. By their nature, distributed systems consist of many processes running in a computationally diverse environment. Often, the processes must interact in complex ways without interfering with each other. Two broad classes of inter-process problems, called *liveness* and *mutual exclusion*, have been widely studied [7, 8, 9, 10]. Both problems contain at their core the sub-problem of inter-process timing constraints.

Temporal specification systems have been widely studied but often only in connection with real-time systems [11, 12, 13]. Unlike real-time systems where a global clock either exists or can be assumed, distributed systems normally have no global clock. Provision of a global clock is equivalent to solving distributed consensus problems which is known to be difficult and, in some situations, impossible [14, 15, 16]. Most real-time specification systems use state progressions as the means of marking the passage of time. Either an interleaved or sequential model of state transition is usually assumed. Since a distributed environment has no global clock and processors run at various speeds, neither model of state transition matches reality very well.

The goal of this work is to investigate and formulate a formal specification methodology that has sufficient expressive power to specify time-based system behaviors. The specification system must be expressive enough so that it does not rely on state transitions alone to describe time. The specification language should be based on logic and algebraic in nature. The reason for basing the language and method on logic is that both modal logic and first order logic have been extensively studied and are well understood. We specifically address formal languages that are algebraically based for two reasons:

1. We consider algebraic languages to be better suited to automated tools and existing automatic rewrite systems than other approaches.

2. Algebraic languages are declarative in nature, and as such, are more likely to be implementation-detail independent. The intention is to specify the required *behavior* rather than how the behavior is obtained.

It is likely that verification of actual software specifications, especially those involving

timing constraints such as liveness, deadlock, and mutual exclusion, can only be practically addressed by automated systems. If this is true, the specification system will have to be embodied in automated tools in order to have significant impact.

Finally, the specification system should be as intuitive as the constraints of the problem allow. Initial problem descriptions always originate with humans. The success of the methodology will be determined by the degree to which the specifier is relieved of details while retaining enough precision to describe exactly what should happen.

We meet these goals through several new developments. First, we introduce a temporal model into an existing algebraic specification language. The temporal model includes development of an HOLD predicate for describing invariant properties. We first encountered the idea of HOLD in Allen's [17, 18, 19] work but we have extended the use and definition beyond the original concept.

Second, we introduce a new method of specifying alternate future sequences by employing dual modalities. Temporal specification systems frequently have difficulty with alternate future execution sequences, generally known as the *branching time problem.* The most common solution is to use quantification over modal operators to describe multiple future sequences, as in [20]. This approach has its difficulties, however [21, 22]. Our approach avoids quantification over execution sequences, and consequently, the difficulties quantification introduces.

The remainder of the thesis is organized as follows. Chapter 2 introduces modal and temporal logic, both of which are used throughout this work. Chapter 3 surveys a variety of specification methods capable of describing timing constraints. Chapter 4 presents our extensions to an algebraic language with temporal logic. Chapter 5 introduces a new approach to temporal specification based on intervals. This approach uses the standard modal-temporal operators for time but introduces the use of another modal operator as a replacement for quantification for branching time descriptions. Finally, Chapter 6 contains concluding remarks and discusses future work.

CHAPTER 2

Modal Logic

Many commonly used temporal logic systems are based on modal logic. Complete descriptions of modal logic can be found in [23, 24, 25], but a brief introduction of fundamental concepts is provided in this section in order to provide a basis for temporal logic discussions later.

Modal logic augments standard quantified first-order predicate logic by introducing operators for describing what is *possible* and *necessary* in addition to what *is* or *is not*. Just as quantifiers specify "all" or "some" in a formula, modal operators permit specification of necessity and possibility. In English, there is a clear distinction between "Today is Saturday" and "It is possible that today is Saturday." The former's truth or falsity is determined by the value of "today". The latter's truth is determined by the values of "todays" throughout multiple contexts where the statement "Today is Saturday" can be made. Contrast the two statements when the value of "today" is not Saturday. In elementary logic, the statement "today is Saturday" is clearly false on Tuesday, while "it is possible that today is Saturday" is true. In logic terms, for a given predicate P, [possible] P can be true when P, in a particular setting, is false.

The intuitive idea introduced above is one of a collection of possible *worlds*, including our present one. In each world, each logic sentence can be evaluated to true or false. Thus, a world is an instantiation of a set of variables and value assignments particular to this world. A sentence is said to be *necessary* if it is true in all worlds. A sentence is said to

be *possible* if it is true in at least one world. The difference between the earlier statements "It is possible that today is Saturday" and the first order statement "It is not the case that for all days, no day is Saturday" lies in the object of quantification. In the former, modal logic requires quantification over all worlds, or possible settings of the statement "today is Saturday." In the latter statement, quantification ranges over all days, but only within one setting, that is, one world.

Given a sentence A, the notation $\Box A$ is used to indicate A's necessity. The notation $\Diamond A$ is used to indicate A's possibility. Given an infinite collection of worlds, W, P abbreviates an infinite sequence $P_0, P_1, P_2, \ldots$ of subsets of W. The pair $\Lambda = (W, P)$ represents a model. Given a sentence A, and a world $p_n \in P$ in the model, the interpretation of

$$\models^{\Lambda}_{p_n} A \tag{2.1}$$

is A is true at world p_n in Λ.

Several tautologies for world models can be written as follows:

1. $\models^{\Lambda}_{p_n}$ True.
2. $\neg \models^{\Lambda}_{p_n}$ False.
3. $\models^{\Lambda}_{p_n} \Box A \Longrightarrow$ for every $p_j \in \Lambda, \models^{\Lambda}_{p_j} A$
4. $\models^{\Lambda}_{p_n} \Diamond A \Longrightarrow$ for some $p_j \in \Lambda, \models^{\Lambda}_{p_j} A$

Statement (1) means that the logic constant *True* is always true in all worlds. Likewise, Statement (2) means *False* is never true. Statement (3) defines *necessary* as requiring A to be true in all worlds in the subset P of worlds in the model. Statement (4) similarly defines *possible* as requiring A to be true in some world. Given the notation $\models A$ means A is true in all models in all worlds, if A is a tautology, then $\models A$ and $\Box A$ are true for all worlds.

If we take $\Box$ as a primitive then we can define $\Diamond$ as

$$\Diamond A = \neg \Box \neg A \tag{2.2}$$

denoting what is possibly true cannot always be not true. The relationship is symmetrical, that is,

$$\Box A = \neg\Diamond\neg A \tag{2.3}$$

indicating what is necessarily true cannot be not possible.

2.1 Temporal Logic Systems

The majority of temporal logic operators are based on systems derived from modal logic. As many as 15 different modal systems [25] can be constructed depending on the axioms and inference rules chosen for the system. When choosing a modal system, and especially when giving the modal system a temporal semantic interpretation, the path is strewn with many traps that can lead to inconsistent systems or systems that allow statements that make no sense in the real world.

All modal systems assume the fundamentals of logic including standard definitions for disjunction ($\vee$), conjunction ($\wedge$), Modus Ponens, De Morgan's laws, *etc.* To these are added definitions and axioms that describe the characteristics of *necessity* and *possibility* as described in the previous section. We follow convention and use the notation $\Box P$ for "P is necessary" and $\Diamond P$ for "P is possible."

Following the naming convention of Cresswell and Hughes [24], the simplest of the modal systems is called T. T consists of definitions **TDEF1** and two axioms, **TA1** and **TA2**:

TDEF1 $\Box p \equiv \neg\Diamond\neg p$

TA1 $\Box p \Longrightarrow p$

TA2 $\Box(p \Longrightarrow q) \Longrightarrow (\Box p \Longrightarrow \Box q)$

where p and q are any well-formed formulas. Definition **TDEF1** is basic to all modal and temporal systems and states "what is possible (henceforth, always) true is not possibly (eventually, sometimes) not true." Axiom **TA2** permits $\Box$ to commute with implication.

System S4 [24] is obtained by adding only one additional axiom to T:

S4A1 $\Box p \Longrightarrow \Box\Box p$

S4A1 enables the reduction of compounded modal operators but also produces a profound difference between System T and S4. The combined use of **TDEF1** and **S4A1** allow any sequence of compounded modal operators to be reduced to one of the following:

1. –none– (no operator)
2. $\Box$
3. $\Diamond$
4. $\Box\Diamond$
5. $\Diamond\Box$
6. $\Box\Diamond\Box$
7. $\Diamond\Box\Diamond$

When combined with the negated forms, only 13 different compounded modal operators (including *no* modal operator) are possible in S4. Contrast this with an infinite number of compounded modal operators in T, which has no reduction rule.

System S5 [24] is obtained from T by adding:

S5A1 $\Diamond p \Longrightarrow \Box\Diamond p$

Rule **S5A1** is similar to **S4A1**, enabling reduction of compounded modal operators. In contrast to **S4A1**, **S5A1** allows reduction of any sequence of modal operators to no more than one operator. Compounded operators can be reduced in S5 to either no operator, $\Diamond$, or $\Box$. Although powerful reductions are possible in system S5, we will see below that such reduction power introduces semantic difficulties when S5 is used in a temporal context.

S4 and S5 are simple variants of the T system, but each has distinct semantics. For example, consider the temporal implications of Axiom **S5A1**. While a tool to reduce any long sequence of modal operators may be convenient, the tool also permits the following reduction:

$$\Diamond\Box p \Longrightarrow \Box p \tag{2.4}$$

If the symbols $\Box$ and $\Diamond$ are assigned the meaning *henceforth* and *eventually*, respectively, then Theorem (2.4) takes the meaning "if p is eventually henceforth true then it is true now

and from now on." This theorem, while a logic consistency in S5, makes no sense in the real world. In fact, as will be seen in Section 5.4.3, specification of a latch is not possible in S5.

The problem with Theorem (2.4) perhaps lies not with the logic of S5 but with the assignment of meaning to operators for necessity ($\Box$) and possibility ($\Diamond$). An important semantic choice is whether the necessity operator will represent all time or all *future* time. When necessity is associated with "always," possibility is analogously associated with "sometimes." This system is referred to as non-ordered time and is best axiomatized by S5. Re-interpreting Theorem (2.4) under non-ordered time semantics, the meaning becomes "what is sometimes always true is always true," which does make sense. If making strong statements about ordering in time is a requirement of the temporal specification system, then choosing a non-ordered time logic system would be inappropriate. Ordered time requires interpretation of necessity, ($\Box$), and possibility, ($\Diamond$), as *henceforth* and *eventually*, respectively.

The modal system applicable for order-time semantics is S4. The necessity operator is assigned the meaning *henceforth* and the possibility operator becomes *eventually*. The S4 system does not include Theorem (2.4) and thus the interpretation "eventually, henceforth p implies henceforth p" is not a problem. The frequently referenced temporal systems of Manna and Pnueli are generally based on S4 with the passage of time indicated by state transitions.[1]

Specification systems often include predicates and quantification to increase expressive capability. There are several ways to add predicates and quantification to a logic system [23, 24, 25]. In combination with the 15 or so possible modal systems, various ways of quantification can produce 50 or 60 different quantified modal systems. As with the fundamental modal systems, the choice of axioms has a major bearing on expressive power and meaning, and consequent theorems of the logic. Great care must be taken to prevent theorems whose semantics are inappropriate, as in the example of Theorem (2.4).

[1] Ostroff [13] concisely consolidates much of the work of Manna and Pnueli.

Consider a temporal system based on S4 to which we wish to add quantification. Quantification is often added to temporal systems to enable statements about alternate futures. This problem is discussed later in detail in Section 5.3. As a preview, Ina Jo, also discussed later in Section 3.6, contains S4 with standard quantification as found in normal predicate logic. In the expressions that follow, we denote individual, single value variables by simple letters such as p, q, x, y. We denote complex expressions involving operators and predicates by Greek letters such as α, β.

First, we add to S4 the definition of a stronger implication operator[2] $\overset{\Box}{\Longrightarrow}$:

SIMP $\Box(p \Longrightarrow q) \equiv p \overset{\Box}{\Longrightarrow} q$

In addition to standard inference rules such as Modus Ponens, add the rule:

N1 $\vdash (\alpha \Longrightarrow \beta) \longrightarrow \vdash (\Box\alpha \Longrightarrow \Box\beta)$

Rule **N1** permits us to infer that whenever theorem α implies theorem β, *henceforth* α implies *henceforth* β.

To the axioms for S4, we add the following axioms for quantification:

Q1 $\forall x(\alpha \Longrightarrow \beta) \Longrightarrow (\forall x(\alpha) \Longrightarrow \forall x(\beta))$

Q2 $\forall x(\alpha \equiv \beta) \Longrightarrow (\forall x(\alpha) \equiv \forall x(\beta))$

Q3 $\forall x(\alpha) \equiv \neg(\exists x \neg(\alpha))$

The scope of the quantifiers is indicated by the parenthesized expression immediately following the quantifier. As expected, $\forall x(\alpha)$ means x takes any value wherever it occurs within the expression α. Variable x is normally called a *bound* variable in such an expression.

S4 without quantification is sufficient for simple interpretations of time, but when several alternate future state sequences are allowed, quantification is required to distinguish between characteristics of sequences.

The additional problems quantification introduce can be seen from the following example. Axiom **Q1** means that, if for all x, expression α implies expression β is true, then for

[2] Many modal logic texts such as [24] include $\overset{\Box}{\Longrightarrow}$ as part of the definition of system T. The intent of strong implication is to describe causality between two statements.

all x over α implies for all x over β is also true. The importance of **Q1** lies in the ability of universal quantification to distribute over implication. Replacing "implies" ($\Longrightarrow$) with *causally implies* ($\stackrel{\Box}{\Longrightarrow}$) makes the statement "if for all x, α causes β, then for all x over α causes β to also be true for all x." Since **Q1** is valid, it seems intuitive that its analog, expressed symbolically,

$$\forall x(\alpha \stackrel{\Box}{\Longrightarrow} \beta) \Longrightarrow (\forall x(\alpha) \stackrel{\Box}{\Longrightarrow} \forall x(\beta)) \tag{2.5}$$

would also be true. Yet, Formula (2.5) is not a theorem of our new system[3] Expanding Formula (2.5) by **SIMP** produces:

$$\forall x(\Box(\alpha \Longrightarrow \beta)) \Longrightarrow \Box(\forall x(\alpha) \Longrightarrow \forall x(\beta)) \tag{2.6}$$

Expression (2.6) appears correct, but notice that operators $\Box$ and $\forall$ switch scopes from the left to the right side of the central implication. The axiom set currently contains no tools permitting Formula (2.5) to be proven true. A similar expression that can be proved is:

$$\Box(\forall x(\alpha \Longrightarrow \beta)) \Longrightarrow \Box(\forall x(\alpha) \Longrightarrow \forall x(\beta)) \tag{2.7}$$

Formula (2.7) follows directly from **Q1** and **N1**. The difference between Formulas (2.6) and (2.7) lies in the order of operators $\Box$ and $\forall$ on the left side of the central implication. If we had the following axiom in our system:

$$\forall x(\Box(\alpha \Longrightarrow \beta)) \Longrightarrow \Box(\forall x(\alpha \Longrightarrow \beta)) \tag{2.8}$$

then we could prove Expression (2.6) from Expression (2.7) and (2.8). Expression (2.8) is a special form of :

$$(\forall x(\Box\alpha)) \Longrightarrow (\Box(\forall x(\alpha)) \tag{2.9}$$

which can be rewritten using negation and the definitions of $\forall$ and $\Box$ to:

BF $\Diamond(\exists x(\alpha)) \Longrightarrow \exists x(\Diamond\alpha))$

[3]Formula (2.5) *can* be shown to be a theorem of S5, but as we have already seen, S5 based systems are not appropriate for the kind of specifications we wish to write.

Formula **BF** is commonly known as the *Barcan formula* [24], and its converse is the *Inverse Barcan formula.*

The Barcan formula has important semantic implications if included in S4 based temporal systems. Permitting temporal and quantification operators to commute can facilitate proving theorems, but the semantics associated with Barcan can constrain the behavior of the system with respect to its objects.

Consider the meaning of Barcan:

$$\Diamond \exists x(P(x)) \Longrightarrow \exists x(\Diamond P(x)) \tag{2.10}$$

Expression (2.10) means if eventually there will exist an object x with property $P(x)$ then there (already) exists an object x for which $P(x)$ will eventually become true. Since the right side of Formula (2.10) implies that x already exists, the formula constrains the universe of objects to not grow. This follows from the ability Barcan gives us to replace "eventually there exists x" with "there exists x." Likewise, inverse Barcan implies the eventual existence of an object x satisfying $P(x)$, based on the existence of x now. Therefore, inverse Barcan implies the universe of objects cannot shrink. Barcan written as:

$$\Diamond \exists x(\alpha) \iff \exists x(\Diamond \alpha) \tag{2.11}$$

implies a statically sized universe of objects.

What is the practical implication of Barcan? Suppose we wish to specify behaviors of a system allowing users to log on and off. A timesharing system or a distributed database are examples. Inclusion of Barcan requires all users to always be a part of the specified system and to exist in a "logged-on" state or a "logged-off" state instead of randomly appearing and disappearing from beyond the boundaries of the system. The flexibility of permitting commutation of quantification and temporal operators must be weighed against potential constraints imposed by a statically sized universe. Statically sizing the universe may require extra predicates to express the membership or non-membership of an object within a system. Even with membership predicates, all *potential* objects require identification and specification.

The example of Barcan in the previous paragraphs illustrates the sensitivity of semantics to the choices for axioms in the temporal system. Unlike standard predicate logic, quantification cannot be added in an ad hoc fashion to modal systems without potentially serious side effects. As seen by the example above, adding axioms to a system to facilitate proof of a theorem that seems intuitive can result in unintended meanings, such as the statically sized universe.

CHAPTER 3

Survey of Temporal Specification Systems

This section reviews several well-known specification systems for specifying program properties that require special attention to temporal details. The systems surveyed are chosen to represent a broad range of specification techniques. Not all of the surveyed systems use modal-based temporal logic, but all systems attempt to handle relationships in time generated by concurrent and distributed systems problems.

Approaches to the temporal specification and semantic representation of programs can be grouped into three categories:

Operational: In this approach, programs are considered to be generators of program state sequences. Temporal reasoning about a program proceeds by examining properties of possible sequences of states.

Denotational In this approach a program is regarded as a function from some set of inputs to some set of outputs. Since it is difficult to introduce time into the concept of function mapping, this approach, in its pure form, is not often applied to temporal specifications.

Deductive: This approach emphasizes the behavior of the system as derived from sets of axioms. Deductive systems are necessarily based in logic and usually take an algebraic

form.

The operational approach to temporal specification and verification seems to be the most prevalent today. Operational specifications can often be executed or simulated directly, deductive approaches are used less but have greater expressive power. The deductive system analog to execution and simulation in an operational system is theorem proving. To be truly useful for significant applications, a deductive system requires automated theorem proving tools. The denotational approach is used rarely for temporal specification, mostly because function application tends to be atemporal.

In this paper we survey a variety of techniques for specifying temporal properties of systems. Each of the approaches discussed above is covered by one of more of the techniques.

The remainder of this chapter is organized as follows: In Section 3.1 we survey process algebras and discuss their origins relative to CSP. In Section 3.2 an example of an extended process algebra called LOTOS, is presented. Section 3.3 examines temporal characteristics of process algebras in more detail. Section 3.4 surveys two specification systems that focus on the flow of data through the system, known as *Streams*. In Section 3.5 the temporal characteristics of stream networks are examined in more detail. An algebraic deductive system called Ina Jo, extended with temporal operators, is presented in Section 3.6. Section 3.7 presents an alternative to temporal logic through a system that contains no temporal operators. Common features of the systems are drawn together and contrasted in Section 3.8.

