CMU-CS-83-146

Theoretical Issues in the Design and Verification of Distributed Systems

Aravinda Prasad Sistla

August 1983

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-83-146

Theoretical Issues in the Design and Verification of Distributed Systems

Aravinda Prasad Sistla

Department of Computer Science,
Carnegie-Mellon University
August 1983

Submitted to Harvard University in
partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

This research was supported by NSF grant MCS-81-05553 at Harvard University and by NSF grant MCS-82-16706 at Carnegie-Mellon University.

Synopsis

With the rapid decrease in the cost of hardware distributed computing is finding wider application. The parallelism inherent in distributed processing makes it much more difficult to design reliable systems. Many software development techniques such as hierarchical design and exaustive testing used for large sequential programs are no longer adequate because of the high degree of nondeterminism present in parallelism. This thesis addresses the two aspects of correctness and performance in the design of distributed and concurrent systems.

In chapters 2 through 5 we consider different temporal logics and their extensions, as formal systems for reasoning about concurrent programs. In chapter 2 we investigate the complexity of decision procedures for different versions of Propositional Linear Temporal Logics(PTL). We present a space efficient decision procedure for the full logic. We also present optimal decision procedures for other restricted versions of this logic. We investigate the problem of automatic verification of *simple* concurrent programs using correctness specifications given in PTL. PTL can not express many important correctness properties of concurrent programs. For this reason in chapter 3, we extend PTL by allowing quantifiers over propositions. We investigate the complexity of decision procedures for these logics. We show that for a *weaker* version of this logic, there is a tight space complexity hierarchy with the number of alternations of quantifiers, for the set of valid sentences in this logic. In chapter 4, we consider a branching time temporal logic for automatic verification of finite state concurrent processes. We present efficient algorithms for the automatic verification of finite state concurrent programs using specifications given

in this logic. We show how this method can be applied to check the correctness of well known practical problems like the *Alternating Bit Protocol* and a solution to a mutual exclusion problem. In chapter 5, we extend linear temporal logic by introducing spatial modalities. This logic allows us to speak about temporal and spatial properties in a unified logical system. We show how this logic can be used for reasoning about many problems in fixed connection multiprocessor networks.

In chapters 6,7 we investigate correctness and performance issues in the area of inter-process communication. In chapter 6, we explore the possibility of using linear temporal logic for characterizing and axiomatizing different buffered message passing systems. We prove that all bounded buffers are characterizable and axiomatizable in linear temporal logic. We show that unbounded FIFO buffers are in general not axiomatizable in PTL, while unbounded LIFO and unbounded unordered buffers are axiomatizable. In chapter 7, we consider the problem of distributed implementation of *fair communication* among a set of processes that communicate through *rendezvous*. Specifically, we consider distributed implementation of Hoare's Communicating Sequential Processes that ensure certain fairness properties. We introduce two different fairness properties: *weak fairness*, *strong fairness*. For a natural class of algorithms that ensure weak fairness, we prove a non-trivial lower-bound on the time complexity of any algorithm in this class. We present near optimal algorithms in special cases. We also give new better algorithms for ensuring the above fairness properties.

Acknowledgment

I am extremely grateful to all my teachers who taught me in my educational career.

I am especially grateful to my advisor, Prof. Edmund M. Clarke for suggesting the problems addressed in this thesis, for his untiring efforts, lots of enthusiasm and zeal in directing my research work. The final form and contents of my thesis owe much to his able guidance, insightful comments and excellent judgement. His advice on both technical and nontechnical matters was invaluable. I am honored to be one of his doctoral students.

To Professors Amir Pnueli and John Reif I express my sincere appreciation for agreeing to serve on my research committee. I also thank Professors Michael O. Rabin and Albert Meyer for their insightful suggestions. I am also grateful to professors John Reif and Nissim Francez for the many of discussions I had with them.

The Center for Research in Computing Technology provided the right environment for my research work. I am also thankful to the administration of the Computer Science Department at Carnegie-Mellon University for giving me the facilities during my stay of one year there.

Special thanks are to all my friends especially Umeshwar Dayal, Allen E. Emerson and Anil K. Nori for their encouragement throughout my research work. I am thankful to Bhubaneswar Mishra for reading parts of my thesis.

Finally, I express my sincerest gratitude to my parents,brothers and sisters for their constant support and encouragement.

Table of Contents

Chapter 1

Introduction

Due to the rapid decrease in the cost of hardware distributed computing is finding wider application. The parallelism inherent in the distributed processing makes it much more difficult to design reliable systems. Many software development techniques such as hierarchical design and exaustive testing that have been used for large sequential programs are no longer adequate because of the high degree of nondeterminism present in parallelism. The inadequacy of these techniques becomes more apparent when the system under construction should continue function even in the presence of partial hardware failures.

In this thesis we address the two important aspects of correctness and performance in the design of distributed and concurrent systems. We consider formal systems based on temporal logic for reasoning about concurrent processes. We investigate the complexity of decision procedures for different temporal logics and the complexity of automatically verifying simple concurrent programs. We also present a very practical system for automatically verifying finite state concurrent programs using specifications given in a version of temporal logic. We introduce extensions of temporal logics that are useful in specifying large collection of concurrent processes.

We also investigate correctness and performance problems in the area of inter-process communication. The two types of communication we deal with are *buffered message passing*

systems and process communication through *rendezvous*(or unbuffered communication). In the former case, we explore the possibility of using temporal logic for characterizing and axiomatizing different buffered message passing systems. In the rendezvous type of communication, we develop efficient distributed algorithms that ensure certain *fairness* properties in communication among a set of concurrent processes. Specifically, we consider distributed implementations of Hoare's *Communicating Sequential Processes* [Ho78], that ensure various fairness properties.

Linear temporal logic was introduced in [Pn77] as an appropriate formal system for reasoning about parallel programs. This logic permits the description of a program's execution history without explicit introduction of time. Many important correctness properties of concurrent programs like mutual exclusion, deadlock freedom and absence of starvation can be elegantly expressed in this system. Proving that a parallel program satisfies some correctness property consists of deducing the formula expressing the property from *program axioms* which characterize the possible interleaving of the atomic statements of the individual processes. An important special case occurs when the programs are finite state. In this case the program axioms and the correctness properties can be specified in the propositional version of the logic, Propositional temporal logic (PTL). PTL is also used in automatically synthesizing concurrent programs as in [Wo82]. In the second chapter, we consider different decision procedures for different temporal logics. For the full PTL we give a polynomial space bounded decision procedure for satisfiability and show that the satisfiability problem for this logic is PSPACE-complete. The previous decision procedure given for this in [Wo81] is tableau based and requires exponential space. For a fragment of PTL which uses only the temporal operator F(*eventually*) we prove a linear size model theorem and present a decision procedure in NP. We also consider the complexity of *truth in a structure* for different versions of PTL.

Though PTL is widely used, it can not express many important correctness properties. For this reason we extend PTL by introducing quantifiers over propositions to get QPTL(*Quantified PTL*). We show that the set of true sentences in QPTL which are in normal form with a single quantifier alternation, is decidable using exponential space. We also consider a logic WQPTL, which is same as QPTL except that in all it's models all the propositions are false throughout the future after certain instance. We show that there is a tight space complexity hierarchy with the number of alternations of quantifiers for the set of true sentences in normal form of this logic. WQPTL is as expressive as WS1S(*Weak Monadic Theory of One Successor*). No such tight hierarchy is known for WS1S.