3.1 Process Algebras — CSP and CCS

Communicating Sequential Processes [26] is an early attempt to specify the properties of concurrent and distributed programs. Instead of a functional theory based on the semantic content of the program, CSP, and its derivatives are based on the notion of primitive interaction and communication. The following example illustrates the motivation for focusing on interaction and communication:

Suppose we have two fragments of a program:

$X := 1$

and

$X := 0$

$X := X + 1$

In the absence of any interfering agent, the post-condition of both fragments is $X = 1$. On the other hand, if some devious agent is executing in parallel with the above fragments and interferes with the second fragment by setting $X := 1$ between the statements $X := 0$ and $X := X + 1$, the two fragments produce different results. The difference in behavior between the two sections of code is due to communication through variable X and interaction at an inopportune time. The problem illustrated in this example has led to specification systems that concentrate on interaction and communication.

CSP is a procedural approach to specifying concurrent system properties that relies heavily on the notion of agents, interaction and communication. The major features of CSP are Dijkstra's guarded commands for sequential process control, both sequential and parallel composition constructs, the latter to permit multiple processes to begin and execute together, and a simple notation for communication and synchronization between processes. Since CSP has been widely covered in the literature, only the basic constructs and ideas required for background will be discussed here.

The central idea of CSP is a set of concurrent cooperating communicating processes. Each process can manipulate local process variables with a variety of assignment and arithmetic operators. Communication between processes is accomplished by two basic operators. A statement of the form $C!V$ generally denotes the value V output on *channel* C. A statement of the form $C?V$ generally denotes waiting for and receiving a value on channel C which is assigned to V. Channels between processes are assumed to exist by virtue of definition and are associated with an identifier, such as C in the example. An interaction between processes through a channel is assumed to be instantaneous and to require the complement operation in some other process for completion. For example, each $C?V$ form must have a matching $C!V$ form ready to execute somewhere in the model for the former process to continue. Matching transactions explicitly synchronize processes and form the

basis of interactions between processes.

A CSP construct of the form $A||B$ indicates parallel execution of A and B. Combined with the channel operators, a construct of the form

$$(n = n + 1; C!n)||(C?m) \tag{3.1}$$

describes a classic producer-consumer relationship. In Statement (3.1) everything between parentheses denotes a process. Two processes in parallel composition are described in Statement (3.1). The first process generates numbers that are presented to the second process on channel C. The second process consumes the numbers offered by the first process. The production and consumption are perfectly synchronized by the requirement that each send operation must find a matching receive operation in order for each to continue.

Communication and synchronization are permitted between any number of matching processes, thus one producer can satisfy many consumers simultaneously. Guarded command construction and semantics follow Dijkstra's definition, for example:

$$G_1 \Rightarrow Cl_1[]G_2 \Rightarrow Cl_2[]\ldots[]G_n \Rightarrow Cl_n, \tag{3.2}$$

where G_i is the guard statement predicating the execution of statement Cl_i. CSP also contains an iterative command, variable typing, and many more constructs, matching the facilities of many programming languages. Unlike languages introduced later, CSP variable typing is fairly simple and limited to simple non-aggregate types such as integer and character.

The primary contribution CSP has made to other specification systems is a syntax and semantics for specifying, and even executing, parallel processes in a synchronized fashion. CSP contains no explicit temporal operators since temporal relationships are implied through the construction of the CSP program.

About the same time CSP was being developed, R. Milner proposed *A Calculus of Communicating Processes* (CCS) [27]. CCS has been further refined [28] and serves as the basis of the LOTOS specification system discussed later.

Like CSP, CCS is based on the notion of interaction and communication but adds the dimension of encapsulation and *observational equivalence*. The processes of CSP are known

as *agents* in CCS. CCS is a mix of procedural and algebraic systems that contains rewrite rules for transforming statements. CCS rewrite rules are used for simplification and to denote the meaning of the specification. CCS is not based on logic and is not directly deductive. Instead, CCS relies heavily on its syntax and context for meaning. Both CCS and CSP fall into a class of specification systems called *process algebras.*

The following example, taken from [28], illustrates the use of CCS. Suppose we have two people, *jobbers*, sharing a hammer and a mallet at a workbench on a production line. Unassembled parts flow in from one side of the bench and finished assemblies are required to flow from the other side of the workbench. Component assembly consists of inserting a peg into a base, which can sometimes be done by hand, sometimes with either a mallet or a hammer, and occasionally for very difficult jobs, only with a hammer. This example involves scheduling of limited resources (a hammer and a mallet) and coordination between agents (the jobbers). The description of a jobber is shown in Figure 3.1. Line 1 of Figure 3.1 contains an axiom that specifies when *Jobber* receives input, denoted **"in"**, for some job, *job*, the rule whose left side is labeled *Start* is followed next. Although this appears deductive, the semantics more closely follow an algebraically specified transition rule, hence our justification for calling CCS "procedural".

Several other characteristics of CCS are illustrated in this example. The identifier to the left of the dot in each rule denotes *Action* while the subject to the right of the dot denotes the *Agent* receiving the action. A complete rule, or axiom, consists of a new "state" (the agent operating next) and the actions involved in the transition. Parameters such as *job* can be passed between rules. Formal descriptions of the semantics for value passing is very complex and omitted here.

Line 5 of Figure 3.1 illustrates the use of alternation, denoted by +. The *Usetool* predicate can be satisfied either with *Usehammer* or *Usemallet. Usetool* appears as a next state from Line 4 in the *Start(job)* rules. Line 6 of Figure 3.1 specifies how a hammer can be used.

CCS uses the idea of a *labeled transition* to indicate action. Figure 3.2 shows a diagram for the interaction between the jobber and the hammer. In CCS notation, overlines are

$$
\begin{aligned}
\textit{Jobber} &\stackrel{def}{=} \mathbf{in}(\textit{job}).\textit{Start}(\textit{job}) \\
\textit{Start}(\textit{job}) &\stackrel{def}{=} \textbf{if } \textit{easy}(\textit{job}) \textbf{ then } \textit{Finish}(\textit{job}) \\
&\quad\ \textbf{else if } \textit{hard}(\textit{job}) \textbf{ then } \textit{Usehammer}(\textit{job}) \\
&\quad\ \textbf{else } \textit{Usetool}(\textit{job}) \\
\textit{Usetool}(\textit{job}) &\stackrel{def}{=} \textit{Usehammer}(\textit{job}) + \textit{Usemallet}(\textit{job}) \\
\textit{Usehammer}(\textit{job}) &\stackrel{def}{=} \overline{\mathbf{geth.puth}}.\textit{Finish}(\textit{job}) \\
\textit{Usemallet}(\textit{job}) &\stackrel{def}{=} \overline{\mathbf{geth.puth}}.\textit{Finish}(\textit{job}) \\
\textit{Finish}(\textit{job}) &\stackrel{def}{=} \overline{\mathbf{out}}(\textit{done}(\textit{job})).\textit{Jobber}
\end{aligned}
$$

Figure 3.1. CCS Description of Jobber's Behavior

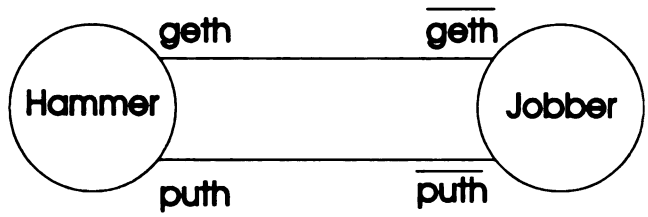


Figure 3.2. Transisition Diagram for Jobber and Hammer

used to signify complementary sides of a communication. Neither side is designated as the input or output. CCS rules require that only complements can interact.

When Line 6 of Figure 3.1 is combined with the diagram in Figure 3.2, we see that the axiom specifies that the completed transaction of getting and putting a hammer is equivalent to the transitions that put the jobber into a *Finish* state.

Parallel composition of several agents is denoted by a vertical bar. Using this notation, the two jobbers and the hammer would be written as

$$(\textit{Jobber}|\textit{Jobber}|\textit{Hammer}). \tag{3.3}$$

If we add *Mallet* to Expression (3.3) we get Expression (3.4) that gives a complete specification for the jobshop:

$$(Jobber|Jobber|Hammer|Mallet). \tag{3.4}$$

As a specification grows in complexity and requires higher levels of abstraction, complex agents can be constructed from a series of simpler agent expressions. In CCS definitional meta-notation, but not in the CCS language itself, action and transition are denoted by:

$$P \xrightarrow{l} Q \tag{3.5}$$

where P and Q are agents associated with states and l is the action that caused the transition, or agent Q to act. Note that this notation is not part of CCS, itself, but rather a notation used to describe actions in CCS. Referring again to the jobshop example, the rule:

$$Hammer \stackrel{def}{=} \mathbf{geth}.Busyhammer \tag{3.6}$$

signifying that a hammer becomes busy through the **geth** action (see Figure 3.2) is written in labeled transition notation as:

$$Hammer \xrightarrow{geth} Busyhammer \tag{3.7}$$

If we intend to use the entire jobshop as an agent in a larger model, we may wish to hide the interactions between jobbers, the hammer and the mallet. Hiding is accomplished through a *Restriction* operator, "\." An expression of the form $(A|B)\backslash c$ means the c action is private to the composition of A and B.

If an action between members of a composite is not hidden, it is possible for the action to be supplied by a third agent. Suppose we define agents A and B as follows:

$$A \stackrel{def}{=} a.A' \tag{3.8}$$

$$A' \stackrel{def}{=} \overline{c}.A \tag{3.9}$$

$$B \stackrel{def}{=} c.B' \tag{3.10}$$

$$B' \stackrel{def}{=} \overline{b}.B \tag{3.11}$$

Since:

$$A \xrightarrow{a} A' \tag{3.12}$$

we can infer that

$$A|B \xrightarrow{a} A'|B \tag{3.13}$$

because B does not take part in A's transition. a can be supplied to A by an outside agent and leaves B unaffected.

When placed in composition, we can immediately see that A and B can interact, or handshake, through complementary[1] actions c and $\overline{c}$ without the outside world. Statements (3.12) and (3.13) imply a rule for writing transitions for compositions, but the handshake produces

$$A' \xrightarrow{\overline{c}} A \text{ and } B \xrightarrow{c} B' \tag{3.14}$$

This indicates that A and B change state simultaneously and presents the question of how to write a handshake transition.

CCS solves this with a τ transition, also called a *perfect transition*, which is written as follows:

$$A'|B \xrightarrow{\tau} A|B'. \tag{3.15}$$

The meaning of τ is that both agents change state simultaneously through a handshake interaction. τ transitions require pairs of actions of the form $(a, \overline{a})$ within the composition. A statement of the form of (3.15) denotes that although both c and $\overline{c}$ are available outside of the composition, there is an internal, hidden handshake possible between them. The restriction operator can explicitly hide the handshake:

$$C = (A|B)\backslash c \tag{3.16}$$

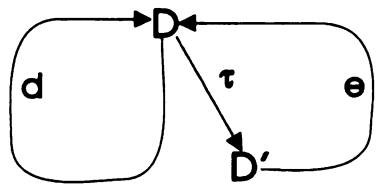
so only transitions a and $\overline{b}$ are observable. Agent C is now defined by the equations:

$$C_0 \stackrel{def}{=} \overline{b}.C_1 + a.C_2 \tag{3.17}$$

$$C_1 \stackrel{def}{=} a.C_3 \tag{3.18}$$

$$C_2 \stackrel{def}{=} \overline{b}.C_3 \tag{3.19}$$

[1] The actions are complementary in the sense that they are opposite sides of a handshake operation. The actual contents of the actions do not have to form a "complement" in the logical sense



$$C_3 \stackrel{def}{=} \tau.C_0 \tag{3.20}$$

Multiple compound internal handshake operations can produce expressions such as

$$P_1 \xrightarrow{\tau} P_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} P_n \tag{3.21}$$

that CCS rules allow reduction to

$$P_1 \xrightarrow{\tau} P_n \tag{3.22}$$

Although it may seem that τ actions can always be eliminated and ignored, the following example shows this case is not true. One has to be very careful how an agent with internal structure is reduced[2] Consider a recursively defined agent:

$$D = d.D + \tau.e.D \tag{3.23}$$

with a transition graph represented as follows:

The meaning of Equation (3.23) and Figure 3.1 is that state D can result from action d or an internal transition (τ) followed by action e.

If the transform

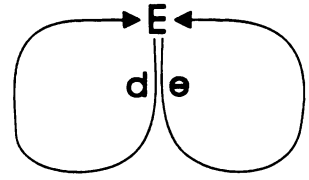
$$\tau.P \rightarrow P \tag{3.24}$$

were always valid then agent D described by Equation (3.23) would be equivalent to and agent E described by:

$$E = d.E + e.E \tag{3.25}$$

and E's transition graph would be as follows:

[2]Milner claims in [28] one has to be very careful how τ involved statements are reduced.



But are agent D and agent E really the same? Agent E always accepts either d or e and "resets" again ready to accept either action. The behavior of D is different. From the initial state of D, only d is valid. If the τ transition has occurred, only e is acceptable. The external, observable behavior of D and E differ in that D appears non-deterministic while E is deterministic.

We suggest, without proof, that the τ transition can always be ignored in composition except when it occurs first, that is, syntactically on the left side of a construct. At all other times, τ is truly internal and unobservable. When on the left, it is observable in the sense that the handshake(s) must occur before the composed agent is ready for other external actions.

The justification for τ arises from the premise that only observable interactions are important. In a composition, an internal handshake is not observable directly, although its effects are. The handshake therefore needs a representation (for the reasons just seen), but not a detailed description.

As mentioned earlier, CCS includes a variety of complex expressions and inference rules for reducing equations to equivalent forms. The major agent expressions of CCS, some of which have already been introduced are:

1. $\alpha.E$, denotes α is a prefix action to agent E
2. $\Sigma_{i \in I} E_i$ is shorthand for $E_1 + E_2 + E_3 + \ldots$, denoting alternation.
3. $E_1 | E_2$ as introduced, denotes the parallel composition operation.
4. $E \backslash L$ is written for restriction that hides all actions in L (L may be a set of actions)
5. $E[f]$ denotes relabeling where f is a relabeling function that re-writes E according to the rules of f.

For each of the constructs above, there is at least one inference rule to manipulate the algebra. For example, the composition rules already introduced have the following rules:

$$\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \tag{3.26}$$

$$\frac{E \xrightarrow{l} E',\ F \xrightarrow{\bar{l}} F'}{E|F \xrightarrow{\tau} E'|F'} \tag{3.27}$$

A simple transition is defined as :

$$\frac{}{\alpha.E \xrightarrow{\alpha} E} \tag{3.28}$$

Rule (3.26) and (3.27) have already been discussed. Rule (3.28) is a formalization of the intuitive notation that E is followed after the α action takes place. Although deductive in form, these rules are not part of a deductive system.

The τ transition appears in some sense to be at the core of CCS. Composition and observable action are foundational CCS ideas, and the τ transition is important to both. Observable action, as seen by the example of an equivalent C agent constructed for A and B above, forms a central theme appropriately called *observational equivalence.* In that example, Statements (3.8) through (3.11) were replaced by Statements (3.17) through (3.20) thus forming agent C. Agents that appear to behave equivalently can be inferred to be interchangeable. τ plays the role of abstracting away handshake details internal to A and B's interaction. Process algebras statements, and CCS statements in particular, are frequently manipulated for simplification and equivalent forms.

A frequent proof technique is to show that one form of an agent expression is equivalent in some way to another. This concept is known as *Bisimulation.* Bisimulation, and therefore observational equivalence, can take on many forms. The two primary forms are called *strong* and *weak*, each denoting equivalence between agents from a certain point of view. It appears that there may be many forms of observational equivalence, depending on what is observed and how *equivalence* is defined.

Traditional automata theory provides a definition of equivalence based on the languages a state machine accepts. Both strong and weak equivalence are based differently. As an

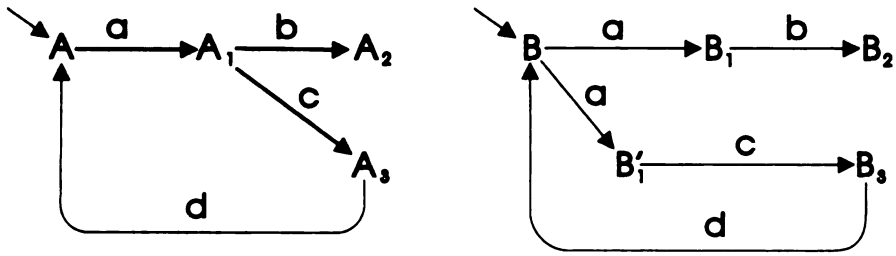


Figure 3.3. Language equivalent state machines with differing behavior

example, borrowed with modifications from [28], consider the two descriptions in Figure 3.3. If treated as finite state automata, we can easily see that both accept the language:

$$L = (a.c.d)^*.a.b.0 \tag{3.29}$$

where 0 denotes a state from which the agent cannot escape. Although L represents the language for both diagrams, B is non-deterministic and after accepting a can choose to either accept c or b but not both. Agent A, on the other hand, accepts either b or c. An experimenter trying multiple runs of sequences on both A and B would identify this behavioral difference, and find nothing she can do determines whether b or c is accepted next after a by A. Agents A and B are not strongly equivalent. To be strongly equivalent, two agents must have corresponding transitions for each action. This means they must always behave in an identical fashion for a sequence of operations.

The algebra used to completely describe and reason about concurrent systems in CCS is much more complex than described here and space limitations do not permit a thorough treatment of the subject. Milner has more completely described process algebras and the ideas on which CCS is based in [28].

3.2 LOTOS

LOTOS [29] (Language of Temporal Ordering Specifications) is a formal description language based on many of the formalisms and syntax from CCS and CSP. Although LOTOS is a process algebra, its syntax is more procedural than CCS or CSP and resembles imperative programming languages. In fact, systems for executing LOTOS specifications have been constructed [30]. LOTOS's CCS heritage is so strong that the LOTOS interpreter described in [30] transforms LOTOS specifications into CCS prior to execution.

LOTOS contains a rich set of operators and has a syntax at least as complex as CCS. For a description of the entire LOTOS language see [29]. The basic LOTOS constructs are presented in this paper as a framework for discussion of the temporal characteristics of LOTOS.

LOTOS's procedural constructs emphasize communication and interaction between processes using notions very similar to CSP and CCS. It's primary emphasis on procedural contents, using the interaction-communication paradigm, are the justification for classifying LOTOS as a process algebra.

Basic LOTOS units of action, equivalent to CCS agents, are called *processes.* Expressions involving process identifiers, communication and synchronization constructs, and process composition, are called *behavior expressions.* LOTOS specifications are a marriage of two types of specification methods: procedurally oriented state transition descriptions that are used for process descriptions, and an algebraic abstract data type (ADT) description language used for complex objects and special variable types manipulated by processes.

LOTOS's ADT language is a typed algebra employing axioms and is derived from ACT ONE [31]. The ADT portion of LOTOS is used to describe the behavior of the objects manipulated by processes in the procedural portion of a LOTOS specification. Objects described in the ADT language can be freely used in behavior expressions and behavior expressions can invoke ADT functions defined within the ADT specification.

An example of a commonly found use of LOTOS behavior expressions combined with ADT specifications is a system that contains communicating systems with multiple processes

requiring a queue for interaction. The behavior of the system processes is described with a process definition containing behavior expressions that resemble CSP and CCS constructs. The queue data structure is described through a series of function signatures and universally quantified axioms specifying how the queue and its functions behave. An example using an ADT description of a natural number type is given later.

LOTOS adopts the "!" and "?" operators from CSP to denote synchronization through output and input, respectively, on an associated channel. A sequential series of actions are composed with the ";" operator. For example, the expression

$$a?x; b?y; c!largest(x, y); \textbf{stop} \tag{3.30}$$

specifies reading variable x from port a followed by reading variable y from port b, then outputting the larger of the two values on port c. Two special actions, **stop** and **exit**, signify halting a process and terminating a process normally, respectively. The **stop** process is capable of no further action, including termination. It offers no actions that can be matched and it can match no actions. The special process **exit** signifies normal termination. The process expression containing an **exit** terminates normally, an event which may be observed and used by other processes.

Statement (3.30) is not quite explicit enough, in that we can infer from the context of specification that x and y are integers or natural numbers, but the type is not conveyed explicitly. Both x and y need typing. Typing is written in LOTOS by post-fixing variables with *:type*. A better form of Expression (3.30) with typing information added would then be:

$$a?x : nat; b?y : nat; c!largest(x, y); \textbf{stop}. \tag{3.31}$$

The description of function $largest(x, y)$ is found in conjunction with the definition of type *nat*. This is a typical example of an ADT exported function definition in LOTOS.

CCS and CSP style interactions are augmented in LOTOS by providing an optional predicate to be satisfied before the interaction can complete successfully. To illustrate, the following expression:

$$a?x : nat[x > 3]; b!x; \textbf{stop} \tag{3.32}$$

requires inputting x on port a to succeed only for offered values of 0,1 and 2. Until one of these values is offered by another process, this process waits.

Interactions within composed processes is captured by the concept of the **i** action that denotes "internal", or hidden action. The **i** action is equivalent to the CCS τ transition. The idea is that an action external to a macro-process may trigger a series of actions with the macro-process that are to be hidden from the rest of the system. *Hidden*, in the LOTOS, sense means actions on channels are not to be available for transition matches elsewhere in the system. *Hiding* appears to have its greatest use in composed processes that require internal synchronization that is not to be interfered with by any later additions or deletions to the specification.

LOTOS contains a complete set of constructs for composing processes sequentially, concurrently, with guards, and with alternation. Normal sequential composition has already been introduced in the form $A;B$. *Enabling composition* is denoted by $A \gg B$ and is interpreted to mean B is enabled upon, and only upon, the successful termination of action A. Note that if A contains a **stop**, B will never be enabled.

LOTOS has three major constructs to signify parallel operation and synchronization of processes. The formal description is somewhat complex, so the explanation here will be by example. We will begin by breaking down the following expression:

$$a; c; \mathbf{stop} | [c] | b; c; \mathbf{stop} \tag{3.33}$$

The three composed actions to the left of "|" form one composite process made from actions a, c, and **stop**. The three terms to the right of the second "|" form a second process. The first process is free to execute action a at any time and similarly, the second is free to execute action b at any time. Once both a and b have occurred, each macro-process is ready to execute action c, but only in synchronization. Vertical bars in LOTOS denote parallel composition of processes. Processes of the form of Expression (3.33) mean that the parallel composition can continue only when each process is ready to execute the shared actions between the bars. The shared actions are called *synchronization gates* in LOTOS. In Expression (3.33), the synchronization gate is c. This statement represents the temporal

notion "a and b before c"

When the set of synchronization gates is empty, the composed processes are free to execute in parallel in a pure interleaved mode. This is denoted by $A|||B$ in LOTOS notation. When all gates (actions) are in common between processes, the composed processes are in full synchronization and must proceed in lockstep as each action of each process occurs. This is denoted by $A||B$.

The alternation command takes the form $A[]B$. The informal meaning of alternation is either A or B may be executed depending on the interaction of A and B with their environments. The alternation expression

$$a; b; c; \mathbf{stop}[]b; a; c; \mathbf{stop} \tag{3.34}$$

offers either action a then b followed by c, or b then a, followed by c. Expression (3.34) is, in fact, exactly equivalent to Expression (3.33). Alternation also permits the specification of non-determinism as follows:

$$a; b; \mathbf{stop}[]a; c; \mathbf{stop} \tag{3.35}$$

Expression (3.35) permits either b or c after a as neither is specified as preferred and there is no environmental action possible that would choose one over the other. Unlike a non-deterministic automata, Expression (3.35) implies no look-ahead. Once an alternative has been (non-deterministically) chosen, execution will continue along the chosen path. If $a; c$ is the required behavior in Expression (3.35), then the expression permits $a; b$ to be nondeterministically chosen and a deadlock to potentially occur.

Alternation combined with the hidden **i** action is often used in protocol specifications to provide a kind of asymmetric nondeterminism. Statements of the form:

$$\textit{normal-course-of-action}[]\mathbf{i}; \textit{disconnect-indication} \tag{3.36}$$

appear frequently in connection with protocols.

Guarded commands take a form very similar to alternation. Given a guard predicate P, $[P] \rightarrow A$ has the expected interpretation of executing A when P is true. A series of guards combined with alternation forms a case statement.

```
type natural is
    sorts nat
    opns zero:→ nat
         succ: nat → nat
         largest: nat, nat → nat
    eqns ofsort nat
         forall x,y: nat
              largest(zero, x) = x;
              largest(x, y) = largest(y, x);
              largest(succ(x), succ(y)) = succ(largest(x, y));
endtype
```

Figure 3.4. Algebraic description of type *nat*

Returning to the example in Statement (3.31), above, type *nat* and function *largest*() require definition. This is accomplished in the LOTOS abstract data typing language (ADT) as shown in Figure 3.4. The ADT language is an algebraic specification language derived from ACT ONE [31] based on axioms and rewrite rules, but unlike many algebraic languages, LOTOS's language has provisions for parametric types. The ADT language plays no particular role in the temporal characteristics of LOTOS specifications.

3.3 Temporal Characteristics of Process Algebras

All process algebras concentrate on temporal properties such as synchronization, safety, and precedence from a procedural, process interaction point of view. For example, LOTOS Expressions (3.34) and (3.35), earlier, both make equivalent algorithmic statements about the temporal precedence of *a*, *b*, and *c*. Assuming the existence of a two place infix predicate "<" meaning one action is before another, the intention of both Expressions (3.34) and (3.35) could be concisely expressed in declarative form as:

$$(a<c) \wedge (b<c) \tag{3.37}$$

The attractiveness of process algebras is their close connection to what occurs in actual implementations and the *lack* of need for explicit temporal operators. On the other hand, we

feel that a specification presented in process algebra is more complex and difficult to grasp intuitively than an algebraic counterpart. We argue that this follows directly for the same reason behavior is harder to derive from program code than from a properly constructed declarative specification.

Another problem found in the three process algebras surveyed is their reliance on syntax and context for meaning without an underlying formal system. Earlier, the statements:

$$a; b; c; \mathbf{stop}[]b; a; c; \mathbf{stop} \tag{3.38}$$

$$a; b; \mathbf{stop}[]a; c; \mathbf{stop} \tag{3.39}$$

were presented as examples of alternation. A question that may arise is why Statement (3.38) represents a choice of actions while Statement (3.39) represents nondeterminism. Both statements appear very similar.

Verification techniques for process algebras rely heavily on bisimulation. Bisimulation itself is a complex topic to which many pages have been devoted. While deductive systems draw on inference rules for proofs, process algebras use various forms of comparisons of externally observable interactions, which is exactly what bisimulation is.

As we will see in subsequent sections, many non-process algebraic specification techniques mark time by state transitions. Such systems associate the semantics of temporal operators with sequences of states as a substitute for actual time passage. While all of the process algebra systems surveyed contain the notion of labeled state transitions, transitions are generally hidden except at an interaction point. Multiple state encapsulation reduces the number of observable states to those that are most important to the process being described. We may not need to know, or even be permitted to know, that a LOTOS process goes through 43 internal states before visible interaction I occurs. All we externally observe is what amounts to the process's 44th state where interaction I takes place. We therefore know nothing about the temporal properties of the first 43 states. We only know that I has occurred and how this occurrence relates to other observable actions in the system. This property of process algebras is a consequence of focusing on the interaction rather than state transitions, and yields the dividend of higher levels of abstraction.

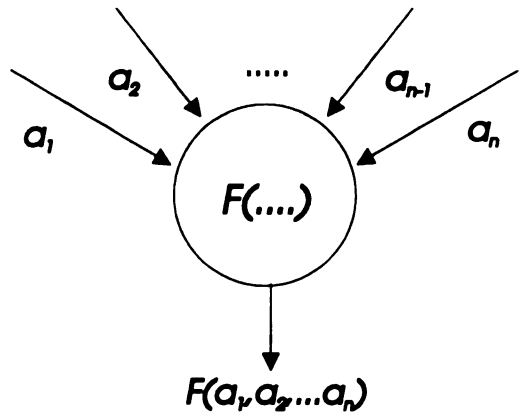


Figure 3.5. An operator agent F

3.4 Data Network Specifications

As an alternative to focusing on the interaction and communication between agents, another approach to specification is possible by concentrating on the flow of data through a network of processes. Both Lucid [11] and the *streams* concept of communicating agents [32] are examples of this approach. We adopt Broy's terminology of *stream* denoting a finite, or infinite, continual flow of data through a network of connected nodes. Processing occurs at the nodes, which are termed *agents.* Each agent represents a function from its set of inputs to an output. Figure 3.5 schematically depicts an agent continuously accepting inputs and transforming them according to some function F.

Figure 3.6 shows how a simple arithmetic mean operation would be depicted in a stream-agent system. In Figure 3.6, X and Y represent a continual series of discrete input values. Each agent waits until its inputs are ready, then performs the indicated function. The value "2" on the input to the "/" agent represents a continual stream of the constant 2, *e.g.*, $2, 2, 2, 2, 2, 2, 2 \ldots$.

Lucid [11] is one such stream-agent system. In addition to arithmetic functions, Lucid has several operators, called temporal operators in Lucid terminology, that operate on input

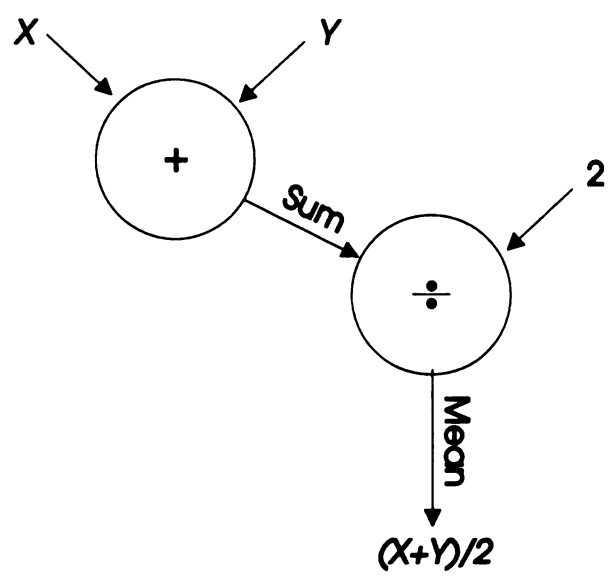


Figure 3.6. Lucid network for calculating arithmetic means

streams. The table in Figure 3.7 lists Lucid temporal operators and their function. Since streams in both Lucid and Broy's *stream* system are considered to be infinite sequences, a **last** operator would not make sense.

In addition to a graphical representation, Lucid has a procedural language that is simple and intuitive. The *Mean* program from Figure 3.6 is shown in language form in Figure 3.8.

The exact instant when an agent produces an output is clearly a function of the time the

Operator	Function description	Result on $A = 1,2,3,4,\ldots$ and $B = 10,20,30,40,\ldots$
next A	Next operator	$2,3,4,\ldots$
first A	First of sequence	$1,1,1,1,1,\ldots$
A fby B	Followed by	$1,10,20,30,\ldots$

Figure 3.7. Table of temporal Lucid operators

```
mean
    where
        mean = sum/2
        sum = x + y
    end;
```

Figure 3.8. Language form of the *mean* program

inputs are ready and how long it takes to perform the agent function. Given a network of agents and the time required for each agent's execution, the earliest and latest times output can appear at any agent in the network can easily be calculated. These calculations can be captured by a series of agents connected together in a network. This means that performance of a Lucid data network can be calculated by a similar Lucid data network. Skillicorn and Glasgow [11] present a technique that transforms any Lucid network into two networks that calculate the earliest and the latest times output from the original network is produced. Although their technique is specific to Lucid, it should work with slight modifications for any stream-agent network.

Temporal specifications in Lucid are limited to calculating performance values of a system that can be represented in a stream-agent paradigm.

Broy's [32] stream abstraction enriches the expressive power of stream-agent networks through the addition of recursion and explicit temporal operators. Figure 3.9 shows a specification for an agent that picks the next value from stream a for values from stream s that are true, or the next value from stream b otherwise. The "&" operator represents concatenation of streams in this notation. Guards are denoted as $P \Rightarrow action$, where P is any arbitrary predicate. The **first** operator produces the first element of the stream and the **rest** operator produces the entire stream minus its first element. The result of input streams:

$a = 0\ 2\ 4\ldots$

agent schedule
input stream data a, **stream data** b, **output stream data** r,
first s = true $\Rightarrow$ r = **first**a&schedule(**rest** $a, b,$ **rest** s),
first s = false $\Rightarrow$ r = **first**b&schedule($a,$ **rest** $b,$ **rest** s),
end

Figure 3.9. Stream schedule agent

$b = 1\ 3\ 5\ldots$

$s = true\ true\ false\ false\ true\ false\ldots$

supplied to agent schedule would be:

$r = 0\ 2\ 1\ 3\ 4\ 5\ldots$

per the specification.

Notice that infinite recursion is used in agent schedule in Figure 3.9. At each invocation, **first** operates on the stream that remains. Since the streams are potentially infinite, no terminal condition for the recursion is provided.

The example in Figure 3.9 illustrates several characteristics of this specification method. First, the specification of agent behavior is in an algebraic style. Second, interactions are explicit functional style constructions. Contrast this style to process algebras where the communication is through channels. Finally, recursion is required in order to operate on the relevant part of the stream. Typically, the remainder of the stream is passed on to subsequent invocations of a function for further recursive processing.

To provide an alternative to recursion, Broy has defined three temporal operators. Use of the temporal operators produce a more compact notation than recursion, for equivalent meaning. The general form of a temporal operation in Broy's notation is $Ts : P$ where T is an operator, s is a stream variable and P is a predicate over the stream. Intuitively

$$\Box r : P \tag{3.40}$$

means P must be satisfied for every element of the stream r. Similarly,

$$\Diamond r : P \tag{3.41}$$

means P must be satisfied for some elements of the stream r, although it is hard to see how this operator would be used. The third temporal operator:

$$\bigcirc r : P \tag{3.42}$$

denotes the **rest** of r after the current element. The temporal operator $\bigcirc$ is frequently used to mean "in the next state" [13] in temporal logic specifications.

The agent description

$$\Box a, b, r : \textbf{first } r = \textbf{first } a + \textbf{first } b \textbf{ end} \tag{3.43}$$

signifies adding all corresponding numbers from stream a to numbers from stream b to produce a stream of sums called r. Example (3.43) is obviously simpler and more compact than an equivalent description using recursion. Broy notes that a description for agent *schedule* using temporal operators is "less obvious" than the example in Figure 3.9.

3.5 Temporal Characteristics of Stream Networks

As we saw was the case for process algebras in Section 3.3, none of the stream network specification methods directly reference time. All of the stream temporal operators are substitutes for quantification over some property of a stream. For example, the $\Box$ operator, commonly interpreted as "henceforth" in temporal logic, means "for the rest of the stream". This expresses the concept of universal quantification *i.e.*,"for all (remaining) members of the stream." The $\Diamond$ operator, interpreted in temporal logic as "eventually", means "for some members of the stream," thus expressing existential quantification over the stream. Both of these operators only refer to "real" time, to the extent the production of the stream relates to time.

While temporal operators do shorten the notation and reduce the need for recursive specification, the quantifiers **for all**, and **there exists**, as applied to a stream, would serve an equivalent function. We must conclude then, that the temporal operators in stream

network systems are only useful for characterizing subsequent members of streams and cannot be used for general temporal specifications.

3.6 Temporal Extensions to a Non-Procedural Language

Ina Jo is a non-procedural specification language that is based on first order predicate calculus [33]. Ina Jo's primary use is to specify properties of secure operating systems. Although Ina Jo is algebraic, the underlying model is a nondeterministic state machine where each state is a mapping from a set of typed variables to values. State transitions are defined to consist of changes to one or more of the values of state variables.

Wing and Nixon have extended Ina Jo [20] with a set of temporal operators based generally on the S4 modal system with quantification. Their motivation was based on requirements for specifying concurrency behaviors of programs in addition to functional behaviors. Before starting a description of the Ina Jo temporal extensions, a basic description of non-extended Ina Jo is given.

Figure 3.10 shows the basic syntactic outline of an Ina Jo specification. The overall specification can be broken into roughly three parts: a description of the global state as defined by typed variables and constants, a set of assertions, and a set of transforms that describe legal state transitions. State variables introduced in the **type**, **constant**, and **variable** declarations can be either primitive built-in types with simple identifiers or complex constructed types using previously defined types. Variables may take parameters to represent one value of a set of values, similar to the concept of arraying referencing.

Assertions can be expressed in a variety of forms in an Ina Jo specification. Assertions in the **axiom** section state what is true in all models independent of any state machine. Assertions in the **define** section are named and can be used in subsequent assertions. Assertions in **define** sections operate like syntax macros within the body of the specification. Assertions in the **criterion** section state what must hold in all states of the state machine. Assertions in the **initial** section specify initial conditions that hold in all starting states of the state machine.

```
specification <system name>
    type < ... >
    constant < ... >
    variable < ... >
    axiom < ... >
    define < ... >
    criterion < ... >
    initial < ... >
    transform <transform name>
        refcond < ... >
        effect < ... >
        :
        :
    transform <transforms name>
        refcond < ... >
        effect < ... >
        :
        :
    end <system name>
```

Figure 3.10. Ina Jo Specification Template

The **transform** section describes legal state transitions for the underlying state machine. This section may take input parameters but may not explicitly return a result. Return results are specified implicitly by specifying new values of state variables in terms of present values of state variables.

Each **transform** consists of a pre-condition specified in a **refcond** clause and a post-condition specified in an **effect** clause. Transforms are enabled when the predicate in a **refcond** is satisfied, thus "firing" the set of transform rules in the body of the **effect** clause. The **effect** clause contains rules for determining the values of variables in the next state from values in the present state. The state machine is nondeterministic because multiple **refcond** rules may be enabled at any one time. When this occurs, the state variable values in the next state can be determined by any of the matching **effect** clauses. Non-determinism may also be introduced in the **effect** clause by specifying disjuncts, any of which may be satisfied.

```
specification LIVE
    variable
        x : integer;
        done : bool;
    initial
        x>0 & ~done
    criterion
        x>0 & ~done — x=0 & done
    transform decrement
        effect
            (x > 1) => N''x = x - 1 <> x = 0 & done = true
end LIVE
```

Figure 3.11. Ina Jo non-temporal example

Figure 3.11 shows a small example of an Ina Jo specification that will be cast later in temporal terms. The specification **LIVE** simply stipulates that the variable x starts out positive and is decremented to zero.

The construction N''x, signifies the next value of x. Similarly, Ina Jo has quantification operators A'' to denote *for all*, and E'' to denote *there exists*. The use of double prime after an operator is a special Ina Jo construct to permit combinations of one or more letters to represent operators.

The construction p => q <> r denotes "if p is true then q, else r." For example, (x > 1) => N''x = x − 1 <> x = 0&*done* = *true*. If the guard is true, then the first condition must be satisfied, otherwise the condition after "<>" must be satisfied. Since Ina Jo is non-procedural, all statements are predicates to be satisfied.