In the traditional approach to concurrent program verification the proof that a program meets it's specification is constructed by hand using various axioms and inference rules in a deductive system. The task of proof construction is in general quiet tedious, and a good deal of ingenuity may be required to organize the proof. Mechanical theorem provers have failed to be of much help due to the inherent complexity of the simplest logics. In chapter 4, we argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model theoretic approach which will mechanically determine if the system meets it's specification. The global state graph of a concurrent system can be viewed as a finite kripke structure. In chapter 2 we showed that automatically checking if such a system meets a specification given in PTL is very hard. In this chapter we use a version of branching time temporal logic called *Computation Tree Logic*(CTL) introduced in [EC80], as a specification language. We modify the semantics of the logic so that only *fair* computations are considered. For this logic, we give an efficient algorithm(with complexity linear in the size of the specification) which takes a global state graph, and a specification given in the above logic, and checks if the specifications are met by the global

state graph. We illustrate how this method can be used to automatically verify the Alternating Bit Protocol and a mutual exclusion problem.

When verifying parallel programs, which involve a large collection of processes temporal logic may be very cumbersome to use and in some cases may be inadequate. For these reasons in chapter 5, we introduce a modal logic which can be used to reason about synchronous and asynchronous fixed connection multiprocessor networks such as VLSI. In addition to the temporal modalities it has spatial modalities as well. The temporal modalities used are *until, eventually* and *nexttime.* The spatial modalities used are *somewhere, everywhere, across such and such connection.* The spatial modalities allow us to relate properties of the current state of a process with the current states of the other processes, while the temporal modalities allow us to relate the current state of a process with the succeeding states of the process. We give examples of the diverse applications of our logic to *packet routing, firing squad problems* and *systolic algorithms.* We also consider the decidability issues of the different versions of the logic.

Exchange of information between executing processes is one of the primary reasons for process interaction. Many distributed systems implement explicit message passing primitives to facilitate intercommunication. Typically, a process executes a *write* command to pass a message to another process, and the target process accepts the message by executing a *read* command. The semantics of *write* and *read* may differ depending on the method used for buffering messages that have been sent but not yet received. In chapter 6, we consider the possibility of *characterizing* and *axiomatizing* the different message buffering mechanisms in linear temporal logic. Specifically, we consider FIFO, LIFO and *unordered* buffers. The set of distinct messages that can be written into the buffer is the message alphabet. We specify a message buffer as the set of all valid infinite input/output

message sequences. Characterizing a message buffer consists of obtaining a formula that is true exactly on these sequences. We show that bounded buffers over a finite alphabet are characterizable in PTL. We prove that we can not give a *domain independent characterization* of unbounded buffers in first order temporal logic, but such a characterization can be given for bounded buffers. A *model* of a buffer is an infinite sequence of states denoting a series of legal read/write operations on the buffer. The theory of a message buffer is the set of all PTL formulae that are true in all models of the buffer. Since bounded buffers are characterizable in PTL, they are axiomatizable. We show that unbounded FIFO buffers over an alphabet of cardinality ≥ 2, are not axiomatizable. In fact we prove that their theory is Π_1^1-complete. Surprisingly we prove that unbounded unordered buffers and LIFO buffers are axiomatizable and in fact their theories are decidable.

Communicating Sequential Processes(CSP) was introduced in [Ho78] as an appropriate Programming Language for distributed systems. The original semantics of CSP did not require *fairness* in the selection of processes waiting to establish communication. However, in practice such a restriction may be highly desirable. In chapter 7, we consider the problem of implementing such a fairness property in CSP which allows input as well as output statements in the guards of alternative commands. We introduce a formal model for this and consider two different fairness properties; *weak fairness*, *strong fairness*. For example, in weak fairness we require that if two processes are willing to communicate continuously then they should eventually establish communication. We consider algorithms for distributed schedulers that ensure the different fairness properties. In this model neighboring schedulers can talk to each other using shared variables, each of which can be updated by only one process. We give simple global algorithms to ensure the fairness

properties. Next we consider algorithms in which the interaction between the schedulers is restricted in a natural way. For these algorithms we present an $O(\gamma^2)$ lower-bound on the time complexity of any algorithm that ensures weak fairness where γ is the chromatic number of the communication graph. In the special case when the communication graph is a complete graph we present a near optimal algorithm that ensures weak fairness. After this we consider algorithms with improved interaction between the scheduler processes. In this model we present better algorithms for weak fairness and also give algorithms for strong fairness.

In chapter 8, we conclude with remarks and open problems.

Chapter 2

Complexity Of PTL

2.1. Introduction

Linear Temporal Logic was introduced in [Pn77] as an appropriate formal system for reasoning about parallel programs. This logic permits the description of a program's execution history without the explicit introduction of program states or time. Moreover, important correctness properties such as mutual exclusion, deadlock freedom, and absence of starvation can be elegantly expressed in this system. Proving that a parallel program satisfies some correctness property consists of deducing the formula for that property from *program axioms* which characterize the possible interleaving of atomic statements of the individual processes. An important special case occurs when the program is finite state. In this case, the program axioms and correctness specification can be expressed in the *propositional* version of the logic and provability becomes *decidable.* A number of researchers (e.g., [MW81]) have attempted to use such a decision procedure for constructing correct finite-state programs.

In this chapter we examine the inherent complexity of decision procedures for *validity, satisfiability,* and *truth in a particular structure* for propositional logics with the temporal operators F (eventually), G (globally), X (nexttime), U (until) and S (since). We first consider the logic L(F) in which F is the only temporal operator. We prove a *linear size model theorem* from which a nondeterministic polynomial time bounded decision procedure

for satisfiability can be obtained. It immediately follows that satisfiability is NP-complete for L(F). This result is surprising since it shows that the set of satisfiable formulae in L(F) is no higher in the complexity hierarchy than the set of satisfiable formulae in ordinary propositional logic.

It is to be observed that we can not obtain an elementary decision procedure for propositional linear temporal logic by translation into the language of the structures $(N, <, P_1, P_2, \ldots)$ where N is the set of natural numbers, $<$ is the natural ω-ordering and $P_1, P_2, \ldots$ are monadic predicates, as it is shown in [Ro] that any decision procedure for satisfiability of formulae in the later logic has to be non-elementary. A tableau based decision procedure for propositional linear temporal logic was given in [Wo81]. However this procedure requires exponential space. We give a polynomial space bounded decision procedure for satisfiability of formulae in L(U,S,X). We show that satisfiability for the logics L(F,X), L(U), L(U,X), L(U,S,X) and for the extended temporal logic given in [Wo81] is PSPACE-Complete. These results are surprising because all of these logics have different expressive powers (some are more powerful than others).

Finally, we consider the question whether a temporal formula is true on some path starting from a node of an *R-structure.* R-structures model finite state parallel programs. We show that the above problem is NP-complete for L(F) but is PSPACE-complete for the other above mentioned logics. The corresponding problem for branching- time logics has been shown to be in P [CE81].

This chapter is organized as follows: Section 2.2 defines the syntax and semantics of the linear temporal logic that we use in the remainder of the chapters. In Section 2.3 we prove the linear size model theorem for L(F) and the corresponding NP-completeness

results. Section 2.4 contains the PSPACE-completeness results for L(F,X), L(U), and L(U,S,X). In Section 2.5 we show how our results can be extended to the extended logic given in [Wo81].

2.2. Notation and Basic Definitions

We use the following convention for symbols:

P,Q,R,...	denote atomic formulae and are drawn from the set $\mathcal{P}$.
f,g,h,...	denote formulae.
s,t,u,...	denote finite or infinite sequences. We always assume $s = (s_0, s_1, \ldots)$.
S,T,W,...	denote structures.

If $O_1, \ldots, O_k \in \{X,F,G,U,S,Y\}$ are distinct operators then $L(O_1, \ldots, O_k)$ denotes the propositional temporal logic restricted to these operators, e.g. L(F,G), L(X,F,G), etc.

A *well-formed formula* in propositional linear temporal logic is either an atomic proposition or is of the form $\neg f_1$, $f_1 \wedge f_2$, Xf_1, $f_1 \, U \, f_2$, Yf_1, $f_1 \, S \, f_2$ where f_1, f_2 are well-formed formulae. In addition, the following abbreviations will be used:

$f_1 \vee f_2 \equiv \neg(\neg f_1 \wedge \neg f_2)$, $f_1 \supset f_2 \equiv \neg f_1 \vee f_2$,

$Ff \equiv$ True U f, $Gf \equiv \neg F \neg f$.

Let $\tilde{L}(F,X)$ be the logic that uses the boolean connectives $\wedge,\vee$, the temporal operators F,X and with negations allowed only on the atomic propositions.

A *state* is a mapping from the set of atomic propositions into the set {True, False}. An *interpretations* is an ordered pair (t, i) where t is an ω-sequence of states and $i \geq 0$ is an integer specifying the present state. We define the truth of a formula f in an interpretation (t,i) ($t, i \models f$) inductively as follows:

$t,i \models P$	where P is atomic iff $t_i(P)$ = True;
$t,i \models f_1 \wedge f_2$	iff $t,i \models f_1$ and $t,i \models f_2$;
$t,i \models \neg f_1$	iff not$(t,i \models f_1)$;
$t,i \models Xf_1$	iff $t,i+1 \models f_1$;
$t,i \models f_1 U f_2$	iff $\exists k \geq i$ such that $t, k \models f_2$ and $\forall j$ $i \leq j < k$, $t,j \models f_1$;
$t,i \models Yf_1$	iff $i>0$ and $t, i-1 \models f_1$;
$t,i \models f_1 S f_2$	iff $\exists k \leq i$ such that $t, k \models f_2$ and $\forall j$ such that $k < j \leq i$ $t,j \models f_1$;

X,U,Y,S are the "nexttime", "until", "last-time", and "since" operators respectively.

We define the semantics so that $F f_1 \equiv$ True Uf_1 and $Gf_1 \equiv \neg F \neg f_1$. Length (f) denotes the length of the formula f and SF(f) is the set of sub-formulae of f or their negations after eliminating double negations. We assume that $\mathcal{P}$ is finite at many places.

Though, we have defined a state to be mapping associating truth values for each atomic proposition, in the present chapter we differentiate between a state and the associated mapping. We use a different notation for convenience. A Structure S = (s, ξ) where s = $(s_0, s_1, \ldots)$ is an ω-sequence of states and $\xi : \{s_0, s_1, \ldots\} \rightarrow 2^{\mathcal{P}}$. Intuitively, ξ specifies which atomic propositions are true in each state. We also assume that all the states appearing in the sequence of a structure are all distinct. An interpretation is pair (S,s_j) where S is a structure defined as above. Since we have assumed all states in S to be distinct, any state in S uniquely defines it's position. It is easily seen how we can go from the earlier interpretation to the present interpretation. The truth of a formula f in the new interpretation is defined exactly the same as we did in the previous interpretation.

An R-structure T is a triple (N,R,η), where N is a finite set of states, $R \subseteq N \times N$ is a total

binary relation (that is $\forall t \in N \exists t' \in N$ such that $(t,t') \in R$), and $\eta : N \rightarrow 2^{\mathcal{P}}$. A *path* p in T is an infinite sequence $(p_0,p_1, \ldots)$ where $\forall i \geq 0$, $p_i \in N$, and $(p_i,p_{i+1}) \in R$. Throughout this chapter for a path p in a R-structure $T = (N,R,\eta)$ we let S_p denote the structure (s,ξ) where $\forall i \geq 0$, $\xi(s_i) = \eta(p_i)$.

The global behavior of a finite state parallel program can be modelled as an R-structure. In the R-structure each path starting from the initial state represents a possible interleaving of executions of the individual processes in the program. In many cases, the correctness requirements of the concurrent system can be expressed by a formula of propositional linear time logic. The system will be correct iff every possible execution sequence satisfies this formula; i.e., every path beginning at the initial state in the corresponding R-structure satisfies the formula. For these reasons the following problem (which we call the *determination of truth in a R-structure*) is important in verifying finite state parallel programs:

Given a R-structure T, a state $p_0 \in N$, a formula $f \in L$, is there a path p in T starting from p_0 such that $S_p,s_0 \models f$?

2.3. The Complexity of L(F)

Let $S = (s,\xi)$ be a structure and let $s'' = (s_j,s_{j+1} \ldots)$ be the maximal suffix of s such that for each s_k in s'' the following condition holds:

$\forall \ell \exists i$ such that $i > \ell$ and $\xi(s_i) = \xi(s_k)$,

that is, there exist infinitely many states in s'' which have the same assignment of atomic propositions, as s_k. It is easily seen that such an s'' exists (because $\mathcal{P}$ is finite), and s'' is unique. Let $s = s' \cdot s''$. Define *init(s)* = s', *final(s)* = s'', *range(s)* = $\{\xi(s_k) \mid s_k \text{ is in } s''\}$ and *size(s)* = length(init(s)) + card(range(s)). Thus, range(s) is the set of all assignments of

atomic propositions which occur infinitely often in s. Note that init(s) is a finite sequence (and can be the null sequence!), final(s) is an infinite sequence, and range(s) is a subset of $2^{\mathcal{P}}$.

THEOREM 2.1: (Linear size model theorem for L(F)). If $f \in L(F)$ *is satisfiable then there exists a structure* $S = (s,\xi)$ *such that* size(s) $\leq 2 \cdot$ length (f) *and* $S, s_0 \models f$.

Proof of Theorem 2.1 is based on the following lemmas which provide insight on the expressive power of the F operator.

The following lemma shows that all states in final(s) with the same assignment of atomic propositions satisfy the same formulae in L(F).

LEMMA 2.2: *Let* $S = (s,\xi)$ *be a structure and let* s_j, s_k *be states in* final(s) *such that* $\xi(s_j) = \xi(s_k)$; *then for all* $f \in L(F)$, $S, s_j \models f$ *iff* $S,s_k \models f$.