The temporal model for extended Ina Jo is based on five temporal operators, each of which is prefixed by either universal or existential quantification, for a total of ten operators. Temporal operators range over state sequences of the Ina Jo state machine in a branching time model.

Branching time models assume that there are a finite number of possible alternate state sequences after each state that, all together, can be viewed as a tree. The root of the tree corresponds to the initial state and the leaves correspond to final states. A computation

consists of one entire path from the root of the tree to a leaf. Quantification operators specify over what the temporal operator will range: either all branches from the current state or just some branch from the current state. Given formulas p and q without temporal operators, formulas involving p and q with temporal operators make statements about the values of p and q in subsequent states. The ten Ina Jo temporal operators and their informal definitions in terms of a branching time tree, a current state s, and formulas p and q are as follows:

ah$''p$ holds in s iff p is true at all states of all branches rooted at s including s

eh$''p$ holds in s iff there exists a path departing from s such that p is true at all states on this path

av$''p$ holds in s iff every path departing from s has on it some state at which s is true.

ev$''p$ holds in s iff there exists a path departing from s such that p is true at some state on this path.

an$''p$ holds in s iff p is true at every immediate successor of s

en$''p$ holds in s iff p is true at some successor of s

p **au**$''q$ holds in s iff every path departing from s has on it some state s' satisfying q such that p is true at every predecessor state of s'.

p **eu**$''q$ holds in s iff some path departing from s has on it some state s' satisfying q such that p is true at every predecessor state of s'.

p **ab**$''q$ holds in s iff every path departing from s eventually q is true, then for every path, q is false at every predecessor to the first state at which p is true.

p **eb**$''q$ holds in s iff some[3]path departing from s eventually q is true, then for every path, q is false at every predecessor to the first state at which p is true.

The h$''p$ and e$''p$ operators correspond to $\Box p$ and $\Diamond p$, respectively, discussed in Section 2.1. The operators N$''p$, p u$''q$, and p b$''q$ are defined as "in the next state after p", "p until q", and "p before q" respectively, and can all be defined from "henceforth" and "eventually", except for "next". "Next" has a non-time related meaning that pertains directly to state sequencing. In modal logic, it is very difficult to define the concept of "next" without reference to some sort of transition or other marker event.

[3]The definition in [20] reads "every" and is incorrect.

```
Assn ::= '~'Assn | Assn BinOp Assn | '(' Assn ')'
        | Assn '=>' Assn '<>' Assn
        | Quant Binding {, Binding} '('Assn'('
        | Term '=' Term | Term | Term
        | UnOp Assn | Assn TBinOp Assn
Term ::= Var | 'N"' Var | Func_Name
        '('Term {, Term},')' | 'N"'Func_Name
        '('Term {, Term ')'
BinOp ::= '&' | '|' | '- >' | '< - >'
UnOp ::= 'ah"'|'eh"'|'av"'|'ev"'|'an"'|'en"'|' ah"'|'N"'
TBinOp ::= 'au"'|'eu"'|'ab"'|'eb"'
Quant ::= 'A"' | 'E"'
Binding ::= Id {, Id}:Type_Name
```

Figure 3.12. Ina Jo Syntax for Assertions

A major distinction between Ina Jo and the specification systems presented so far is that Ina Jo has a formal foundation (temporal logic) and a well-defined underlying reference model (*i.e.*, the state machine). Process algebras, especially CCS, tend to be more syntactic in nature with semantics based on imperative actions implied by statements in the specifications. As a consequence, inferencing rules in a formally based system that are used to verify properties of a system from its specifications are easier to construct. As mentioned in Section 3.1, verification in CCS is not logic based, but rather relies on a concept of observational equivalence, called bisimulation. Ina Jo, on the other hand, is an example of a system from which properties of a system can be deduced through application of inference rules and axioms of the system.

Since we suggest that formal foundations are important, we will briefly present Ina Jo's formal foundation and present formal definitions of the informally introduced temporal operators.

The relevant syntax of Ina Jo's assertion language is presented in BNF in Figure 3.12.

Each Ina Jo model is defined as an ordered quintuple $\langle Init, State, Dom, Trans, Eval \rangle$ of:

Init: a set of alternative initial states

State: a set of states, $State, Init \subseteq State$

Dom: a primitive domain of typed values

Trans: an finite set of binary state transition relations on *State*

Eval: a semantic evaluation function for the class of Ina Jo assertions in Assn of Figure 3.12

First, a description of the state machine will be presented, finally building up to the definition of the *Eval* function. The temporal definitions we are seeking are all in terms of the *Eval* function.

Let Id be a set of identifiers. A machine, $M \subseteq Id$, is a set of identifiers representing Ina Jo state variables as declared in an Ina Jo specification. A state, $s \in State$, of a machine is a function:

$$s : M \rightarrow Val \tag{3.44}$$

where *Val* is a set of primitive semantics values. Let there be a class of all functions f,

$$(f : Dom^i \rightarrow Dom) \in Val \tag{3.45}$$

each mapping i-tuples of *Dom* into *Dom*. We consider simple state variables x as zero-placed functions, so that we have for $s \in State$,

$$s(x) \in Dom^0 = Dom \tag{3.46}$$

as primitive semantic values. This definition follows the intuitive concept that a state assigns values to all variables.

For Ina Jo state variable f of finite non-zero degree (parameterized variables, as mentioned earlier), we have:

$$s(f) \in Dom^{Dom^i} = Dom^{\langle v1,v2,\ldots,v_j\rangle} \tag{3.47}$$

Thus $s(f)$ assigns values to the arguments of f in the usual way, then evaluates f in state s. The type value of f is the type of the value of the range of f, as expected.

A binary state transition relation, $tr \in Trans$ is such that for every $s, s' \in State$, there is at least one $x \in M$, $\{v, v'\} \subseteq Val$ and $\langle x, v\rangle \in s$ such that:

$$tr(s, s') \iff s' = s[\langle x, v'\rangle / \langle x, v\rangle] \tag{3.48}$$

The notation $\langle x, v \rangle$ for $x \in M$ denotes the assignment of value v to variable x. The "/" denotes value replacement. Statement (3.48) states formally the informal notion that a state transition is defined by replacing some value of a state variable from one state to the next and that the new state assigns the new value to the changed variable.

The set of *immediate accessibility*, R, and *accessibility*, R^* is defined next.

$$R = \{\langle s, s' \rangle : \exists tr \in \mathit{Trans}, \langle s, s' \rangle \in tr\} \tag{3.49}$$

Definition (3.49) defines a binary relation on all immediate transitions from one state to the next for the entire model. The relation R^* is defined as the reflexive transitive closure of R, and thus contains all computation paths. A *computation path* is formally defined as a countable sequence s_j of states such that $tr(s_j, s_{j+1}) \in \mathit{Trans} \subseteq R$. Wing and Nixon require R to be total[4]which, together with the formal definitions, always require a successor state for any given state. Always requiring a successor state implies computational paths are countably infinite in length. Countably infinite path lengths seems to create the unusual convention, which Wing and Nixon do not discuss, that halting states repeat infinitely. For a halting state s_h, the definitions require $tr(s_h, s_h) \in R$.

To obtain a semantic interpretation for assertions in Assn (reference the grammar in Figure 3.12), a state-induced assignment function is defined:

$$A : \mathrm{Id} \times \mathit{State} \rightarrow \mathit{Val} \tag{3.50}$$

so that for all $s, s' \in \mathit{State}$ and $x \in \mathrm{Id}$:

$$A(x)s = s(x), \text{ for } x \in M \tag{3.51}$$

$$A(x)s = A(x)s', \text{ for } x \in (\mathrm{Id} - M) \tag{3.52}$$

Function A simply produces the value of x in state s. Definition (3.52) defines how to map a variable that is not in the machine M. It is provided for formal completeness and is not used in the development below.

The A-induced valuation function V maps terms in the assertion, a state, and the state's successor into values:

[4]The domain of R is the entire set *State*.

$$V : \text{Term} \times State \times State \rightarrow Val \tag{3.53}$$

Given terms x, $t_i, 1 \leq i \leq n$, and $f(t_1, \ldots, t_n)$, V is recursively defined as:

$$V(x)s\ s' = A(x)s \tag{3.54}$$

$$V(\mathrm{N}''x)s\ s' = A(x)s' \tag{3.55}$$

$$V(f(t_1, \ldots, t_n))s\ s' = s(f)(V(t_1)s\ s', \ldots, V(t_n)s\ s') \tag{3.56}$$

$$V(\mathrm{N}''f(t_1, \ldots, t_n))s\ s' = s'(f)(V(t_1)s\ s', \ldots, V(t_n)s\ s'). \tag{3.57}$$

Equation (3.54) says the valuation of a simple variable s is the value of the variable s in that state. An expression not involving the N'' operator does not use the value of s'. Equation (3.55) says the value of a simple variable under the N'' operator is the value of x in its next state. Equation (3.56) says the value of a function variable is defined by the mapping f applied to the valuation of its arguments. Equation (3.57) is a more subtle, and says the value of operator N'' applied to a function variable is the valuation of the function f in the next state but with the value of the arguments to f from the current state.

Finally, the foundation for the *Eval* function is complete. Function *Eval* maps an assertion and two states (current and next) into true or false:

$$Eval : \text{Assn} \times State \times State \rightarrow \{\text{true,false}\} \subseteq Val \tag{3.58}$$

A few examples of the use of *Eval* will illustrate its use. For the complete application of *Eval* to Ina Jo constructs, see [20].

For $t \in \text{Term} \cap \text{Assn}$, $t_1, t_2 \in \text{Term}$, $a, a1, a2, a3 \in$Assn:

$$Eval((t))s\ s' = V(t)s\ s' \tag{3.59}$$

$$Eval((t_1 = t_2))s\ s' = V(t_1)s\ s' = V(t2)s\ s' \tag{3.60}$$

$$Eval(\neg a)s\ s' = \neg Eval(a)s\ s' \tag{3.61}$$

For $\otimes$ as any binary operator in BinOp:

$$Eval(a1 \otimes a2)s\ s' = Eval(a1)s\ s' \otimes Eval(a2)s\ s' \tag{3.62}$$

$$Eval(a1 => a2 <> a3)s\ s' =$$

$$(Eval(a1)s\ s' \Longrightarrow Eval(a2)s\ s' \wedge (\neg Eval(a1)s\ s' \Longrightarrow Eval(a3)s\ s') \tag{3.63}$$

Equations (3.59) through (3.62) are straightforward statements of what a simple term, equality, negation, and binary operation means. Equation (3.63) defines the meaning of the "if–then–else" construct.

To extend the definition of *Eval* to temporal operators, we need to introduce some additional notation.[5] We will define $\bigwedge_{t \in R^*(s',t)}(P(t))$ and $\bigvee_{t \in R^*(s',t)}(P(t))$ to be the conjunction and disjunction, respectively, of $P(t)$ evaluated over all states in R^* from s' onward, where $P(t)$ is some predicate over state t. With this notation, the four most important temporal operators are defined as:

$$Eval(\mathrm{ah}''a)s\ s' = Eval(a)s\ s' \wedge (\bigwedge_{t \in R^*(s',t)} (Eval(\mathrm{ah}''a)s'\ t)) \tag{3.64}$$

$$Eval(\mathrm{eh}''a)s\ s' = Eval(a)s\ s' \wedge (\bigvee_{t \in R^*(s',t)} (Eval(\mathrm{eh}''a)s'\ t)) \tag{3.65}$$

$$Eval(\mathrm{av}''a)s\ s' = Eval(a)s\ s' \vee (\bigwedge_{t \in R^*(s',t)} (Eval(\mathrm{av}''a)s'\ t)) \tag{3.66}$$

$$Eval(\mathrm{ev}''a)s\ s' = Eval(a)s\ s' \vee (\bigvee_{t \in R^*(s',t)} (Eval(\mathrm{ev}''a)s'\ t)) \tag{3.67}$$

The other temporal operators can be defined in terms of the operators in Equations (3.64) through (3.67) in a similar manner.

Equation (3.64) defines "for all, henceforth" as the conjunction of the value of a in all subsequent states. Equation (3.65), defining "there exists, henceforth", requires a to be true at s and continue true down some branch of the state tree from s onward. Equation (3.66), defining "for all, eventually," requires a to be true somewhere down all branches of the state tree from s onward. Equation (3.67), defining "there exists, eventually," only requires a to

[5]This notation is a deviation from that used by Wing and Nixon. We find the original notation in [20] somewhat imprecise, although the intended meaning is clear. We believe our notation is clearer and more concise.

be true at some state subsequent to s. Ina Jo's state based treatment of temporal events is complete in the sense that it permits expression of predicate values at each state on all branches, one whole branch, somewhere on each branch or in any subsequent state of the state tree.

The axioms of Ina Jo's temporal operators are analogous to those of the S4 system with quantification except that temporal operators without their quantifiers are not defined. Defining quantification directly with temporal operators is one way of avoiding the problems of branching versus linear time expressibility that Lamport and others have encountered [21, 22].

Contrasts between Ina Jo's temporal system and S4 are illustrated by the two following examples of axioms. Ina Jo has the axiom:

$$\text{ev}''a \iff \neg\text{ah}''\neg a \tag{3.68}$$

The corresponding S4 axiom is **TDEF1**:

$$\Box a \iff \neg\Diamond\neg a \tag{3.69}$$

The difference between S4 and Ina Jo arises because the negation of universal quantification yields existential quantification. Ina Jo also has the axiom:

$$\text{ah}''(a \Longrightarrow b) \Longrightarrow (\text{eh}''a \Longrightarrow \text{eh}''b) \tag{3.70}$$

The closest corresponding S4 axiom is **TA2**:

$$\Box(a \Longrightarrow b) \Longrightarrow (\Box a \Longrightarrow \Box b). \tag{3.71}$$

If the axiom expressed by Equation (3.70) is correct as stated in Wing's paper [20] it expresses inclusion of an axiom that would be better proved as a theorem, if possible.

To see how axioms and inference rules can be used to check properties of a specification, consider a temporally based version of specification LIVE in Figure 3.13. Figure 3.13 specifies the same properties as the specification in Figure 3.11 except Figure 3.13 does not require the boolean variable *done*.

Consider two kinds of theorems that one may wish to show about a specification. The first kind demonstrates that initial conditions satisfy the state machine invariants. Let

```
specification LIVE
    variable
        x : integer;
    initial
        x > 0&ah''(an''(N''x = x − 1))
    criterion
        x > 0− > ev''(x = 0)
    transform decrement
        effect
            an''(N''x = x − 1)
end LIVE
```

Figure 3.13. Temporally based version of specification LIVE

IC designate the contents of the **initial** statement and *CR* designate the contents of the invariant **criterion** statement. Then the initial condition theorem takes the form:

$$\text{ev}''(IC \Longrightarrow CR) \Longrightarrow (IC \Longrightarrow CR) \tag{3.72}$$

The second kind of theorem demonstrates the correctness of the transforms. This theorem must take into account the pre-conditions in **refcond** for which we write R, and the post-conditions in **effect** written as E. The invariant must also be satisfied, therefore the correctness theorem takes the form.

$$R \wedge E \wedge CR \Longrightarrow \text{en}''CR \tag{3.73}$$

For the example in Figure 3.13, this would be written:

$$\text{an}''(\text{N}''x = x - 1) \wedge (x > 0) \Longrightarrow \text{ev}''(x = 0)) \Longrightarrow \text{en}''(x > 0 \Longrightarrow \text{ev}''(x = 0)) \tag{3.74}$$

Ina Jo's temporal logic specification system is far more expressive than any of the previous examples. Quantifiers permit any or all branches of execution sequences to be expressed. Since all the specification systems examined permit expression of nondeterministic actions, methods of specifying what *may* happen versus what *must* happen are important. Ina Jo's system handles both through existential and universal quantifiers.

The Barcan formula and reverse Barcan formula, discussed in Section 2.1, are both assumed valid in the Ina Jo system. This implies that the number of objects in an Ina

Jo specification does not change. As described in Section 2.1 this implies the need for predicates over the universe of objects to either mark objects as "included" or "excluded" at any point in time.

3.7 Specifying Temporal Properties Without Temporal Logic

Since all of the specification methods surveyed use temporal operators to simply make assertions about sequences of states or actions, or sequences in an input stream, and do not strictly rely on the sense of passing time, is there an alternative method that does not rely on modal logic? Alpern has presented a method [34] of specifying and verifying temporal systems without use of modal logic. Although application of his ideas become somewhat involved, the basic idea is simple and elegant: He replaces temporal operators with Buchi automata.

A Buchi automaton is a finite state machine that accepts or rejects infinite sequences of input symbols. As with regular automata, a Buchi automaton can be used to specify properties of an input sequence of symbols. In Alpern's method, the infinite sequence of symbols is replaced by an infinite sequence of states from some program under examination.

Edges between automaton states are labeled by program state predicates, called *transition predicates*, that define transitions between states. Each transition predicate is evaluated over program states read by the automaton. Figure 3.14 shows a state machine M_1 that after a finite prefix, accepts program states for which p is true. Any subsequent state for which p is not true takes the machine into a forever non-accepting state q_2. Machine M_1 is defined to accept the sequence if it stays in state q_1 for an infinite number of input symbols. In this case, p must remain true for all subsequent program states input to M_1.

Machine M_1 is equivalent to the temporal logic statement

$$\Diamond\Box p. \tag{3.75}$$

Alpern defines a program property as a sequence of program states over which some

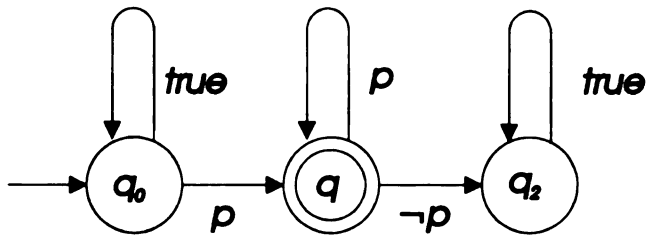


Figure 3.14. Buchi automaton

predicate holds true. As we have seen, formulas of temporal logic can also be interpreted as predicates over sequences of states. Each property of the system being specified can be written using an automaton. Let π denote the temporal property of a program. To prove a program satisfies all properties of a specification, one has to write and prove a conjunction of all properties of the program:

$$\Pi = \pi_1 \wedge \pi_2 \wedge \ldots \pi_n \tag{3.76}$$

Proving Π involves proving that each π_i separately is true. Alpern shows that each π_i can be written as the *disjunction* of formulas equivalent to automata:

$$\pi_i = D_1 \vee D_2 \vee \ldots D_p \vee \neg D_{p+1} \vee \ldots \vee \neg D_{p+n} \tag{3.77}$$

where each D_i is specified with a deterministic Buchi automaton. Automata D_1 through D_p are termed positive automata and D_{p+1} through D_{p+n} are called negative automata. A program property π_i is satisfied if one of the positive automata accepts the program state sequence or one of the negative automata rejects the state sequence.

The specification and proof process continues one level deeper by writing first order predicate statements equivalent to each automata. Next, for each automata, and thus for each first order predicate statement, three proof instruments are constructed:

1. an *invariant* clause that must always hold true and relates all program states of the program being verified.