Proof. The proof is by structural induction on f. If f is an atomic proposition, then the lemma holds trivially. Assume that the lemma holds for f_1, f_2. Then it is easily seen that the lemma hold for $f_1 \wedge f_2$, $\neg f_1$. We must prove that the lemma holds for $f = Ff_1$. Suppose $S, s_j \models Ff_1$. Then there is a state s_ℓ such that $\ell \geq j$ and $S,s_\ell \models f_1$. Since s_ℓ is in final(s), there are infinitely many m such that $\xi(s_m) = \xi(s_\ell)$ and (by induction) $S,s_m \models f_1$. Hence, there is an $m \geq k$ such that $S, s_m \models f_1$, that is $S, s_k \models f$. □

LEMMA 2.3: *Let* $S = (s,\xi)$, $T = (t,\pi)$ *be structures such that* length(init(s)) = length(init(t)); *for all* $j <$ length(init(s)); $\xi(s_j) = \pi(t_j)$; $\xi(s_0) = \pi(t_0)$ *(this is necessary for the case when length(init(s)) = 0) and* range(s) = range(t); *then for all* $f \in L(F)$, $S,s_0 \models f$ iff $T,t_0 \models f$. □

The above lemma can also be proved by induction on f, and it states that formulas in L(F) cannot distinguish the order of occurrence of states in final(s).

Let $s = (s_0,s_1,\dots)$, $t = (t_0,t_1,\dots)$ be finite or infinite sequences with all states in s,t being distinct. t *is a subsequence of* s *(written* $t \leq s$) iff there exist integers $i_0,i_1 \dots$ s.t. $i_0<i_1<i_2<\dots$ and for all $j \geq 0$, $s_{i_j} = t_j$. Let $S = (s, \xi)$ be a structure and t be a subsequence of s. We define init(t),final(t),range(t),size(t) are appropriately defined with respect to S. t is an *acceptable subsequence of* s (written t[s or s]t) if $t \leq s$, final(t) $\leq$ final(s) and if any s_j in final(s) is contained in t, then $\xi(s_j) \in$ range(t). We assume that the structure with respect to which '[' is defined, is understood from the context. If t [s then size(t) $\leq$size(s). Note that if $t \leq s$ then it is possible that size(t) >size(s).

LEMMA 2.4: *Let* $S = (s,\xi)$ *be a structure and let* t[s *be such that for all* $j \geq 0$, $S,t_j \models f$ *where* $f \in L(F)$. *Then* (a) *there exists an infinite sequence* u *such that* u[s, size(u) < c·length(f) *for some constant* c *and* (b) *for all structures* $W = (w,\varphi)$, *where* u [w [s, and φ *is the restriction of* ξ *to the states in* w, *the following condition holds:*

For any i *if* w_i *is present in* t *then* W, $w_i \models f$.

Proof:

Using D'morgan's laws and the identities $\neg Ff = G\neg f$, $\neg Gf = F\neg f$, any $f' \in L(F)$ can be converted to an equivalent formula f in which all negations apply to atomic propositions only. For formulas of this kind we prove Lemma 2.4 with c = 1. The proof is by induction on the length of the formula.

Basis: f = P or ¬P. u = null sequence satisfies the lemma.

Induction:

(i) $f = f_1 \wedge f_2$. For all $t_j \in t$, $S,t_j \models f_2$ $S,t_j \models f_2$. By induction hypothesis there exist u_1,u_2 such that size(u_1) < length(f_1), size(u_2) < length(f_2), and (b) holds for u_1,f_1 and u_2,f_2. Let $u \lceil s$ be the sequence containing the states of u_1 and u_2. Then size(u) $\leq$ size(u_1)+size(u_2)<length(f), and it is easily seen that (b) holds for u,f.

(ii) $f=f_1 \vee f_2$. The argument is similar as in (i).

(iii) $f=Ff_1$.

Case 1: t is finite. Let t_n be the last state of t. $S,t_n \models Ff_1$. Hence there is a s_j appearing after t_n in s such that $S,s_j \models f_1$. If s_j is in init(s), then let $t' = (s_j)$; *otherwise* let t' be the subsequence of all states s_k in final(s), such that $\xi(s_k) = \xi(s_j)$. For all s_k in t', $S,s_k \models f_1$. By induction hypothesis there is a $u' \lceil s$ such that size(u') < length(f_1) and (b) is satisfied for u', f_1 and with $t=t'$. Now let $u \lceil s$ be the sequence containing all states of u' and t'. Then size(u) $\leq$ size(t')+size(u') = 1+ size(u')<length(f) and (b) holds.

Case 2: t is infinite. There exist infinitely many k such that $S,s_k \models Ff_1$. Let $t' = (t'_0,...) \lceil s$ be such that t' is infinite, for all $j \geq 0$ $\xi(t'_j) = \xi(t'_{j+1})$, and $S,t'_j \models f_1$. The remained of the argument is as in case 1.

(iv) $F = Gf_1$. If t_0 is in init(t) then let t' = suffix of s starting from t_0, otherwise let t' = final(s). Clearly for all $j \leq 0$, S, $t'_j \models f_1$. By induction hypothesis there is a $u' \lceil s$ such that size(u')<length(f_1) and (b) holds for u', f_1 and with $t=t'$. Since t' is a suffix of s, it is easily observed that (b) holds for u,f, and t.

Since any formula $f \in L(F)$ can be converted into an equivalent formula in which negations are applied to atomic propositions only, and whose length is no more than double the length of the original formula, we see that Lemma 2.4 is true with c=2.

Proof Sketch of Theorem 2.1. Assume f is satisfiable and let $V = (v,\varphi)$ be a structure such that $V,v_0 \models f$. Let t be the sequence as follows. If length(init(v))> 0 then $t = (v_0)$; otherwise t is the sequence containing all states v_i such that $\varphi(v_i) = \varphi(v_0)$. Clearly $t \sqsubset v$, and due to Lemma 2.2 for all $i \geq 0$, $V, t_i \models f$. Now applying Lemma 2.4 with $S = V$, we get an infinite sequence $u \sqsubset v$, such that size(u)< 2·length(f) and u satisfies the condition given in Lemma 2.4. Let $s \leq v$ be the sequence containing all the states of t and u. Then $s \sqsubset v$ and size(s) ≤ size(t)+size(u)≤2·length(f). Let $S = (s,\xi)$ where ξ is the restriction of φ to the states appearing in s. Then from Lemma 2.4, $S,s_0 \models f$. □

THEOREM 2.5: *The following problems are NP-complete for the linear time logic* L(F).

(i) *Determination of truth in a* R*-structure.*

(ii) *Satisfiability.*

Proof: (i) We will prove that determining truth in an R-structure is NP-hard by reducing 3-SAT to this problem. Let $g = C_1 \wedge C_2 \wedge ... \wedge C_m$ be a boolean formula in 3-CNF where $C_i = \ell_{i1} \vee \ell_{i2} \vee \ell_{i3}$ (for $1 \leq i \leq m$), $\ell_{ik} = x_j$ or $\neg x_j$ ($1 \leq k \leq 3$) for some j such that $1 \leq j \leq n$. $x_1, x_2, ..., x_n$ are the variables appearing in g. Let $T = (N,R,\eta)$ be the R-structure defined as follows:

$$\mathcal{P} = \{ C_i \mid 1 \leq i \leq m \}$$

T can be described by the graph shown below:

$$N = \{ \tilde{x}_i \mid 1 \leq i \leq n \} \cup \{ \tilde{x}'_i \mid 1 \leq i \leq n \} \cup \{ y_i \mid 0 \leq i \leq n \}$$

$$R = \{ (y_{i-1}, \tilde{x}_i), (y_{i-1}, \tilde{x}'_i), (\tilde{x}_i, y_i), (\tilde{x}'_i, y_i) \mid 1 \leq i \leq n \} \cup \{ (y_n, y_n) \}$$

$$\eta(\tilde{x}_i) = \{ C_j \mid x_i \text{ appears as a literal in } C_j, \text{ i.e., for some k } 1 \leq k \leq 3, \ell_{jk} = x_i \}$$

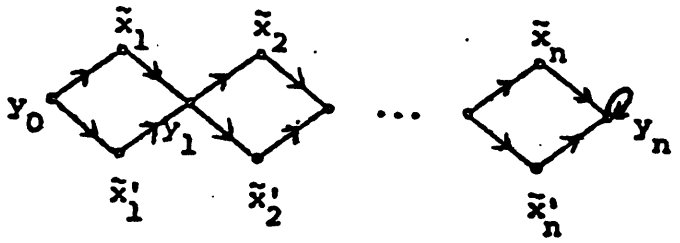


Figure 2-1:

$\eta(\tilde{x}'_i) = \{ C_j | \neg x_i$ appears as a literal in $C_j\}$

$\eta(y_j) = \varnothing$

It can easily be proved that g is satisfiable iff there exists a path p in T staring from y_0 such that $(S_p,s_0) \models F\ C_1 \wedge F\ C_2 \wedge \ldots \wedge F\ C_m$. The above reduction is a polynomial reduction. Hence determination of truth in a R-structure is NP-hard for the language L(F).

□

Let T = (N,R,η) be an R-structure. Any path p in T can be uniquely decomposed into p', p" such that p = p'·p", any state that appears in p" appears in it infinitely often, and p" is the maximal such suffix. All the states in p" belong to a strongly connected component in the graph of T. Using Lemma 2.4 it can be shown if there is a path q in T staring from q_0 such that $(S_q,s_0) \models f$, then there is a path p in T starting from q_0 such that $S_p,s_0 \models f$, p = p'·p", and length(p')$\leq$2·length(f)·card(N). A nondeterministic TM M guesses p' and the set C of states appearing in p". Next, it verifies that p' is a finite path starting from q_0 in T, that the subgraph containing nodes of C is strongly connected, and that there is an edge

from the last state of p' to a state in C. Then M uses the following algorithm to verify if $(S_{p'},s_0) \models f$. M labels each node x in p' or C with sub-formulae of f as follows:

For each node x label(x)← ∅;

For each formula g∈SF(f) in the increasing order of length(g) **do**

For each node x in p' or in C **do**

Case of g

$g=P$:	If $P \in \eta(x)$ then label(x)←label(x)∈ {P};
$g=\neg g_1$:	If $g_1 \notin$ label(x) then label(x)←label(x)∪{g}';
$g=Fg_1$:	If $g_1 \in$ label(y) for some y∈C then label(x)←label(x)∪{g}; If x is in p' and there is a state y in p' after x such that $g_1 \in$(y) then label(x)← label(x) ∪ {g};
$g=g_1 \wedge g_2$:	if $g_1, g_2 \in$ label(x) then label(x)←label(x)∪ {g};

End Case

End For

End For;

Accept iff f∈label(q_0).

It can easily be shown that the above algorithm works correctly and that it is polynomial time bounded in card(N) + length(f). Thus, determination of truth in a R-structure is NP-complete.

(ii) Satisfiability is NP-hard because boolean satisfiability is NP-hard.

Left fϵL(F) and $\mathfrak{P}$ = the set of atomic propositions appearing in f. From Theorem 2.1 if f is satisfiable, then it is satisfiable in structure S = (s,ξ) where size(s)$\leq$2·length(f). A nondeterministic TM M which checks for satisfiability of f operates as follows: M guesses init(s) and range(s) such that length(init(s))$\leq$2·length(f), card(range(s))$\leq$2·length(f). Next it uses a labelling algorithm similar to the one in (i) to accept or reject f. Clearly M is polynomial time bounded in length(f). □

We can also prove by the previous techniques a linear size model theorem (Theorem 2.1) for the logic $\tilde{L}$(F,X) and show that Theorem 2.5 holds for this logic as well.

2.4. The Complexity of L(F,X), L(U) AND L(U,S,X)

The main results of this section are summarized in the following theorem.

THEOREM 2.6: *The following problems are* PSPACE-complete *for the logics* L(F,X), L(U), *and* L(U,S,X):

(i) *Satisfiability,*

(ii) *Determination of truth in an R-structure.*

The proof of the above theorem is based on the following lemmas.

Let S = (s,ξ), T=(t,π) be structures such that for some m$\geq$0 the following conditions are satisfied:

$\forall i\ 0\leq i\leq m\ t_i=s_i$, $\forall i\ i>m+1\ t_i=s_{i-1}$

and π is an extension of ξ such that $\pi(t_{m+1})=\pi(t_m)$

i.e. T is obtained by duplicating the m^{th} state in S successively once. The following lemma is easily proved by induction on the formula f.

LEMMA 2.7: *For any* f∈L(U), $T,t_m \models f$ *iff* $T,t_{m+1} \models f$ *and for any* δ *in* s $S,\delta \models f$ iff $T,\delta \models f$. □

The above lemma states that by duplicating a state successively we do not change the truth value of a formula in L(U). Note that the lemma is not true for L(U,X).

LEMMA 2.8: *Determining truth in an R-structure is polynomial-time reducible to satisfiability for* L(F,X), L(U) *and* L(U,S,X).

Proof. Let $T = (N,R,\eta)$ be an R-structure and let f∈ L(U,S,X). Let $\mathcal{P}_1 = \{P_x | x \in N\}$ and $\mathcal{P}_1 \cap \mathcal{P} = \emptyset$. $\mathcal{P}_1$ contains one new atomic proposition for each state in N.

Let g_1 be the conjunction of all Q such that $Q \in \eta(x)$, g_2 be the disjunction of all Q such that $Q \in \mathcal{P} - \eta(x)$, g_3 be the disjunction of all P_y such that $(x,y) \in R$.

$$f_x = G(P_x \supset (g_1 \wedge (\neg g_2) \wedge Xg_3)).$$

Let h_1 be the disjunction of all P_y such that y∈N, h_2 be the conjunction of all f_x such that x∈N, let h_3 assert that exactly one proposition in $\mathcal{P}_1$ is true at any point. Then

$$f' = G(h_1) \wedge h_2 \wedge G(h_3)$$

Any structure $T = (t,\pi)$ such that $T,t_0 \models f'$ has the following property. At each state in t exactly one proposition in $\mathcal{P}_1$ is true, and if P_x is true at a state then all propositions in $\eta(x)$ are true in that state, all propositions in $(\mathcal{P} - \eta(x))$ are false in that state and in the next state P_y is true for exactly one y such that $(x,y) \in R$. Let $f'' = f' \wedge f \wedge P_q$. It can easily be seen that there is a path p in S starting from q such that $S_p,s_0 \models f$ iff f " is satisfiable. If f∈L(F,X) then f "∈ L(F,X).

In f', we can avoid the X operator as follows. We replace the formula Xg_3 by g' defined as follows. If $(x,x) \notin R$ then let $g' = (P_x U g_3)$,

otherwise let $g' = (G(P_x) \vee [P_x U(g_3 \wedge \neg P_x)])$.