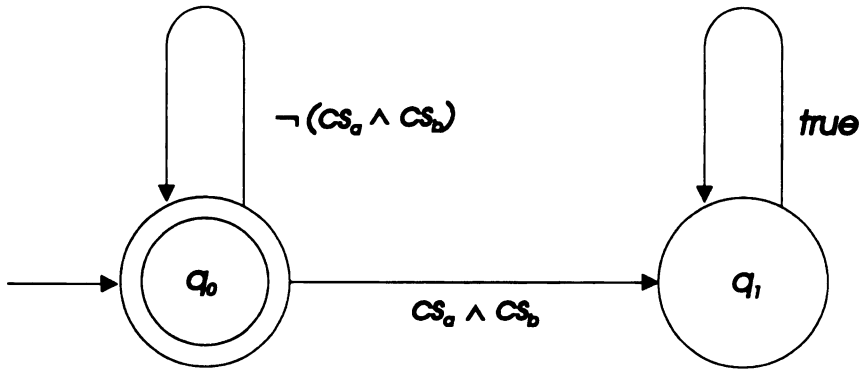


Figure 3.15. Buchi automaton specifying mutual exclusion

2. a *candidate function* that identifies negative automata that can never again enter an accepting state

3. a *variant function* that bounds the number of times the candidate function can become empty before a positive automaton enters an accepting state.

In practice, this method degenerates into a vast number of predicate formulas that combine to form a very complex series of first order formulas covering each program state. For example, a specification of mutual exclusion is shown in Figure 3.15. Variables cs_a and cs_b in Figure 3.15 represent the properties of being in the critical sections of two different routines. Mutual exclusion requires that neither routine can be in their critical sections when the other is in its critical section. In practice, "being in a critical section" involves specification of the program states associated with the critical section. While Figure 3.15 clearly specifies what is (or is not) to happen, the realization of formulas to show that this condition holds is complex.

Rewriting the automaton in Figure 3.15 in terms of the predicates from the edges and in terms of the proof obligations results in a large conjunctive formula over program states of the form:

$$pc_a = 1 \wedge cond_1 \wedge pc_a = 2 \wedge cond_2 \ldots pc_a = n \wedge cond_n \tag{3.78}$$

where pc_a represents the program state, or program counter value during execution for routine a, and $cond_i$ represents some condition, for example, not both being in a critical state, or looping while trying to enter the critical section. A similar form is required for routine b, then the conjunction of both must be considered to prove mutual exclusion.

While the automata ideas are interesting, the method amounts to replacing temporal operators by long sequences of disjuncts and conjuncts over states. This is analogous to replacing $\forall$ and $\exists$ in a similar fashion. The process works without appeal to temporal operators, but is very time consuming and complex in operation.

It is interesting to note Equation (3.75) in light of the discussion of the S4 versus S5 in Section 2.1. In Section 2.1 it was stated that this equation is equivalent to $\Box p$ in S5 but not in S4. The automaton in Figure 3.14 is clearly not equivalent to one without the states accepting the prefix. Implicitly then, this proof system is apparently operating under the axioms of S4.

3.8 Comparison of Systems

Specifying temporal properties of a system with temporal operators can be classed into two broad categories: Process algebras and algebraic deductive systems based on modal logic. Methods to specify temporal properties without temporal operators do exist but require very long and complex formulas to cover all states of the system.

None of the methods presented make a direct reference to time *per se*, but rather to events that occur through time. Each specification system has its own particular class of events. Process algebras mark time by the occurrence of interactions. The temporal logic of Ina Jo refers to state sequences. Agent-stream systems reference the members of the data stream. All of the methods are anchored in point-wise semantics requiring the equivalent of a clock "tick" to move the system forward to the next state. Properties of a system can only be discussed in association with some sort of transition event between states.

This form of temporal logic sidesteps the philosophical question of whether time is continuous or discrete. It also requires every event to be tied to a state transition. In a very

large distributed system the number of state transitions will be extremely large and state transitions may not be assumed to occur in a synchronous fashion. In large distributed systems, we will need to specify properties that hold over intervals of time without having to appeal to the lowest levels of state transition. Process algebras have the potential of doing this, but verification appears to come in the form of simulation rather than proof. Algebraic systems, such as Ina Jo's, have the tools but still appeal to states. One of the effects of the state oriented approach is the requirement of quantification operators, such as in Ina Jo, to cover alternate sequences of states.

CHAPTER 4

The Extension of Larch for Temporal Specifications

4.1 Introduction

Many formal specification languages are based on first-order logic [2, 5, 6]. Like first-order logic, formal languages have difficulty addressing properties of systems involving time. One goal for this investigation is to facilitate the specification of system behavior that includes timing elements in a well-defined formal language. Distributed systems are a particular focus of the specification system presented in this section.

Using an existing formal specification language as the basis of the specification system is another major objective of this work. Existing systems are likely to have had their notation, syntax, and semantics already proven correct. In contrast, creation of a completely new language would require extensive testing and debugging. Additionally, tools already exist for some formal languages, which enables extensions to the language to be in use quicker than if there were no tools. Valuable tools to support formal languages include syntax checkers and provers [35, 36, 37, 38, 39]. Syntax checkers help insure the unambiguous nature of the specification and provers are used to verify internal consistency and explore behavior.

Specifications must be somewhat intuitive to be useful. Elegant, complex, formal

systems often have powerful expression capability, but if the specification itself does not have obvious meaning in a paradigm readily grasped, then it could be prone to errors. Furthermore, complex notations with paradigms that are difficult to understand will probably not see widespread use for real problems. Another goal of this work is to permit the specification of temporal and system properties in a straight forward manner.

4.2 Properties of Time

Philosophers, physicists, mathematicians, and more recently computer scientists, have all had much to say about the nature of time. From a model-theoretical viewpoint Van Benthem [40] provides a good treatment of the subject. Three basic properties of time are frequently discussed:

1. Is time continuous or discrete?
2. Are there points in time or only periods?
3. What is the meaning of *simultaneity*?

A complete discussion of the merits of each viewpoint is not required here but may be found in [40]. The nature of time in terms of discrete events, the structure of intervals, and event ordering are important from a specification standpoint, and will be discussed next.

4.2.1 Discrete versus Continuous

In our specification system we consider time to be discrete and to occur in conjunction with unique events. There is no clock "tick" to reference events, and events can be arbitrarily close in time. The only events that exist are those that are generated from within the specification itself. This definition precludes, for example, inferring that there must be some unspecified event between two specified events. The only property of instants specified is *sequence*. The intent is that deductions from the theory found in a specification should follow only from what is stated, not by what is not stated. In addition, only observable features of the system being specified are axiomatized. If an event cannot be deduced

from what is specified about temporal properties, then we assume that we cannot reason about the event. This approach is in agreement with the Larch Shared Language algebraic specification system [1, 6, 41], which is described in the following section.

Furthermore, we assume that two events can never be simultaneous. Two instants that are equal in time are required to arise from the exact same event. From this standpoint, we consider time to be continuous in order to allow any two points in time to be arbitrarily close without being the same. Lamport argues similarly in [9, 10] based on relativity theory. Our convention is motivated by the pragmatic considerations of measuring simultaneity across a distributed system.

Lamport defines a "A precedes B" operator to describe the case where the time-lines for events A and B do not intersect, thus preventing A and B from being judged simultaneous by observation. He also defines "A can casually affect B" for the situation where A and B have intersecting time-lines and thus may be viewed as simultaneous by an observer. It has been shown [15] that for distributed systems, a global time model where everyone sees events occurring in the same order is equivalent to Lamport's system. This result means that we can treat events from a distributed system execution as a set of points in time that have absolute "before" and "after" relationships as long as the points are fully ordered. We assume global ordering of points in time, do not allow simultaneity to be observed, and do not assume the existence of a clock.

4.2.2 Points and Intervals

Points or intervals can be taken as the basic unit of time in order to derive the other. Van Benthem [42] provides a mapping between points and intervals. Allen [18] ignores points altogether defining occurrences, or events, in terms of periods. The association of events with points in time and the construction of intervals from successive points in time is easier and more intuitive than defining points in terms of intervals. Consequently, we assume the existence of points in time and associate events with them.

The event-to-point correspondence has received criticism because many "events" do not actually occur instantly. For example, turning on a light switch can be described as an

instant event, even though there is an interval of time required to make the circuit complete. On successive decomposition, the operation of the light switch can be described as a series of actions covering the period of time during which the circuit is made complete. In contrast, we often do not care about the decomposition of an action. At the system specification level, "turning on the light" can be classified as an event without further harm. This notion is not unusual and corresponds to ideas of atomicity and levels of detail. We often speak of an assignment statement instantly changing the state of a variable. At the very lowest levels of decomposition, the state change is not instantaneous nor does it consist of a single event. At the very lowest level, there is a transition time period while quanta of charge accumulate at the junctions of semi-conductor gates.

The use of points does not preclude the use of intervals if an interval is more appropriate. If we need to specify the individual component actions and events in the operation of a light switch, then we could define an event that is the start of the light switch operation and designate another as the ending event. The period of time between these two points, or events, is the interval of light switch operation.

Appropriately, we define an interval as a period of time starting with a beginning event and bounded by an ending event. Notice that the ending event is a bound and is not part of the interval. The purpose of defining an interval in this manner is to avoid dual membership of events in two successive intervals that meet. Consider the atomic assignment of a boolean variable. The variable starts at T_0 with the value 0. At T_{on} the variable changes state to the value 1. Assume the time T_1 terminates the period of consideration. During the interval $[T_0, T_{on}]$, the variable is set to 0. During the period $[T_{on}, T_1]$ the variable is set to 1. If the event T_{on} were owned by both intervals then the intervals would intersect in one point. At the common point in time, the implication is that the variable is both 1 and 0. We may argue that the variable is neither 1 nor 0 due to transition, but this assertion requires values for boolean variables other than 1 or 0.

4.2.3 Events, Intervals and Properties

Further extending the above discussion, *events* will be associated with points in time when a state change occurs or an action takes place. Since the specification can only display points in time connected with some observable event, such as a state change or property occurrence, and since inference of non-specified points in time are not permitted, events become the identifiable object associated with markers in time.

In our common everyday experience we often associate an event occurrence with the instant it occurs. We speak of the "time" something has occurred, but this sense of time is actually a comparison of the event sequence to the standard set of events being generated by a clock. We usually call the clock events "ticks". In this work, we do not use an absolute sense of time with a clock for two reasons:

1. A clock can be defined in the system instead of being an axiomatic part of the system.

2. In practice, global clocks in distributed systems are difficult to manage.

We will be able to determine event sequence, but the idea of the "time" something has occurred expressed as an absolute number will be absent.

A *property* will denote a proposition that can either hold or not hold over an interval of time. A property will usually be associated with the event that triggers the interval connected with the property. The predicate HOLD(p, I) is true if and only if predicate p is true during interval I. HOLD describes properties of the system under specification. HOLD(light = on, I) describes the property of the light being on during I.

The definition of an interval and what it means for a property to hold follows intuition. The general model is as follows: Some action occurs, thus marking the start of an interval. The same action causes a set of conditions that enable the property. The property now holds during the interval. Examples of conditions that may be set include the assignment of a value to a variable or a critical section of code being entered. Finally, a complementary action occurs, changing the conditions thus disabling the property, marking the end of the interval, and ending the period during which the property holds.

We next introduce important properties of the HOLD predicate. The HOLD predicate can be recursively defined to be true if it is true over all its sub-intervals. This property can be expressed formally as:

$$\text{HOLD}(p, I) \iff \forall J, J \subseteq I \Longrightarrow \text{HOLD}(p, J) \tag{4.1}$$

If predicates p and q both hold over an interval, then they hold separately over the interval.

$$\text{HOLD}(p \wedge q, I) \iff \text{HOLD}(p, I) \wedge \text{HOLD}(q, I) \tag{4.2}$$

Logical or also distributes across HOLD:

$$\text{HOLD}(p \vee q, I) \iff \text{HOLD}(p, I) \vee \text{HOLD}(q, I) \tag{4.3}$$

If a predicate p does not hold over an entire interval then we can assert that HOLD is false over the interval:

$$\text{HOLD}(\neg p, I) \Longrightarrow \neg\text{HOLD}(p, I) \tag{4.4}$$

Note that Statement (4.4) is not symmetric. If the left and right sides of the implication in Statement (4.4) are swapped, then we obtain:

$$\neg\text{HOLD}(p, I) \Longrightarrow \text{HOLD}(\neg p, I) \tag{4.5}$$

Statement (4.5) is not true. Non-intersecting sub-intervals, J and K of I may exist such that $\text{HOLD}(p, J)$ and $\text{HOLD}(\neg p, K)$. In other words, $\neg\text{HOLD}(p, I)$ implies that p is not always true over the entire interval I, thus allowing the possibility for a sub-interval of I during which p is true. Conversely, $\neg\text{HOLD}(p, I)$ implies that as long as interval I is not empty, some sub-interval of I when $\neg p$ is true must exist. This condition can be expressed as:

$$\neg\text{HOLD}(p, I) \Longrightarrow \exists J, (J \subseteq I) \wedge \text{HOLD}(\neg p, J) \tag{4.6}$$

Statement (4.6) claims that there is at least one sub-interval during which $\text{HOLD}(\neg p, J)$ is true. As we will see in the next section, properties built from predicates and intervals play an integral role in the specification method.

4.3 Extension of Existing Language with Temporal Notation

In this section temporal logic concepts presented in previous sections are incorporated into an existing specification language. Specifically, the Larch family of specifications are introduced in this section. Rather than introduce an entirely new language, Larch will serve as the host. The task for this section is to introduce definitions and axioms into Larch that satisfy the properties of time described in earlier sections.

4.3.1 Larch Specifications

Property-oriented formal specification methods define a system's behavior indirectly in terms of sets of properties the system is expected to exhibit. The properties are usually stated in terms of axioms that the system must satisfy. Property-oriented specification systems can be divided into axiomatic methods and algebraic methods. Axiomatic methods are based on Hoare's work on correctness proofs of implementations of abstract data types and use first-order predicate logic for preconditions and post-conditions. Algebraic methods specify behavior through sets of axioms written as equations in a heterogeneous algebra. Algebraic methods are also usually based on first-order logic with the addition of types.

Larch [1, 6] is a two-tiered, property-oriented method that combines both axiomatic and algebraic specifications. The axiomatic component specifies state dependent behavior in a language that is tailored to the expected final implementation programming language. The axiomatic component is called the *interface specification.* The algebraic component of Larch specifies state independent properties of data and programs in a common *Larch Shared Language* (LSL). A LSL specification introduces functions and operators that are used to construct preconditions and post-conditions in interface language statements. Several interface languages are available to match commonly available programming languages such as C [43] and Modula-3 [44]. In this paper, we limit our discussion to the LSL.

Units of LSL specifications are called *traits.* Each trait defines a theory, which is a set of formulas without free variables, in typed first-order logic with equality. In Larch

```
1 Stack(E, C) : trait
2     introduces
3         new :→ C
4         push : C, E → C
5         top : C → E
6         pop : C → C
7         isEmpty : C → Bool
8     asserts
9         C generated by new, push
10        ∀ stk : C, e : E
11            top(push(stk, e)) == e
12            pop(push(stk, e)) == stk
13            isEmpty(new)
14            ¬isEmpty(push(stk, e))
15    implies
16        C partitioned by top, pop, isEmpty
17        converts top, pop, isEmpty
18            exempting top(new), pop(new)
```

Figure 4.1. Example trait for a push down stack

terminology, types are called *sorts* to avoid confusion with similar terms found in programming languages. The trait in Figure 4.1 specifies the behavior required of the implementation of a push down stack. The trait consists of several parts. The **includes** clause in line 2 of Figure 4.1 brings in other traits required for axioms of this trait. Larch's modularization permits a complete specification to make use of many previously defined components. The Larch Handbook contains a rich library of commonly needed traits [6].

Lines 3 through 7 of Figure 4.1 contain signatures for the functions introduced in this trait. The **introduces** section provides the mapping of sorts to sorts for the functions used in the trait. Each function is considered to be exported for use in other traits or in an interface specification. The **asserts** section, starting in line 8, contains the axioms that define the main body of the theory. The **generated by** clause on line 9 provides hints necessary for generator-inductive proofs.

Formulas in the **asserts** section describe the exact behavior of the objects in the trait. In this case, the behavior of a stack is described. Formulas are always universally quantified

```
1  Eventobj : trait
2      introduces
3          new :→ eventobj
4          event : eventobj, evnt → eventobj
5          before : eventobj, evnt, evnt → Bool
6      asserts
7          eventobj generated by new, event
8          ∀ z : eventobj, a, b, c : evnt
9              before(event(z, c), a, b) == if c = a then false
10                 else if c = b then true
11                 else before(z, a, b)
12             ¬before(new, a, b)
13             a = b == ¬before(z, a, b) ∧ ¬before(z, b, a)
```

Figure 4.2. Event behavior from a container point of view

over variables in a **forall** clause, as in line 10. Formulas without the == operator are considered to be equated to *true*. Thus, line 13 contains the assertion that *isEmpty(new)* is always *true*. The **partitioned by** clause in line 16 of Figure 4.1, asserts that the operators listed can be used to distinguish between instances of sort C. The **converts** clause specifies that the operators listed are fully defined within this trait, except for cases listed in the **exempting** clause.

4.3.2 Temporal Notation for Larch

As described in the previous section, Larch is built on a sound basis of typed first-order logic with equality. Just as first order predicate logic requires modal operators to accommodate time, so does Larch. The extension of Larch with modal operators *henceforth* and *eventually* that are defined outside of Larch is due directly to Larch's basis in logic, and thus the inability to define these operators from logic (and Larch) constructs alone.

In Section 4.2.1, time was defined to consist of a series of events, no two of which are simultaneous. All events have a universal ordering in time, determined by the *before* operator, regardless of the observation point. *Before* is intended to have the same intuitive meaning as it normally has with respect to two events. The behavior expected from *before*

and events can be captured by a Larch trait.

The trait shown in Figure 4.2 describes the behavior of events as if they were stored in an *eventobj* container. A brief explanation of the trait body is in order: Since LSL is an algebraic specification language, axioms are expressed in a functional form. This notational form requires "unfolding" containers from the end to reach contained objects. "Unfolding" is performed in lines 9 and 11 of Figure 4.2 as follows: The first argument to *before* in line 9 is the *eventobj* obtained from adding c of sort *evnt* to the *eventobj* denoted z. If none of the clauses in lines 9 or 10 are satisfied, *before* is recursively invoked in line 11, but the first argument to *before* in the line 11 invocation is just z, as it was before c was added. To illustrate further, the form of an *eventobj* after adding a, b, and c to a new, empty *eventobj* would be:

$$event(event(event(new, a), b), c) \tag{4.7}$$

Thus, in the first argument of *before* in Line 9, z would have the value $event(event(new, a), b)$. This construction permits unfolding the container to access successively earlier and earlier events.

Lines 4 and 7 in the specification in Figure 4.2 specify that new events of type *evnt* are added to an event set with the *event*() function. Operator *before* determines sequence by moving backward from the last added event towards the first event. If b is found first while unrolling the container, then the function is true. Otherwise, the function is false. Function *before* judges earlier added events as before later added events. Event occurrence times are not mapped to a number and there is no sense of a clock to mark time. The trait only specifies how to determine event ordering. The only method for signifying that an event has happened is *event*. The specification also includes an unusual axiom that requires, for any two events a and b, $a = b$ implies the events are identical.

The *eventobj* trait is not suitable for inclusion into a trait library. Its constructions are cumbersome to use and a specification of more than one *eventobj* could lead to confusion. The *evnt* type also presents a problem. The intent is to capture the time of occurrence of state change, but state change is associated with some invocation of an axiom in a trait.

```
1  EventRules : trait
2      includes temporal
3      includes Integer
4      introduces
5          τ_ : Int → instant
6          τ_ : Bool → instant
7          _ < _ : instant, instant → Bool
8      asserts
9          ∀ a, b : Int, x, y : Bool, e1, e2 : instant
10             e1 < e2 == ¬(e2 < e1)
11             τx = τy == ¬((τx) < (τy)) ∧ ¬((τy) < (τx))
12             τa = τb == ¬((τa) < (τb)) ∧ ¬((τb) < (τa))
13             x ⇝ y ⇒ (τx) < (τy)
```

Figure 4.3. Axioms for behavior of *instants*

For example, suppose a predicate, P, eventually becomes true. The condition is expressed as $\Diamond P = true$, but if used as a parameter in $event(z, \Diamond P)$, a type mismatch in the second argument results. The expression $\Diamond P$ is type boolean (and not type *evnt*) with values *True* or *False*. The type required of the second argument is *evnt* which takes the value of the time when $\Diamond P$ becomes true. In order to define an operator that takes any sort and returns the time when the value changes (that is, the time of action), the trait in Figure 4.3 is introduced.