If $(x,x) \notin R$ then g' causes P_x to repeat successively a finite number of times before P_y is true for some y which is a neighbor of x. However lemma 2.7 states that this does not change the truth value of the formula f', that is f" is true in a structure in which P_x does not repeat iff it is true in a structure in which P_x repeats a finite number of times. It is easily seen that there is a path p in S starting from q such that $S_p, s_0 \models f$ iff f" is satisfiable. The above reductions are clearly polynomial reductions. □

LEMMA 2.9: *Determination of truth in an R-structure is* PSPACE-*hard for* L(F,X) *and* L(U).

Proof. Let $M = (Q, \Sigma, \zeta, V_A, V_R, V_I)$ be a one tape deterministic TM where Q is the set of states, Σ is the alphabet, ζ: $Q \times \Sigma \rightarrow Q \times \Sigma \times \{L,R\}$, V_A, V_R, V_I are the accepting, rejecting and initial states respectively. Let M be S(n) space bounded such that S(n) is bounded by a polynomial in n. M halts on all inputs in state V_A or V_R, thus accepting or rejecting the input. An ID of M is appropriately defined. Let $a = a_1 a_2 \ldots a_n$ be an input to M.

Let $T = (N,R,\eta)$ be an R-structure shown in figure 2.2.

Let $\mathcal{P} = (Q \times \Sigma) \cup \Sigma \cup \{BI,EI\}$ be the set of atomic propositions. The structure in figure 2.2 has S(n) diamonds connected in a chain, and in each diamond there are card($Q \times \Sigma \cup \Sigma$) number of vertical vertices. In each diamond, on each vertical vertex exactly one atomic proposition is true, and every atomic proposition in $Q \times \Sigma \cup \Sigma$ is true on some vertical vertex of the diamond. Each subpath between BI and EI represents on ID of M, and a path from BI represents a sequence of Ids of M.

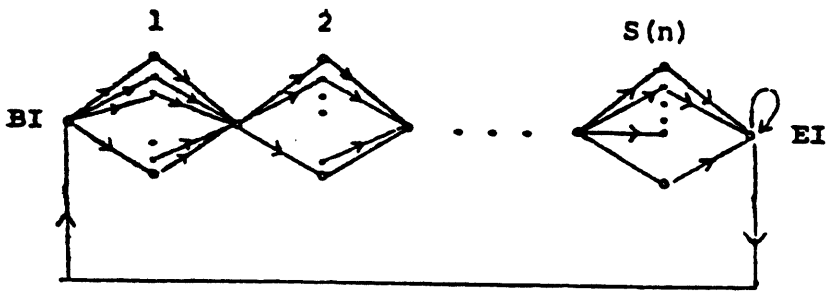


Figure 2-2:

Using S(n) X operators the relation between the contents of a tape cell in successive IDs can be asserted. Because of this, polynomial length bounded formulas in L(F,X) can be obtained asserting the following conditions: All the Ids on a path p starting from BI are valid, the first ID is the initial ID containing the input string $a_1a_2 \ldots a_n$, each successive ID follows from the previous one by one move of M, and the final ID appears on the path.

Let f_a be the conjunction of formulas asserting the above conditions. It is easily seen that there is a path p from BI in T such that $S_p, s_0 \models f_a$ iff M accepts a. For any input a,f_a can be obtained in polynomial time. By introducing additional propositions $P_0, P_1, \ldots, P_{S(n)}$ to mark the left and right end points of successive diamonds we can avoid the X operator using only the U operator. The resulting formula will be in L(U). □

Let $S=(s,\xi)$ be a structure and $f \in L(U,S,X)$. For any state s_i in s let $[s_i]_{S,f} = \{g \in SF(f) \mid S,s_i \models f\}$.

LEMMA 2.10: *In* $S=(s,\xi)$, *if* s_i,s_j *be two states such that* $[s_i]_{S,f} = [s_j]_{S,f}$ *then for the structure* $S' = (s',\xi')$ *where* $s' = (s_0,s_1, \ldots, s_{i-1},s_j,s_{j+1}, \ldots)$ *and* ξ' *is restriction of* ξ *to states in* s, *the following property holds:*

For all s_k *such that* s_k *is present in* s *and in* s', $[s_k]_{S,f} = [s_k]_{S',f}$ □

The above lemma can be proved by induction on the length of the formula f.

A formula g is said to be an U-*formula* if it is of the form g_1Ug_2. A structure $S=(s,\xi)$ is said to be *ultimately periodic* with starting index i and period m if $\forall k\geq i\ \xi(s_k) = \xi(s_{k+m})$. For a structure S and a formula f let $M_{S,f} = \{C\subseteq SF(f) \mid$ there are infinitely many k such that $[s_k]_{S,f} = C\}$.

LEMMA 2.11: *For the structure* $S = (s,\xi)$ *let* i, p *be integers such that* $[s_i]_{S,f} = [s_{i+p}]_{S,f}$ *and for any* $g = g_1Ug_2$, *if* $g\in[s_i]_{S,f}$ *then* $\exists m$ *such that* $i\leq m<i+p$ *and* $S,s_m\models g_2$(i.e. *every* U-formula *in* $[s_i]_{S,f}$ *is* **fulfilled** *before* s_{i+p}). *Let* $S' = (s',\xi')$ *be an ultimately periodic structure with starting index* i *and period* p *such that* $\forall k<i+p\ \xi(s_k) = \xi'(s'_k)$.

Then, $\forall k<i+p\ [s_k]_{S,f} = [s'_k]_{S,f}$ *and* $\forall k\geq i\ [s'_k]_{S',f} = [s'_{k+p}]_{S',f}$.

Proof. By induction we prove that for any $g\in SF(f)$,

(a) $\forall k<i+p\ S,s_k\models g$ iff $S',s'_k\models g$, and

(b) $\forall k\geq i\ S', s'_k\models g$ iff $S',s'_{k+p}\models g$.

Basis: If g is atomic then (a), (b) follow trivially.

Induction: Assume (a),(b) hold for $g_1,g_2\in SF(f)$. By a simple argument it can easily be shown that (a), (b) hold for $g = \neg g_1$, $g_1\wedge g_2$. Below we prove that (a), (b) hold for $g = g_1Ug_2, g_1Sg_2$; a similar argument can be given for $g = Xg_1$.

Case 1: $g = g_1 U g_2$.

We prove (a). (b) can be proved similarly. Assume for some $k<i+p$ $S,s_k \models g$. Assume $k<i$. From the hypothesis of the lemma it follows that for some ℓ such that $k \leq \ell < i+p$ $S,s_k \models g_2$ and $\forall j\ k<j\leq \ell$ $S,s_j \models g_1$. By the induction hypothesis the above holds for S' also. Hence $S',s'_k \models g$. Now assume $i \leq k < i+p$. The interesting case occurs when $\forall j\ k \leq j < i+p$ $S,s_j \models \neg g_2$, $S,s_j \models g_1$. In this case $S,s_{i+p} \models g$ and hence $S,s_i \models g$. From the hypothesis of the lemma and the induction hypothesis for (b) it can easily be seen that $S',s'_k \models g$. The implication in the other direction can also be proved similarly.

Case 2: $g = g_1 S g_2$.

Then for $k < i+p$ $S,s_k \models g$

iff $(\exists\ \ell \leq k$ $S,s_\ell \models g_2$ and $\forall j\ \ell < j \leq k$ $S,s_j \models g_1)$

iff $\exists \ell \leq k$ $S',s'_\ell \models g_2$ and $\forall j\ \ell < j \leq k$ $S',s'_j \models g_1)$

(due to induction hypothesis)

iff $S',s'_k \models g$.

We would like to prove that (b) also holds for g. Assume for $k \geq i$, $S',s'_k \models g$. Then there exists $\ell \leq k$ such that $S',s'_\ell \models g_2$ and for all j such that $\ell < j \leq k$ $S',s'_j \models g_1$. For $k \geq i+p$ or ($k<i+p$ and $\ell \geq i$), the result can easily be seen. So we consider the case when $\ell < i \leq k < i+p$.

In this case due to the induction hypothesis for (a), it can be seen that $S, s_\ell \models g_2$ and for all j such that $\ell < j \leq i$ $S,s_j \models g_1$. Hence $S,s_i \models g$. Due to the hypothesis of the lemma we see that $S,s_{i+p} \models g$. Thus, one of the following two cases holds:

(i) $\exists m(k<m<i+p$ and $S,s_m \models g_2$ and $\forall j$ such that $m<j\leq i+p$ $S,s_j \models g_1)$.

By the induction hypothesis for (a), the above condition is also satisfied by S'. Due to the induction hypothesis for (b) it follows that for all j such that $i+p \leq j \leq k+p$ $S',s'_j \models g_1$. Hence $S',s'_{k+p} \models g$;

(ii) $\forall j\ i \le j \le i+p\ S,s_j \models g_1$.

Due to the induction hypothesis for (a) the above condition holds for S' also. Due the the induction hypothesis for (b)

$\forall j\ k \le j \le k+p\ S',s'_j \models g_1$.

Hence $S',s'_{k+p} \models g$. The induction step for the reverse implication in (b) can be similarly proved. □

THEOREM 2.12: (*Ultimately periodic model theorem*). *A formula* $f \in L(U,S,X)$ *is satisfiable iff it is satisfiable in an ultimately periodic structure* $S = (s,\xi)$ *with starting index* $\ell \le 2^{1+length(f)}$, *period* $p \le 4^{1+length(f)}$ *and* $\forall k \le i\ [s_k]_{S,f} = [s_{k+p}]_{S,f}$.

Proof. Let f be satisfiable. Since f may not be satisfiable at the beginning of a structure, we consider $g = Ff$. Then there exists a structure $T=(t,\eta)$ such that $T,t_0 \models g$. Let ℓ, m be integers such that $[t_\ell]_{T,g} = [t_{\ell+m}]_{T,g}$ and

(*) $\{ [t_k]_{T,g} \mid \ell \le k < \ell+m \} = M_{T,g}$.

It is easily seen that each U-formula in $[t_\ell]_{T,g}$ is fulfilled before $t_{\ell+m}$. We apply reductions of Lemma 2.10 repeatedly to states between t_0 and t_ℓ, or to states between t_ℓ and $t_{\ell+m}$ (excluding $t_0, t_\ell, t_{\ell+m}$)without violating (*), until no more such reductions are possible. In the resulting sequence

(a) there are at most $2^{length(g)}$ states before t_ℓ and

(b) there are at most $(card(M_{T,g}))^2$ states between t_ℓ and $t_{\ell+m}$.

(a) follows trivially if we observe that , in the resulting sequence there are no two states before t_ℓ which satisfy exactly the same sub-formulae of g. If (b) does not hold, then there exist at least card $(M_{T,g})$ + 1 states between t_ℓ and $t_{\ell+m}$ which satisfy the same sub-formulae of g, i.e., there exist at least $card(M_{T,g})$ intervals between these states. It is easily

seen that there exist at least on interval among these, such that for every state within this interval there exist another state outside this interval and between t_ℓ and $t_{\ell+m}$, such that both these states satisfy the same sub-formulae of g. Hence we could have carried a reduction of Lemma 2.10 for the two end states of this interval without violating (*). This contradicts our assumption.

Let t' be the resulting sequence after the reductions and T' = (t', η ') be the structure, where η is the restriction of η to the states in t'. There exist integers $i \leq 2^{length(g)}$, $p \leq 4^{length(g)}$ such that t'_i, t'_{i+p} satisfy the same sub-formulae of g, and using lemma 2.11 we obtain a periodic structure S with starting index $i \leq 2^{length(g)}$, period $p \leq 4^{length(g)}$ such that $\forall k \geq i\ [s_k]_{S,g} = [s_{k+p}]_{S,g}$ and $S,s_0 \models g$.

Proof of Theorem 2.6. Let f be a formula in L(U,S,X) and g = Ff (= True Uf). f is satisfiable iff g is satisfiable at the beginning of an ultimately periodic structure. We describe below a nondeterministic TM M which checks for satisfiability of g. M guesses two numbers $n_1 \leq^{length(g)}$, $n_2 \leq 4^{length(g)}$ which are supposed to be the starting index and period of an ultimately periodic structure. Next, M guesses the sub-formulae that are true at the beginning, verifies that g is in this set. At this point it checks for boolean consistency and it checks that any sub-formula f_1 S f_2 is in this set iff f_2 is in this set.

Subsequently, M guesses the sub-formulae that are true in the next state and verifies their consistency with the sub-formulae that are guessed to be true in the next state. If $Sub_{present}$, Sub_{next} are the formulae guessed to be true at the present state and the next state respectively it verifies that

$Xf_1 \epsilon Sub_{present}$ iff $f_1 \epsilon Sub_{next}$;

$f_1 U f_2 \epsilon Sub_{present}$ iff $f_2 \epsilon Sub_{present}$ or ($f_1 \epsilon$ $Sub_{present}$ and $f_1 U f_2 \epsilon$ Sub_{next});

$f_1 S f_2 \in Sub_{next}$ iff $f_2 \in Sub_{present}$ or ($f_1 \in Sub_{next}$ and $f_1 S f_2 \in Sub_{present}$).

It also checks the boolean consistency whenever it guesses a set of sub-formulae to be true at any state , i.e.

$f_1 \wedge f_2 \in Sub_{present}$ iff $f_1, f_2 \in Sub_{present}$.

$\neg f_1 \in Sub_{present}$ iff $f_1 \notin Sub_{present}$.

It continues the above process each time incrementing the counter. When the counter is n_1, it notes that it is in the periodic part of the structure. It saves the set of sub-formulae Sub_{period} guessed to be true at the beginning of the period, and it re-initializes the counter. It continues guessing the sub-formulae in the next state and incrementing the counter. At each instance it has to keep three sets of sub-formulae: those that are true in present state, those true in the next state and those true at the beginning of the period. When the counter has value n_2, it stops guessing and takes Sub_{period} to be the set of sub-formulae true in the next state. At each step in the above procedure it checks the consistency of the sub-formulae guessed. It also verifies the following condition. Each formula of the form $(f_1 U f_2) \in Sub_{period}$ is eventually fulfilled with in the period, that is f_2 is present in the set of sub-formulae guessed to be true somewhere within the period. It can easily be proved by induction that M accepts an input formulae iff it is satisfiable. Clearly M uses space linear in length(f). Using Savitch's[Sa70] theorem it follows that there is a polynomial space bounded deterministic TM that decides satisfiability.