The trait shown in Figure 4.3 introduces the τ[1]operator that returns sort *instant*, the time when the event in its argument occurred. *Instants* are not mapped to integers, reals, nor are they in correspondence with any numbers. Values of type *instant* have the special properties of time described above. In addition, the trait in Figure 4.3 does not contain a reference to an *eventobj*. The trait assumes *instants* have semantics separate from a container. The reason for dropping the container is illustrated by considering the semantics attached to *event()* additions into two different containers. Since *before* requires an *eventobj* container in its invocation, and there is no defined relation between different *eventobj* containers, there is no defined relation between instants in different containers. Lack of a

[1]The τ operator we introduce here is entirely different from the τ operator of CCS discussed in 3.1.

```
1 temporal : trait
2     introduces
3         □_ : Bool → Bool
4         ◇_ : Bool → Bool
5         _ ⇝ _ : Bool, Bool → Bool
6     asserts
7         ∀ a, b : Bool
8             □a == ¬◇¬a
9             ◇a == ¬□¬a
10            a ⇝ b == □(a ⇒ ◇b)
```

Figure 4.4. Temporal (modal) constructs introduced into Larch

relation between containers, and instants in them, arises from defining the ordering relation between instants in terms of the sequence instants are added to a specific container. There is no global clock that could provide overall ordering across *eventobj* containers, and therefore, across all instants.

Figure 4.3 renames the *before* operator to $<$ and redefines the ordering relation for *instants* to exclude the reference to a container. The $<$ operator in Figure 4.3 behaves exactly as if *before* from Figure 4.2 were defined with a single, global *eventobj* container.

Expressions of any Larch sort may be associated with an event that we wish to specify. Since Larch is strongly typed (or strongly sort-ed), the τ operator must therefore be overloaded with every possible sort that τ may be used on. Extensive overloading of τ serves the useful purpose of recording all the sorts that produce instants significant to a specification.

The trait in Figure 4.4 introduces into Larch the new modal operators needed to specify henceforth, eventually, and causality. The interpretation of $\Box$ and $\Diamond$ are *henceforth* and *eventually*, respectively, as in Chapter 2. We also assume a linear point time model, as described in Section 4.2.2. The *temporal* trait provides the only mechanisms for moving axioms in a specification through time. Since $\Box$ and $\Diamond$ cannot be defined in terms of regular logic operators, they are defined, as in modal logic, in terms of each other.

It is often useful to causally connect properties by asserting that something will happen after something else has occurred. The *leads to* operator provides a convenient mechanism

for expressing this relationship. *Leads to*, denoted by $\leadsto$, is commonly found in contexts where liveness properties are specified [8]. The $\leadsto$ operator is defined as:

$$A \leadsto B \equiv \Box(A \Longrightarrow \Diamond B) \tag{4.8}$$

The right side of the expression states that from now on, whenever A becomes true then either B is true or will become true. Omitting the $\Box$ operator from the right side of Definition (4.8) produces a *leads to* operator that is far too weak. In order to understand why, contrast the expression

$$\Box(A \Longrightarrow \Diamond B) \tag{4.9}$$

with the expression

$$(A \Longrightarrow \Diamond B) \tag{4.10}$$

Expression (4.9) states that "at every instant from now on, A becoming true implies eventually B will become true." While (4.10) represents "*at this point in time* (and perhaps not at any others), A becoming true implies eventually B will become true." Expression (4.9) refers to all future times while Expression (4.10) only refers to this instant.

The Larch equation in line 10 of Figure 4.4, defines *leads to* exactly as does Expression (4.8), earlier.

With the ability to specify time and relationships between *instants*, the next building block required is an *Interval.* Figure 4.5 exhibits the trait that defines intervals. The trait introduces a function in line 4 to make sort *interval* from *instant* and two operators to extract the start ($\uparrow I$), and end ($\downarrow I$), of an interval. An interval is defined to contain all instants between its start and end, including the start but excluding the end. Predicates $\vdash$ and $\dashv$ define sub-interval "starts" and "ends" relationships, respectively. The difference between simple containment and the "starts" and "ends" relationships is that the latter require contained and containing intervals to share beginning or end. Lines 16 and 17 define the meaning of starting and ending an interval in terms of the events that constitute the interval.

```
1  Interval : trait
2      includes evntrule
3      introduces
4          [_,_] : instant, instant → interval
5          ↑ _ : interval → instant
6          ↓ _ : interval → instant
7          _ ⊆ _ : interval, interval → Bool
8          _ ∈ _ : instant, interval → Bool
9          _ ⊢ _ : interval, interval → Bool
10         _ ⊣ _ : interval, interval → Bool
11     asserts
12         ∀ a, b : instant, I, J : interval
13             ↑ [a, b] == a
14             ↓ [a, b] == b
15             (↑ I) < (↓ I)
16             J ⊢ I ==↑ I =↑ J ∧ (↓ J) < (↓ I)
17             J ⊣ I ==↓ I =↓ J ∧ (↑ I) < (↑ J)
18             a ∈ I == ((↑ I) < a ∨ (↑ I) = a) ∧ a < (↓ I)
19             I ⊆ J == (↑ J) < (↑ I) ∧ (↓ I) < (↓ J)
```

Figure 4.5. Trait specifying axioms for intervals

```
1  hold : trait
2      includes evntrule
3      includes interval
4      includes temporal
5      introduces
6          hold : Bool, interval → Bool
7      asserts
8          ∀ I, J : interval, p, q : Bool
9              hold(p, I) == hold(p, J) ∧ (J ⊆ I)
10             □q == hold(q, I) ∧ ¬((↑ I) < (τq))
11             (hold(p, I) ∧ hold(q, I)) ⇒ hold(p ∧ q, I)
12             (hold(p, I) ∨ hold(q, I)) ⇒ hold(p ∨ q, I)
13             ((τq) ∈ J ∧ J ⊆ I ∧ hold(p, I)) ⇒ hold(p ∧ q, J)
14             hold(p, [τp, τp]) == p
15
16
```

Figure 4.6. Trait Specifying Axioms for *hold* Predicate

The final trait needed to describe properties holding over intervals is found in Figure 4.6. A predicate *holds* and defines a property if the argument to the *hold* predicate is true over every sub-interval of the interval. The trait also indicates that the predicate that is the argument to *hold* is true at every instant within the holds interval. This relationship is expressed formally as:

$$hold(p, I) \wedge \tau q \in I \Longrightarrow \tau(p \wedge q) \in I \tag{4.11}$$

Expression (4.11) asserts that whenever q happens in I, the conjunction of p and q happens within I. In other words, p is true at every observable instant in I.

4.4 Dining Philosopher Example

In order to illustrate the use of the temporal traits in Larch, we introduce an example. The example deals with the well-known Dining Philosopher problem. In our statement of the problem, five philosophers are having a dinner party and are already seated at a pentagonal table. Each place is set with a plate, and forks are set between the plates. The problem to be solved is how to allow philosophers to eat using two forks. Philosophers dining in this fashion require careful coordination between neighbors to avoid conflicts. When not eating, a philosopher is thinking. It is obvious that this example is a mutual exclusion problem between dining partners.

Figure 4.7 exhibits a simple attempt at specifying the conditions of the problem. The trait maintains current states for forks, eating, and thinking in three boolean vectors defined in lines 8 through 10. Although eating and thinking are mutually exclusive and could be represented in one vector, using two vectors emphasizes the dual properties of eating and thinking and provides a better example.

First, a few mechanical details of the trait are presented. Each philosopher is identified with a number one through five. Rather than cluttering up the trait with details involving the mod function to handle wrap-around[2], the {} function defined on boolean vectors in

[2]The neighbors of philosopher 1 are (left) philosopher 5, and (right) philosopher 2. Likewise the neighbors of philosopher 5 are 4 and 1.

```
1  Phil_simple : trait
2      includes vector_5(vector)
3      includes hold
4      includes interval
5      includes evntrule
6      introduces
7          τ_ : vector → instant
8          forks :→ vector
9          eating :→ vector
10         think :→ vector
11         eat : Int → Bool
12     asserts
13         ∀ I, J, K : interval, i : Int
14             eat(i) == hold({think, i − 1}, I)
15                 ∧hold({eating, i}, I)
16                 ∧hold({think, i + 1}, I)
17             hold({eating, i}, I) ==
18                 hold({think, i − 1}, I)
19                 ∧hold({think, i + 1}, I)
20                 ∧(τsetleft(forks, i)) ∈ I
21                 ∧(τsetright(forks, i)) ∈ I
22             hold({think, i}, I) == hold(¬{eating, i}, I)
23             hold({eating, i}, I) ⇝ hold({think, i}, J)
```

Figure 4.7. Simple trait for Dining Philosophers

trait *vector*$_5$, handles this detail. Inclusion of the definitions for five-place boolean vectors is accomplished in line 2 of the trait. The complete definition of *vector*$_5$ and *numbers* is found in Appendix 1. Note that if we decided to have a different number of philosophers than five, only traits *vector*$_5$ and *numbers* would have to changed. As a final detail, the signature for τ from *vector* to *instant* is provided to ensure matching sorts throughout the trait.

The primary function of the trait is *eat(i)*. Function *eat(i)* evaluates true when it is permissible for philosopher i to eat. Line 14 of the trait indicates that philosopher i may eat when the philosophers on his right and left are both thinking, and hence not eating. The definition of property *eating* for philosopher i is contained in lines 17 though 21. Lines 17 through 21 equate property *eating* for a philosopher with the property *thinking* holding for philosophers on her left and right. Furthermore, lines 20 and 21 require forks to be picked up during the eating interval. Line 22 simply defines property *thinking* as not *eating*.

In an earlier version of the trait we had incorrectly added the conjunct:

$$hold(\neg(\{forks, i-1\} \wedge \{forks, i+1\}), I) \tag{4.12}$$

to the assertion on line 22. The intent was to specify that forks must be released while thinking. Tracing the complementary meaning of *eating* and *thinking*, it is easy to see that forks beside a thinking philosopher may correctly be in use. To specify that the fork to the left and right of a thinking philosopher are idle is incorrect.

The final axiom on line 23 ensures that a philosopher will not eat forever by asserting that eating eventually leads to thinking. Note that due to the absence of the complementary axiom, *thinking* leads to *eating*, this trait does not *require* eating.

Except for setting the *forks* vector, trait *Phil_simple* does not precisely convey how the transitions from *eating* to *thinking* to *eating* happen. Trait *Phil_complex* in Figure 4.8 defines these transitions more thoroughly. The primary change from Figure 4.7 is found in line 23 of Figure 4.8. Now, *eating* is asserted to lead to *thinking* and to *not eating* and to the return of forks to the table. Lines 25 through 27 specify that forks are returned during an interval of time while *thinking* and that the time be early in the thinking period. Line 27 contains

```
1  Phil_complex : trait
2      includes vector_5(vector)
3      includes hold
4      includes interval
5      includes evntrule
6      introduces
7          τ_ : vector → instant
8          forks :→ vector
9          eating :→ vector
10         think :→ vector
11         eat : Int → Bool
12     asserts
13         ∀ I, J, K : interval, i : Int
14             eat(i) == hold({think, i − 1}, I)
15                 ∧hold({eating, i}, I)
16                 ∧hold({think, i + 1}, I)
17             hold({eating, i}, I) ==
18                 hold({think, i − 1}, I)
19                 ∧hold({think, i + 1}, I)
20                 ∧(τsetleft(forks, i)) ∈ I
21                 ∧(τsetright(forks, i)) ∈ I
22             hold({think, i}, I) == hold(¬{eating, i}, I)
23             hold({eating, i}, I) ⇝
24             (hold({set(think, i), i} ∧ {unset(eating, i), i}, J)
25             ∧(τunsetright(forks, i)) ∈ K
26             ∧(τunsetleft(forks, i)) ∈ K
27             ∧K ⊢ J)
```

Figure 4.8. Refinement of Dining Philosophers to specify transitions

the critical condition requiring the action of forks being returned to occur in an interval K that starts the thinking interval J.

Requiring both forks to be returned to the table coincident with the start of the thinking period would appear to be the natural way to define *thinking*, but this assertion would require simultaneity. Since we do not permit different events to occur at the same time, we must specify the return of the two forks during an interval. We want the forks returned in a timely manner, so we specify that the interval when both forks are returned must occur at the start of the thinking period. The right side of the equation in Figure 4.8, lines 23 through 27, contains two intervals. Interval J corresponds to the time when *thinking* holds. Interval K is specified in line 27 of Figure 4.8 to be the early sub-interval when the forks are returned. In the absence of absolute time, and other constraints on the sub-interval K, the specification permits K to be almost as long as J. In an implementation, an arrangement of intervals such as J and K could mean that a situation where much more time is used putting the forks back than thinking with the forks available could still meet the specification.

The problem to be solved by this specification is mutual exclusion in order to permit sharing of a common resource. As a specification for mutual exclusion, Figure 4.7 contains a naive specification. Figure 4.8 is better but still has no transition periods. Mutual exclusion problems are composed of a non-critical section, an initialization section, a checking section, a section where mutual exclusion is achieved, a critical section, and a return to non-critical. The dining philosopher specification in Figures 4.7 and 4.8 jumps directly from non-critical to critical without periods for checking or resetting in between. A more detailed specification would contain checking periods between *eating* and *thinking*.

CHAPTER 5

Dual Modalities for Temporal Specifications

In this section, an alternate approach to specifying temporal systems is presented. The approach relies on the concept of time intervals without single points, or instants. To handle the problem of alternate future execution sequences, the use of dual modalites as a substitute for quantification is introduced. The two concepts, intervals and dual modal operators, provide the basis for a temporal logic that is more straightforward than a logic with quantification and avoids problems inherent when quantification and traditional modal logic are combined.

5.1 Introduction

Temporal specification systems have largely been based either on procedural, process algebra approaches or on some form of algebraic temporal logic. Temporal logics usually ignore the traditional passage of time as measured by a clock in favor of measuring progress through the occurrence of some class of event, usually state transitions. In this chapter we will consider a temporal specification method based on the modal logic system S4, which is described in Chapter 2, that does not rely on clocking time passage by state transitions. Instead, we will measure time directly. To make the temporal model compatible with the underlying specification logic, we will make time discrete by quantizing it into variable length intervals.

The model does not permit individual points in time. We will consider the shortest interval that can wholly contain an occurrence as the equivalent of a time point.

A frequently encountered problem in temporal specifications is specifying future events while also specifying alternate sequences of events. Branching time models handle both concurrently, but require quantification operators for full expressiveness. While quantification works, there must be something over which to quantify, usually states in the system. The challenge is to stipulate what should happen in alternate futures with a notation that relies on a traditional sense of time and is not state based. Above all, the specification method must be expressive enough to handle situations commonly found in distributed systems.

The remainder of this chapter is organized as follows: In Section 5.2 we introduce the interval model of time and provide its rationale. In Section 5.3 a temporal logic consistent with intervals and modal logic is defined. In Section 5.4 some examples of specification problems are presented. In Section 5.5, related work is considered and compared to the approach in this paper. We conclude in Section 5.6 and point out several directions for future work in Section 5.7.

5.2 Interval Model of Time

Time based portions of temporal specifications require describing two aspects of time: the structure of time during which system events occur, and the structure of occurrences within the time structure. For example, systems often go through an initialization sequence, followed by an execution phase, and conclude with a termination and a cleanup sequence. Here, three different time intervals are specified as serially composed, each with its own set of occurrences. Initialization is not the same as execution and must precede execution. Cleanup cannot occur before initialization and contains sets of occurrences different from the other two intervals.

Certain invariant conditions hold during initialization and events unique to initialization occur during this period. During the execution period, execution related invariants hold and other conditions are triggered and released by the sequence of operations occurring

during execution. Initial conditions or inputs to the system may determine what sequence of state changes occur, but the general structure of time periods containing these events remains unchanged. In other words, there is variability to the occurrence of events within intervals during the execution of a system, but the general interval framework of the system is unchanged.

Statements of the form "system operation consists of initialization followed by execution followed by cleanup" and "execution precedes cleanup" are examples of the *time structure* of a specification. Statements of the form "P is invariant throughout execution," and "when eventually X becomes true, Y will become and remain true" are examples of *occurrence structure* within the specification. The specification of time structure and occurrence structure are orthogonal and can be specified in a separate, but related notation.

Specification of time structure orthogonal to occurrence structure is analogous to specification of data structure separate from procedure structure in a program. A data structure has a special form that is suited for its use in a program. The form of the data structure is separate from its contents and the transitions that are caused by the procedure portion of the program. For example, a queue description requires elements to be in a certain order. The actual values each queue element can take during the use of the queue are a separate description independent of the algorithm using the queue. The algorithm has a separate, but perhaps related, description from the data structure. As a program is composed of both algorithms and data structures, a temporal specification is composed of both occurrences and interval structure.

Many specification systems intertwine time structure and occurrence structure by equating time passage with state changes. Such systems usually assume either an interleaving or synchronous model of time. In the former, any one component, but only one, is allowed to change state at each step. In the latter model, it is assumed that all components of the system perform one action at each step together. Neither model makes a distinction between the structure of time and the events that occur within the structure.

Our alternative to a ticking clock is a clock metric function that assigns a value to an interval in direct correspondence with the interval's duration in time. Events in the system

under specification are tied to intervals rather than point based "instants," as is common in state transition approaches. There are several advantages to this approach. First, the passage of time can still be measured even when no state transitions (or similar system events) are taking place. Second, the potential for modularization offered by this approach is great. A small interval considered atomic at a high-level view of the system could be decomposed into a series of intervals at a lower, more detailed view. As will be made clear below, all of time covering a system execution can be covered by non-intersecting contiguous intervals. Combined with a clock metric function, traditional clock ticks are not required.

We adopt Allen's model and semantics for intervals. In this model, time structure and precedence among the intervals can be completely specified with seven relations (or thirteen, including the inverses). Each of the seven relations, shown diagrammatically in Figure 5.1, is mutually exclusive of the others.

All relations are straightforward except *meets*. The *meet* relation, written $A \mid B$, is defined such that A is before B but no interval exists between A and B. Meeting intervals are also not allowed to share endpoints. *Meet* effectively makes time discrete by not allowing intervals between any two "meeting" intervals. Using a recursive construction, were "meeting" intervals allowed to have intervals between them, an infinite number of intervals could be inferred between *any* two intervals. This would imply that the clock metric function should assign a non-zero value to the intervening intervals, hence the meeting intervals would not really meet. Alternatively, one could assume zero length intervals to which the clock assigned zero (bringing the meeting intervals back together) but this seems of little use.

Expanding on our previous example, if we let *initial* represent the interval during which initialization takes place, *execution* represents the interval containing execution events and *cleanup* represent the termination phase of the system, then with the operators from Figure 5.1 we can write:

$$initial < execution < cleanup \tag{5.1}$$

or another description,

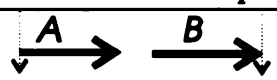
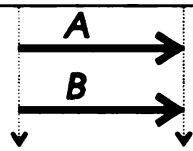
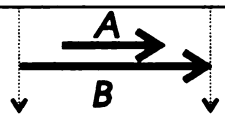
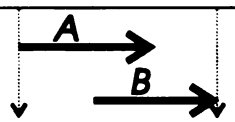
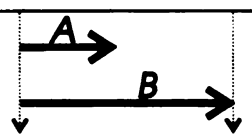
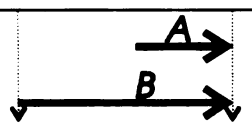
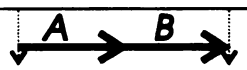
Relation	Symbol	Pictorial Example
A before B	$A < B$	
A equal B	$A = B$	
A in B	$A \sqsubset B$	
A overlaps B	$A \parallel B$	
A starts B	$A \trianglerighteq B$	
A ends B	$A \trianglelefteq B$	
A meets B	$A \mid B$	

Figure 5.1. Possible structural relations for time intervals

initial | *execution* | *cleanup.* (5.2)

Statement (5.2) requires all three phases of the program to flow smoothly together without break. Statement (5.1) permits a time lapse between phases of execution. A state transition based system would have no way to specify the difference between the two cases. If the above specification is part of a large specification in a distributed environment, the difference could become significant. For example, Statement (5.2) precludes interleaving events derived from any other part of the system occurring between *initial* and *execution* or between *execution* and *cleanup.*

Interval semantics raise the question of how to handle point events, which are often modeled as points in time. As hinted at above, our model only permits intervals, although arbitrarily small. At each level of detail in the specification we will use the convention of defining an event as the smallest interval that completely contains the actions required to complete the event. For example, writing to a memory location is composed of a complex series of actions at the hardware level. For most program specifications, this level of detail is far too fine. Since the memory write takes up some amount of time, it truly is not a point in time but rather occurs over a small, but measurable, interval. From the program specification level of abstraction, a memory write is too fine to decompose. We therefore define the memory write event as the smallest interval in which the write takes place. The memory write interval is not decomposed at the program level of specification while it may be at a finer level of detail.

Modeling atomic events as intervals permits modularization of the specification with respect to time. Consider memory access in a shared system. As described above, a memory write is not truly a point event. In a shared memory system the write interval, although short, may be significant because another access may be occurring at the same time. This implies events can somehow overlap. If our definition required point events, we would have to resort to a concept of simultaneity of point events. In contrast, interval semantics permits overlapped action without simultaneity. Further, we have the option of defining the events within the interval, that in a higher level, more abstract specification, represents an event.

Extending the example, we could define the memory write as a sequence intervals preparing the bus, transferring the contents of the memory cell, causing the actual write, releasing the bus, *etc.*

5.3 Temporal Logic Systems

Temporal logic provides the means for describing the *occurrence structure* of the specification. As described in Chapter 2, the majority of temporal logic operators are based on systems derived from modal logic. Two systems in particular, called S4 and S5, were introduced in Chapter 2 as containing axioms appropriate for reasoning about time.

S4 has been shown to be sound [24] with respect to a point-wise interpretation of time. State based progressions of time are based on time points, that is, the points when state transitions occur, and therefore S4 is the system of choice for state transition based time systems. S4 has likewise been shown to be **inconsistent** when used with continuous time, thus the time model must be based on discrete time for S4 to work. The system S5 is the system of choice for continuous time but as seen in Chapter 2, S5 suffers from requiring an interpretation "always" and "sometimes" for the two main modal operators. Such an interpretation does not allow for sequencing of events as does "eventually" and "henceforth."

Since the S4 system has been extensively studied [23, 24, 25], and is well understood, it is a good choice on which to base the temporal logic. Our use of S4 sets the interpretation of the temporal operators as *henceforth* written $\Box$ and *eventually* written $\Diamond P$. Unlike many systems, we will apply the temporal operators to intervals, as defined in earlier sections. The formula $\Box P$ is defined to mean P is now true and true in all subsequent *intervals.* Similarly, $\Diamond P$ is defined to mean P is either true now or in some future interval.

State-based time interpretations of time fall into two classifications called linear time and branching time. In a linear interpretation of time , "eventually P" ($\Diamond P$), means in some future state P will be true. Branching time, on the other hand, envisions a finite number of next state choices at each state that branch into a tree of potential futures. A path through this tree is called a *trajectory.* In Figure 5.2, the state sequence ABFQ forms

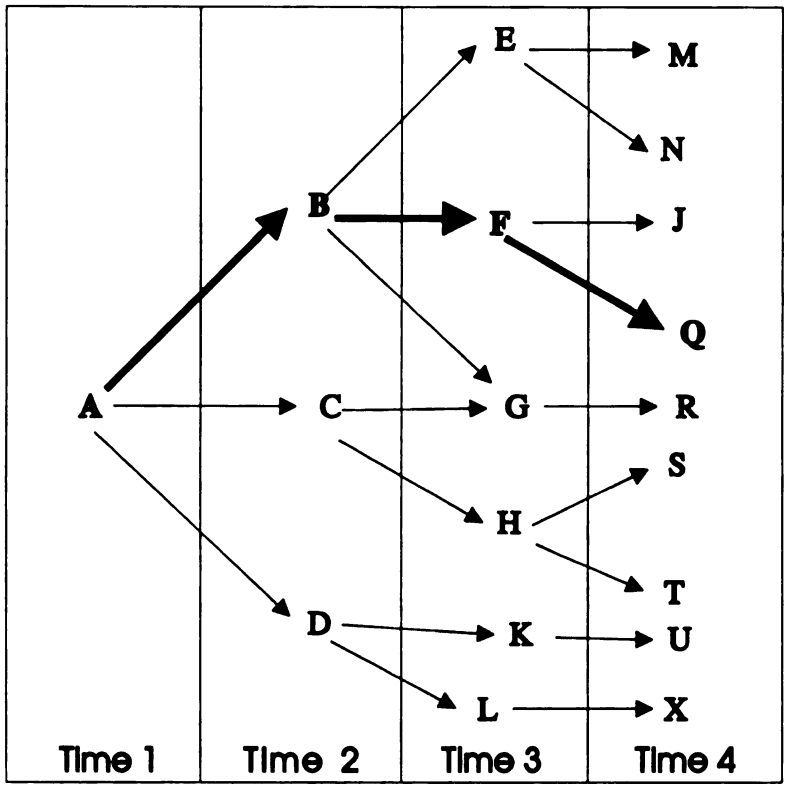


Figure 5.2. Branching time. Letters represent possible states of the system at times 1 through 4. The sequence ABFQ is one possible execution path but so is ADLX.

a trajectory. In branching time, the notation $\Diamond P$ is ambiguous and can either mean P will be true somewhere along *each* trajectory leading from the current state or along *some* trajectory from the current state. The usual interpretation is to require the statement be true along every trajectory. In Figure 5.2, *eventually* P, written $\Diamond P$, stated in relation to state A, means P will be true somewhere along every path leading from state A. But in branching time semantics, a statement of the form:

$$\Diamond\Box\neg \textit{enabled} \vee \Diamond \textit{executed} \tag{5.3}$$

meaning "eventually, *enabled* will henceforth no longer be true or eventually, *executed* will be true," creates problems. Statement (5.3) is a form of eventual fairness, requiring that if enabled infinitely often execution eventually occurs or else further enabling ceases. Statement (5.3) causes no problem for linear time but is not expressible in branching time. For the sub-statement $\Diamond\Box\neg$*enabled* to be true in branching time, every branch must have a point from which $\neg$*enabled* is true thereafter.

Wing and Nixon exhibited a solution to this problem by extending Ina Jo [20] to include universal and existential quantification applied to temporal operators throughout the tree. This capability permits the specification of an event occurring on *some* trajectories while not requiring it on all trajectories.

More importantly, branching time can only have meaning if there are nodes from which to form a tree. Since the nodes are state transitions, eliminating state transitions as a source of measuring time passage also eliminates the nodes where branching occurs.

The problem being addressed by quantifiers over branching time is how to accommodate two different aspects of future executions sequences. The first aspect is simple specification of how events will occur along any single execution trajectory. The second aspect is the description of alternate execution trajectories themselves. Referring to Figure 5.3, if we consider the two aspects as orthogonal dimensions, the first (horizontal) dimension is adequately handled with temporal logic as derived from system S4. The second dimension (vertical), in which quantification is the usual device, requires a new modal operator to specify the equivalent of *there exists* and *for all.* Since *for all* can be implied for all

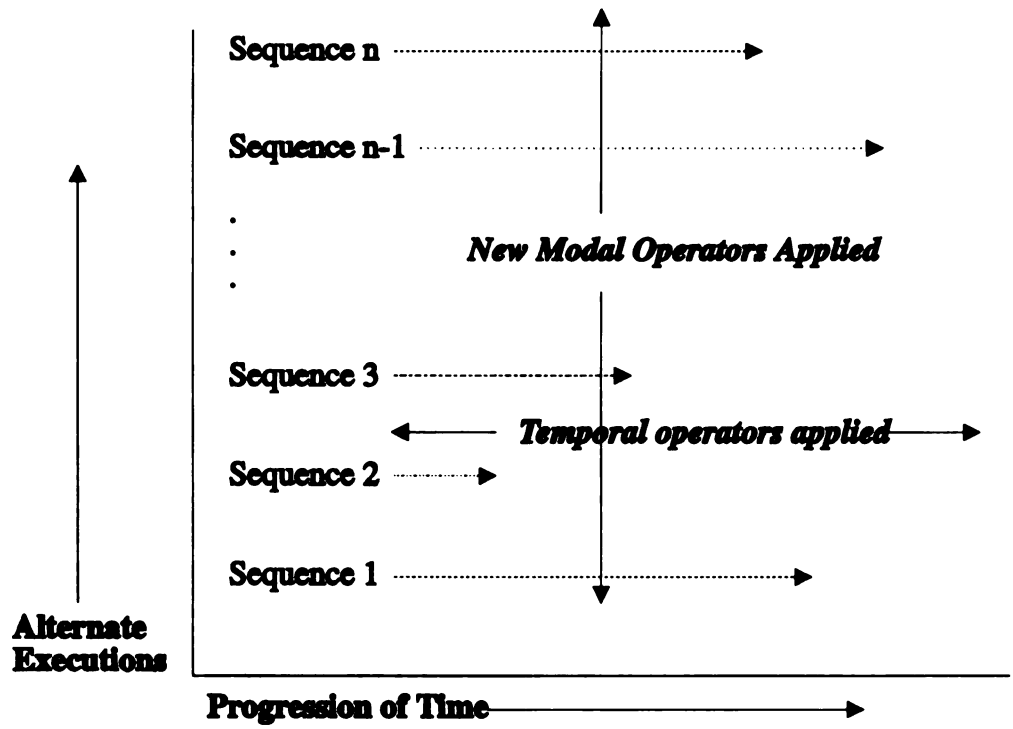


Figure 5.3. Two dimensions for modal operators. The X axis represents the regular progression of time. The Y axis represents alternate execution sequences, each with temporal properties.

trajectories, we only need an existential operator. The appropriate modal operator, already defined earlier, is *possibility.*

Modal possibility contains the idea that in at least one other world a proposition is true, while permitting other worlds where the proposition is false. For specification purposes, the other worlds are precisely alternate execution sequences. Possibility applied in compound with *eventually* can be interpreted to mean "in at least one execution sequence, eventually X will happen," but this also implies that X may *not* happen in other alternate execution sequences. Referring again to Figure 5.3, temporal operators are applied within each sequence extending along the horizontal axis. Modal operators are applied to all sequences along the vertical axis. In quantified branching time models quantification is

always required. When using modal operators, statements stand alone unless possibility is required.

To see why another operator is required, consider specifying alternate futures with the tools we have so far. Suppose we wish to express that although X may fluctuate between true and false, eventually it stabilizes and remains either true or false. A statement of the form:

$$\Diamond(\Box X \vee \Box\neg X) \tag{5.4}$$

clearly will not do. Statement (5.4) reduces to

$$\Diamond \mathit{True} \tag{5.5}$$

which is a tautology.

Statement (5.4) fails because it applies only to one execution sequence. Modal *possibility* solves this problem. To distinguish between *eventually*, written $\Diamond$, and *possibility*, we write *possible* as $\Diamond\!\!\!\!\!-$. The meaning of $\Diamond\!\!\!\!\!-P$ is P is true in at least one other execution sequence but does not have to be true in all execution sequences. We now write the intended meaning of Statement (5.4) as:

$$\Diamond\!\!\!\!\!-\Diamond\Box X \vee \Diamond\!\!\!\!\!-\Diamond\Box\neg X. \tag{5.6}$$

Statement (5.6) requires X to be true sometime in some trajectory. It also permits X to never be true in some trajectory and is even satisfied if X is always true in some execution. Statement (5.4) differs from Statement (5.6) in that $\Diamond\!\!\!\!\!-$ operates over trajectories while $\Diamond$ operates over time in a particular execution.

The companion operator to *possible* is *necessary*. In the discussion that follows, we write *necessary* as $\boxplus$ to avoid confusion with the henceforth operator, ($\Box$). *Necessary* P ($\boxplus P$), requires P to hold in all executions. When a condition holds in all executions, we may arbitrarily pick any execution to apply temporal operators. This is exactly equivalent to linear time without branches. In other words,

$$\boxplus\Diamond X \tag{5.7}$$

reduces to:

$$\Diamond X. \tag{5.8}$$

We may infer that any temporal statement preceded by *eventually* and *necessary* ($\boxplus$) is equivalent to the statement without *necessary*, or:

$$\vdash \boxplus\Diamond\alpha \rightarrow \Diamond\alpha \tag{5.9}$$

In any modal system, $\neg\diamondplus\alpha = \boxplus\neg\alpha$, therefore a statement of the form:

$$\neg\diamondplus\alpha \tag{5.10}$$

reduces to:

$$\boxplus\neg\alpha = \neg\alpha. \tag{5.11}$$

In other words, if temporal formula α does not occur in any trajectory then in all trajectories (and hence any trajectory) $\neg\alpha$ is true. For example, the statement "it is not possible that P is always true" can be written and reduced as follows:

$$\neg\diamondplus\Box P \tag{5.12}$$

$$\boxplus\neg\Box P \tag{5.13}$$

$$\neg\Box P \tag{5.14}$$

$$\Diamond\neg P \tag{5.15}$$

Statement (5.15) says eventually $\neg P$ will be true, which is equivalent to P not remaining true from now on (conveyed by Statement (5.12)).

Note that while a lone $\boxplus$ can be dropped, its negation cannot. Further, negation does not commute across $\boxplus$, for if it did, the reduction:

$$\diamondplus\Diamond P \rightarrow \neg\boxplus\neg\Diamond P \rightarrow \Diamond P \tag{5.16}$$

would be possible.

To see how these operators may be used, consider a frequently occurring situation: In proving a program correct, it is often helpful to reason using the principle that either a property P eventually is true or else it remains false. This statement is expressed in standard temporal logic as:

$$\Diamond P \vee \Box \neg P. \tag{5.17}$$

In Chapter 2 definition **TDEF1** was introduced as:

TDEF1 $\Box p \equiv \neg \Diamond \neg p$

Using definition **TDEF1**, Formula (5.17) becomes:

$$\Diamond P \vee \neg \Diamond P = \mathit{true}. \tag{5.18}$$

Statement (5.18) expresses little of value. The situation exactly suits the use of the new modal operator:

$$\diamondsuit \Diamond P \tag{5.19}$$

It is possible that eventually P is true. In at least one future sequence of events, P must become true at some time. This allows future sequences in which P never becomes true and sequences in which P is always true.

In Formula (5.17) difficulties arise when $\Diamond P$ includes the case $\Box P$. In other words, $\Box P \Longrightarrow \Diamond P$. Deleting the case where P is always true leads to:

$$\neg \Box P = \Diamond \neg P. \tag{5.20}$$

$\Diamond \neg P$ does not have quite the same sense as $\diamondsuit \Diamond P$ since the former *requires* P to be false sometime.

5.4 Examples of Common Specifications

In this section we exhibit specifications of common situations as expressed in our proposed system.

5.4.1 Property Becoming True

We begin with the simple situation where a property that is not true now will eventually be true. This is written as:

$$\neg P \wedge \Diamond P. \tag{5.21}$$

The left half of the conjunction requires P to false at the present time. If P were not constrained to be false currently, later to become true, then the right half of Statement (5.21) would be sufficient.

5.4.2 Fairness Conditions

Fairness conditions are occasionally expressed by stipulating that an enabling condition occurs continuously and eventually results in execution. The is expressed as follows:

$$\Box\Diamond \textit{enabled} \wedge \Diamond \textit{executed}. \tag{5.22}$$

The construction $\Box\Diamond$*enabled* means, from the definitions earlier, in every interval from now, it is true that *enabled* is or will be true in a subsequent interval. In other words, *enabled* occurs as often as possible. The double temporal operator construction $\Box\Diamond P$ could be a common idiom for a continuously activated condition. The first *henceforth* requires the statement $\Diamond P$ to be true in every subsequent interval. Without the leading *henceforth*, the condition could be enabled once without repetition. The right half of Statement (5.22) continues the statement with $\wedge\Diamond$*executed*, meaning there is a subsequent interval when *executed* is true. Since this portion of the statement lacks *henceforth*, execution is only required once. The entire conjunction requires *enabled* repetitively, eventually resulting in *executed*. The left half of the conjunction in Statement (5.22) does not stipulate how frequently *enabled* is true, other than "very often." This is a weakness of temporal-modal logic in general.

5.4.3 Latching Conditions

A latch has the property that once set it remains set. The latching property is written:

(1) $\Diamond\Box P \wedge \Box\neg P = True$: Assumption
(2) $\Diamond\Box P \wedge \neg\Diamond P = True$: Sub $\neg\Diamond P$ for $\Box\neg P$
(3) $\Diamond P \wedge \neg\Diamond P = True$: $\Box P$ is true and $\Box P \Longrightarrow P$ and 2. Contradiction.

Figure 5.4. Proof that $\Diamond\Box P$ and $\Box\neg P$ cannot both be true. If $\Diamond\Box P$ and $\Box\neg P$ are both true, then their conjunct must also be true. The proof shows this assumption leads to a contradiction.

$$\Diamond\Box P. \tag{5.23}$$

Specifying a latch condition requires compounding *henceforth* and *eventually,* as above, but in the opposite order from Statement (5.22). As written, Statement (5.23) *requires* latching to occur sometime. If latching is optional we could write:

$$\Diamond\Box P \vee \Box\neg P. \tag{5.24}$$

Figure 5.4 contains the proof showing why both halves of the disjunct in Statement (5.24) cannot be simultaneously true. Exclusive-or, defined as $(a \vee \neg b \wedge (b \vee \neg a)$, would be more appropriate than *or*. Written with exclusive-or, Statement (5.24) becomes:

$$(\Diamond\Box P \wedge \neg\Box\neg P) \vee (\Box\neg P \wedge \neg\Diamond\Box P). \tag{5.25}$$

Replacing $\neg\Box\neg P$ with $\Diamond P$ on the left side of the disjunct and $\Box\neg P$ with $\neg\Diamond P$ on the right side produces:

$$(\Diamond\Box P \wedge \Diamond P) \wedge (\neg\Diamond\Box P \wedge \neg\Diamond P) \tag{5.26}$$

which reduces to a tautology. Using the new possibility operator, the intention of both Statement (5.25) and Statement (5.26) can be written:

$$\blacklozenge\Diamond\Box P. \tag{5.27}$$

which is neither reducible nor a tautology.

5.4.4 The Leads-to Operator

We introduce the *leads to* operator for specifying one property causes another. *Leads to* is defined and written as follows:

$$P \rightsquigarrow Q \equiv \Box(P \Longrightarrow \Diamond Q). \tag{5.28}$$

Leads to is a stronger form of implication than $\Longrightarrow$ and is designed to connote the notion that *P's* occurrence causally affects Q. Ways of writing causally related implication are not new. While it is true that a condition on the left side of *leads to* may have nothing to do with the event on the right side, we assert that *leads to* specifies observed behavior well enough for formal specification purposes. In the definition of *leads to*, the leading *henceforth* requires *P's* occurrence to imply *eventually* Q at every interval of time from now on.

5.4.5 Mutual Exclusion

Mutual exclusion is a common specification problem that often finds itself embedded in other specifications. The problem description entails several processes competing for exclusive control of a resource, as signified by one process, and only one process, entering its critical section. Each process consists of a noncritical section, *NCS*, a trying section, *TRY*, from which exclusive control is granted, and a critical section, *CS*. We define boolean variables *inNCS*, *inTRY*, and *inCS* to be true only when a process is in its non-critical section, trying section, and critical section, respectively. Liveness requirements, *i.e.*, the property that something must eventually happen, can be specified as:

$$(inTRY \rightsquigarrow inCS) \wedge (inCS \rightsquigarrow inNCS). \tag{5.29}$$

Statement (5.29) requires that once in the trying section, a routine must eventually gain exclusive control. This specification does not require a routine in a noncritical section to ever enter the trying section, however once entered, the right side of the conjunction requires a return to the noncritical section.

The structure of the intervals associated with mutual exclusion is specified using interval operators. In a notation similar to above, let *iNCS*, *iTRY*, and *iCS* represent the intervals

when a routine is in its non-critical section, trying section, and critical section, respectively. Statement (5.30) expresses the condition that there is no gap in time between trying and entering the critical section:

$$iTRY \mid iCS. \tag{5.30}$$

If Statement (5.30) is written:

$$iTRY < iCS \tag{5.31}$$

A period of time between trying and entering the critical section is allowed. State transition semantics always implies the meaning in Statement (5.30).

For higher levels of abstraction, we may wish to define the mutual exclusion action as a single event. We define an interval, $iMUTEX$, as:

$$iMUTEX = iTRY \mid iCS \tag{5.32}$$

which implies the validity of:

$$(iTRY \trianglerighteq iMUTEX) \wedge (iCS \trianglelefteq iMUTEX) \wedge (iTRY \mid iCS) \tag{5.33}$$

Since Equation (5.32), Equation (5.33), and Statement (5.29) define the smallest interval in which mutual exclusion can occur, $iMUTEX$ could be used in a higher level specification requiring mutual exclusion as a component. For example, in an application that reads a value from one shared memory location and writes it to another, the time structure could be written as:

$$(iMUTEX|iREAD) < (iWRITE|\neg iMUTEX) \tag{5.34}$$

where $\neg iMUTEX$ denotes releasing mutual exclusion.

Temporal operators can be combined with time structure operators to infer properties in a specification. From the definition of $\Box P$ we know that if $A \sqsubset B$ and $\Box P$ holds for B then $\Box P$ must also hold for A. If $A \parallel B$ and $\Box P$ is true in A, then P is true some of the time in B, written for interval B as $\Diamond P$. To make the context of the temporal operator explicit we introduce the notation $\langle op \rangle^{\langle interval \rangle}$. For example, we write:

$$\Box^A P \tag{5.35}$$

to mean the temporal operator *henceforth* ($\Box$) applies to interval A, and thus P is true at least throughout interval A.

Denoting iCS_n as the interval when routine n is in its critical section, we can reason about overlapping critical sections as follows:

$$(iCS_1 \parallel iCS_2) \wedge \Box^{iCS_1} P \tag{5.36}$$

then we know that:

$$\Diamond^{iCS_2} P \tag{5.37}$$

must be true. If P represents a property that requires exclusivity, Statement (5.36) and Statement (5.37) imply exclusivity is violated. Similarly, given

$$(A \mid B) \vee (A \parallel B) \wedge \Box^A P \tag{5.38}$$

we know that P is true during all of A and, since interval A may overlap interval B it is possible that P may be true during some part of interval B. This is written as:

$$\Diamond\Diamond^B P. \tag{5.39}$$

5.5 Related Work

Most temporal specification systems either are process algebras or have a form close to process algebras. Examples include CSP [26], CCS [27, 28], and LOTOS [29]. Our goal was to produce a system that could be incorporated into a strictly algebraic deductive system. None of the process algebras are deductive and all lack the properties of deductive systems. Explicit temporal operators are not contained in process algebras because they are not needed. The structure of the specification and the nature of communication between agents and processes implies temporal structure.

Virtually all specification systems that contain temporal specification capability rely on some class of system event for the passive of time. Usually, the event is a state transition

and therefore, temporal operators are applied to state sequences. This is exactly the case with the logic of Manna and Pnueli [12] and with most real time specifications, for example Ostroff's [13]. Even in Broy's stream abstraction [32], temporal operators are applied to data sequences flowing through the stream network. Although Wing and Nixon come closest to the compounded modal operators in extended Ina Jo [20], their approach still is oriented toward state transitions. Indeed, their definitions of ten temporal operators are written using a set R^*, which consists of the reflexive transitive closure of all reachable state pairs in the system being modeled.

5.6 Conclusion

A new way of specifying and reasoning about time using intervals combining both temporal and modal operators has been presented. Temporal and modal operators have the same roots and are essentially the same operators applied to different domains. The domain of temporal operators is intervals of time. The domain of the modal operators is alternate future sequences of events. We suggest that these operators in combination have expressive power equivalent to, or better than, quantification in branching time logic and linear time logic.

The time model presented is superior to occurrence class based measures of time passage because it measures time passage as external and separate from state transitions. Time passage is measured by implicit reference to a global clock, but we attempt to eliminate reference to the clock's tick rate by using interval constructions. Intervals offer semantics consistent with the modal S4 system and permit modularization. Points in time are not required and thus assumptions of atomicity are also not required. Lack of atomicity means, among other things, that neither an interleaving nor a synchronous time model of a distributed system need be followed. Occurrences can be specified directly and intuitively.

5.7 Future Work

Interval time semantics offer the possibility of modularizing at least the time structure portion of a specification. Since modularization for temporal specifications is itself a research area, this work could be extended to define a solid modularization technique.

While none of the intervals in this paper have specific lengths of time attached to them, there is no reason why this could not be so. Many distributed systems, and most real time systems, require specifications containing constraints expressed in actual time. It is not unreasonable to require, for example, that mutual exclusion in some system be obtained within 40 milliseconds. This could be specified in this system, with a few additions, by requiring the period $iTRY$ to be less than 40 milliseconds long.

While we have introduced and demonstrated the use of composed modal operators, the full extend of a double-modal system has not been explored. We assumed the existence of the $\boxplus$ operator even though it was not discussed. As proposed in Section 5.3, $\boxplus$ can be removed when not negated. Even so, there may be a useful interpretation for $\boxplus$ that has not yet been examined.

Finally, the interval semantics, intervals operators, temporal operators and modal operator need to be combined into an algebraic system useful for full specification of distributed systems. We have suggested one notation for stipulating the interval to which a temporal operator applies. There may be a better way to approach this problem. Ultimately, successful algebraic specifications of complex distributed systems will require a formal, intuitive language to which automatic theorem provers and other automated tools can be applied.

CHAPTER 6

Conclusions and Future Work

We have presented two different methods of specifying temporal properties of systems. The language presented in Chapter 4 is explicitly based on the Larch formal specification language. We have extended Larch to mostly preserve the original language and the capability to use automated Larch tools. Traits dealing specifically with the temporal Larch extensions were carefully constructed in Larch. There are a number of disadvantages to the way we have extended Larch. Foremost is that Larch is particularly well suited to structural specification problems, such as the specification of data containers. It is less well suited to problems found in distributed systems such as mutual exclusion. The concept of time-points also presents problems. Our τ operator (not the τ of CCS), time points, and the restriction of not allowing events to be inferred without explicit specification were borrowed from Larch's similar approach to inferring facts about the object under specification. Such an approach may not be suitable for time specifications. When two events occur separated in time it is natural and correct to infer that there is an interval in between, and that something may be happening in that interval. Our approach in Chapter 4 precludes such assumptions.

We abandoned time points in Chapter 5 in favor of time intervals to avoid the problems of atomicity and the inability to infer events not explicitly part of the specification. We also saw an opportunity to permit modularization of a specification through intervals. A 'short', atomic interval in a high level specification can contain an entire specification at a more

detailed level. Additional levels of detail can continue indefinitely. Even with the power of intervals, the approach still has problems. For example, more work needs to be done to rigorously define how modal-temporal operators interact with intervals. Most of the work on modal logic [12, 24, 25, 40] has dealt with time *points*, not intervals. The parallel between second modal operator, *possibility*, introduced in Chapter 5 and existential quantification has been explored by A. N. Prior [45] but again, only in connection with point-wise time. State based approaches, atomic events and time-points are used in the literature because they are intuitive. The correct combination of temporal operators and time paradigm has yet to be definitively demonstrated.

Future work is required in several areas. Most importantly, a complete axiom set for dual mode temporal logic needs development. Before any inference system, automatic or manual, can be designed, the axioms and theorems for dual mode logic must be completed. Second, the equivalence of *possibility* and *necessity* with quantification needs to be formally demonstrated. 'Equivalence' may be too strong a term since what is needed is to show that modal operators are at least as expressive as quantification but without introducing the problems inherent with quantification.

Once a formal basis for dual modalities has been established, suitable automated tools should be constructed. These hopefully can come from modifying existing inference engines, although the only example of an inference engine to handle modal operators has been the Wing and Nixon Ina Jo system [20]. As has been stated several times, we firmly believe the only hope for practical use of formal verification, especially in temporal problems, is through extensive automation. Even simple proofs such as found in the Wing and Nixon paper on Ina Jo [20] are much too complex to do manually[1]

A hybrid approach from Chapter 4 and Chapter 5 that replaces intervals with time-points while retaining dual modal operators shows promise. Time points and modal operators, singly and in combination, have a large amount of supporting literature and may be more intuitive. We can foresee that such an approach would re-introduce the problem

[1] Wing and Nixon apparently feel the same way. Many of the proofs presented in the paper were completed in at least a semi-automated fashion.

of atomic action (always a problem in distributed systems), but such an approach would simplify the dual modal axiom set.

Finally, what is most needed is a suitable sample problem sufficiently large to exercise the ideas in this work while small enough to be manageable. These are often mutually exclusive requirements, but development of a system through overly simple examples is frequently a reason such systems are not used in practice. Even fairly small problems are sufficiently complex in a distributed environment that it is likely that modularization of the specification technique will be required at the outset. This implies that the specification system, including tools, be fairly well developed before the problem is attacked.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. V. Guttag, James J. Horning, and Andres Modet. Report on the Larch Shared Language: Version 2.3. Technical Report DEC SRC 58, Digital Systems Research Center, April 1990.

[2] Cliff B. Jones. *Systematic Software Development Using VDM.* Prentice Hall International Series in Computer Science. Prentice Hall International (UK) Ltd., second edition, 1990.

[3] Betty H.C. Cheng. Synthesis of Procedural Abstractions from Formal Specifications. In *Proceedings of COMPSAC'91: Computer Software and Applications Conference*, September 1991.

[4] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[5] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[6] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report DEC SRC 5, Digital Systems Research Center, July 1985.

[7] A. D. Birrell, J. V. Guttag, J. J. Horning, and R. Levin. Synchronization Primitives for a Multiprocessor: A Formal Specification. Technical Report DEC SRC 20, Digital Systems Research Center, August 1987.

[8] Leslie Lamport. A Simple Approach to Specifying Concurrent Systems. Technical Report DEC SRC 15, Digital Systems Research Center, December 1986.

[9] Leslie Lamport. The Mutual Exclusion Problem: Part I—A Theory of Interprocess Communication. *Journal of the ACM*, 33(2):313–326, April 1986.

[10] Leslie Lamport. The Mutual Exclusion Problem: Part II—Statement and Solutions. *Journal of the ACM*, 33(2):327–348, April 1986.

[11] D. B. Skillicorn and J. I. Glasgow. Real-Time Specification Using Lucid. *IEEE Transactions on Software Engineering*, 15(2):221–229, 1989.

[12] Zohar Manna and Amir Pnueli. The Modal Logic of Programs. In *Automata, Languages and Programming: Sixth Colloquim.* Springer-Verlag, 1979.

[13] Jonathan S. Ostroff. *Temporal Logic for Real-Time Systems.* Research Studies Press LTD., 1989.

[14] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[15] Shai Ben-David. The Global Time Assumption and Semantics for Concurrent Systems. In *The 12th International Conference on Distributed Computing Systems*, pages 223–231. IEEE Computer Society and Information Processing Society of Japan, June 1992.

[16] John Turek and Dennis Shasha. The Many Faces of Consensus in Distributed Systems. *IEEE Computer*, pages 8–17, June 1992.

[17] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[18] James F. Allen. Towards a General Theory of Action and Time. *Artificial Intelligence*, 23:123–154, 1984.

[19] James F. Allen and Patrick J. Hayes. A Common Sense Theory of Time. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 528–531, August 1985.

[20] Jeannette M. Wing and Mark R. Nixon. Extending Ina Jo with Temporal Logic. *IEEE Transactions on Software Engineering*, 15(2):181–197, February 1989.

[21] E. Allen Emerson and Joseph Y. Halpern. "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[22] Leslie Lamport. "Sometime" is sometimes "not never"—On the temporal logic of programs. In *Proceedings of the 7th Annual ACM Symposium on Principles of Programming Languages*, pages 174–185. ACM, New York, January 1980.

[23] Brian F. Chellas. *Modal Logic – An Introduction.* Cambridge University Press, 1980.

[24] G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen and Co. LTD, 1968.

[25] D. Paul Snyder. *Modal Logic and its applications.* Van Nostrand Reinhold, 1971.

[26] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[27] A.J.R.G. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer-Verlag, 1980.

[28] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[29] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[30] L. Logrippo, A. Obaid, J. P. Briand, and M. C. Fehri. An Interpreter for LOTOS, A Specification Language for Distributed Systems. *Software—Practice and Experience*, 18(4):365–385, April 1988.

[31] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1.* Springer-Verlag, 1985.

[32] Manfred Broy. Specification and Top-Down Design of Distributed Systems. *Journal of Computer and System Sciences*, (34):236–265, 1987.

[33] J. Schied and S. Anderson. *The Ina Jo specification language reference manual.* System Development Corporation, Santa Monica CA, Tech Rep TM-(L)-6021/001/01 edition, March 1985.

[34] Bowen Alpern and Fred B. Schneider. Verifying Temporal Properties without Temporal Logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, January 1989.

[35] James Horning and John Guttag. Larch shared language checker. Private communication.

[36] S.J. Garland and J.V. Guttag. A guide to lp, the larch prover. Technical Report TR 82, DEC SRC, December 1991.

[37] Michael R. Laux, Robert H. Bourdeau, and Betty H.C. Cheng. An integrated development environment for formal specifications. In *Proc. of IEEE International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, July 1993.

[38] J.M. Spivey. Type-checker for Z Specifications. Commercially available tool from author.

[39] K.E. Eriksen and S. Prehn. Raise overview. Technical Report RAISE/CRI/DOC/9/V5, Computer Resources International, 1991.

[40] J. F. A. K. Van Benthem. *The Logic of Time*, volume 156. D Reidel Publishing Company, 1983.

[41] Jeannette M. Wing. Using Larch to Specify Avalon/C++ Objects. *IEEE Transactions on Software Engineering*, 16(9):1076–1088, September 1990.

[42] J. F. A. K. Van Benthem. *The Logic of Time*, volume 156, chapter I.4, pages 80–99. D Reidel Publishing Company, 1983.

[43] J. V. Guttag and J. J. Horning. Introduction to LCL, A Larch/C Interface Language. Technical Report DEC SRC 74, Digital Systems Research Center, July 1991.

[44] Kevin D. Jones. LM3: A Larch Interface Language for Modula-3 A Definition and Introduction. Technical Report DEC SRC 72, Digital Systems Research Center, June 1991.

[45] A. N. Prior and Kit Fine. *Worlds, Times and Selves.* University of Massachusetts Press, Amherst, 1977.

APPENDICES

APPENDIX A

This appendix contains the traits for boolean vectors and description of numbers. Larch is an algebraic specification language requiring a precise definition of everything in traits, including the meaning of integer constants.

```
1  vector5(V) : trait
2      includes numbers
3      includes Integer
4      introduces
5          {_, _} : V, Int → Bool
6          [_, _, _, _, _] : Bool, Bool, Bool, Bool, Bool → V
7          set : V, Int → V
8          unset : V, Int → V
9          setleft : V, Int → V
10         setright : V, Int → V
11         unsetleft : V, Int → V
12         unsetright : V, Int → V
13     asserts
14         V generated by [_, _, _, _, _]
15         V partitioned by {_, _}
16         ∀ a, b, c, d, e, val : Bool, i : Int
17             {[a, b, c, d, e], i} == if i = 0 then e
18                 else if i = 1 then a
19                 else if i = 2 then b
20                 else if i = 3 then c
21                 else if i = 4 then d
22                 else if i = 5 then e
23                 else if i = 6 then a
24                 else false
25             set([a, b, c, d, e], i) == if i = 1 then [true, b, c, d, e]
26                 else if i = 2 then [a, true, c, d, e]
27                 else if i = 3 then [a, b, true, d, e]
28                 else if i = 4 then [a, b, c, true, e]
29                 else if i = 5 then [a, b, c, d, true]
30                 else [a, b, c, d, e]
31             unset([a, b, c, d, e], i) == if i = 1 then [false, b, c, d, e]
32                 else if i = 2 then [a, false, c, d, e]
```

```
33              else if i = 3 then [a, b, false, d, e]
34              else if i = 4 then [a, b, c, false, e]
35              else if i = 5 then [a, b, c, d, false]
36              else [a, b, c, d, e]
37          setleft([a, b, c, d, e], i) == if i = 1 then [a, b, c, d, true]
38              else if i = 2 then [true, b, c, d, e]
39              else if i = 3 then [a, true, c, d, e]
40              else if i = 4 then [a, b, true, d, e]
41              else if i = 5 then [a, b, c, true, e]
42              else [a, b, c, d, e]
43          setright([a, b, c, d, e], i) == if i = 1 then [a, true, c, d, e]
44              else if i = 2 then [a, b, true, d, e]
45              else if i = 3 then [a, b, c, true, e]
46              else if i = 4 then [a, b, c, d, true]
47              else if i = 5 then [true, b, c, d, e]
48              else [a, b, c, d, e]
49          unsetleft([a, b, c, d, e], i) == if i = 1 then [a, b, c, d, false]
50              else if i = 2 then [false, b, c, d, e]
51              else if i = 3 then [a, false, c, d, e]
52              else if i = 4 then [a, b, false, d, e]
53              else if i = 5 then [a, b, c, false, e]
54              else [a, b, c, d, e]
55          unsetright([a, b, c, d, e], i) == if i = 1 then [a, false, c, d, e]
56              else if i = 2 then [a, b, false, d, e]
57              else if i = 3 then [a, b, c, false, e]
58              else if i = 4 then [a, b, c, d, false]
59              else if i = 5 then [false, b, c, d, e]
60              else [a, b, c, d, e]
```

```
1 Numbers : trait
2     includes Integer
3     introduces
4         1 :→ Int
5         2 :→ Int
6         3 :→ Int
7         4 :→ Int
8         5 :→ Int
9         6 :→ Int
10    asserts
11        ∀ i : Int
12            1 : Int == succ(0)
13            2 : Int == succ(1)
```

14 $3 : \mathit{Int} == \mathit{succ}(2)$
15 $4 : \mathit{Int} == \mathit{succ}(3)$
16 $5 : \mathit{Int} == \mathit{succ}(4)$
17 $6 : \mathit{Int} == \mathit{succ}(5)$