2.5. Complexity of Extensions of the Logic

In [Wo81] propositional linear temporal logic is enriched with the addition of operators corresponding to regular right linear grammars. Let R be a regular right linear grammar with terminal symbols $a_1,a_2,\ldots,a_n$ and non-terminal symbols $N_1,N_2,\ldots,N_m$. If $f_1,f_2,\ldots,f_n$ are formulae in the logic then so is $N_j(f_1,f_2,\ldots,f_n)$ for $1\leq j\leq m$. For a structure $S=(s,\xi)$, $S,s_k \models N_j(f_1,f_2,\ldots,f_n)$ iff there exists a string $a_{i_1},a_{i_2},a_{i_3},\ldots$ generated by R from N_j such that for all $\ell\geq 0$ $S,s_{\ell+k}\models f_{i_{\ell+1}}$.

Ex: Consider the grammar $N_0 \rightarrow a_1a_2N_0$. It generates the infinite string $a_1a_2a_1a_2\ldots$. $S,s_0\models N_0(\text{True},P)$ iff P holds at all even states in s.

For convenience, we assume that each production rule in the grammar has at most one terminal symbol. Note that for any grammar we can obtain an equivalent grammar with the above property by increasing the size of the grammar by at most a constant factor. For any formula f in this logic we define SF(f) as follows:

If $f = P$ then $SF(f) = P$;

If $f = f_1 \wedge f_2$ or $f_1 U f_2$ or $f_1 S f_2$ then $SF(f) = SF(f_1) \cup SF(f_2) \cup \{f\}$;

If $f = \neg f_1$ or Xf_1 then $SF(f) = SF(f_1) \cup \{f\}$;

If $f = N_j(f_1,f_2,\ldots f_n)$ where N_j is a non-terminal in the above regular grammar,

then $SF(f) = SF(f_1) \cup SF(f_2) \ldots \cup SF(f_n) \cup \{N_j(f_1,f_2,\ldots,f_n) \mid 1 \leq j \leq m\}$

With the above definition of SF(f), it can easily be seen that lemma 2.10 holds for this logic. We can easily show that theorem 2.12 holds by some changes in the proof.

In Theorem 2.6, we assume that the grammars corresponding to the regular operators

are encoded as part of the input. In this case if the input length is n, then care(SF(f))$\leq n^2$ where f is the input formula. To prove Theorem 2.6, we need to modify the previous proof as follows. The two guessed integers n_1,n_2 should be less than or equal to 2^{n^2}, 4^{n^2} respectively. In addition the operation of M is to be modified as follows:

At any time $N_j(f_1,f_2,\ldots,f_n)$ is in the set of sub-formulae guessed to be true at any state, iff either there is production rule of the form $N_j \rightarrow a_k N_\ell$ is in the grammar, such that f_k is also present in the set of formulae guessed to be true in the present state and $N_\ell(f_1,f_2,\ldots,f_n)$ is present in the set of formulae guessed to be true in the next state, or there is production rule of the form $N_j \rightarrow a_k$ so that f_k is present in the set of sub-formulae guessed to be true in the present state.

For each formula of the form $\neg N_j(f_1,f_2,\ldots,f_n)$ present in the set of sub-formulae guessed to be true at the beginning of the periodic part, M keeps a set of sub-formulae denoted by $\varphi(N_j(f_1,f_2,\ldots,f_n))$. These are the sub-formulae that are to be false in the next state. At the beginning of the periodic part this set contains only N_j. If $\varphi_{present}, \varphi_{next}$ denote the value of φ in the present and next state, then φ is updated as follows: $\varphi_{next}(N_j(f_1,f_2,\ldots f_n)) = \{N_\ell |$ there is a production rule $N_p \rightarrow a_k N_\ell$ in the grammar such that $N_p \in \varphi_{present}(N_j(f_1,f_2,\ldots,f_n))$ and f_k is present in the set of sub-formulae guessed to be true in the present state$\}$. M makes sure that $\varphi(N_j(f_1,f_2,\ldots,f_n))$ becomes empty at some point within the periodic part of the structure. This will guarantee that $N_j(f_1,f_2,\ldots,f_n)$ is false at the beginning of the period.

It can easily be proved M accepts an input formula in the extended logic iff the formula is satisfiable. It is easily seen that M is polynomial space bounded.

2.6. Conclusion

In this chapter we have examined the complexity of satisfiability and truth in a particular structure for various propositional linear temporal logics. We have determined that these problems are NP-complete for L(F) and PSPACE-complete for L(F,X), L(U), L(U,S,X), and Wolper's extended logic (see Figure 2.3). Satisfiability for L(U,X) can also be shown to be in PSPACE by translation into SDPDL [HR81]; however this technique does not work for L(U,S,X) or for Wolper's logic with regular operators. It should be also observed that both the X and G operators are necessary for PSPACE-hardness of satisfiability of L(F,X). Thus, the logic $\tilde{L}$(F,X) is NP-complete since it does not permit the G operator.

Finally, it is interesting to compare our results with the corresponding results for branching-time logics. Since branching-time formulae are interpreted over the states of a structure rather than over executions sequences, determining truth in a particular structure is much easier and, in many cases, is in P [CE81]. Satisfiability, on the other hand, can be shown to be exponential-time hard for branching time logics with a nexttime operator and is shown to be PSPACE-complete in [La77] for many branching time logics with F and G operators. Thus satisfiability for the branching time logics is apparently harder than for the corresponding linear time logics.

Logic	Satisfiability	Validity	Truth in an R-Structure
L(F) $\tilde{L}$(F,X)	NP-complete	CO-NP-complete	NP-complete
L(F,X) L(U) L(U,X) L(U,S,X) Linear time logic with Regular operators	PSPACE-complete	PSPACE-complete	PSPACE-complete

Figure 2-3:

Chapter 3

Temporal Logic with Propositional Quantifiers

3.1. Introduction

In the previous chapter we examined the complexities of propositional linear temporal logics. It can be shown that the language L(U,S,X) can not express many interesting properties of parallel programs. It is shown in [Wo81] that we can not express the property that some event should occur at every even state in a sequence. For this reason we extend this logic by introducing quantifiers over propositions. Specifically we consider the language using the temporal operators F, X; which allows quantifiers over propositions. We call this logic QPTL. It can easily be shown that QPTL is as expressive as the monadic second order language of one successor when interpreted over natural numbers. Let Σ_k be the set of formulae of QPTL in *standard form* (i.e. all quantifiers appear in the beginning), and having a quantifier prefix that begins with an existential quantifier and has k-1 alternation of quantifiers. Let $\tilde{\Sigma}_k$ be the set of sentences in Σ_k which are true in all interpretations in which all propositions are false after certain point and all the quantifiers range over such propositions. Let g(k,n) be a function defined as follows. g(k,n) has a stack of k exponents.

$$g(0,n) = n$$

$$g(k+1, n) = 2^{g(k,n)}.$$

g(k,n) has a stack of k exponents.

Let g_k-SPACE = { L | For some polynomial p(n), L is accepted by a deterministic Turing m/c that uses at most g(k,p(n)) of space on each input of length n}.

In this chapter we prove that $\tilde{\Sigma}_{k+1}$ is complete in g_k-SPACE with respect to log-SPACE reductions. We also prove that the set of true sentences of QPTL in Σ_2, is EXSPACE-complete. In section 3.2 we present the upper bounds, while in section 3.3 we prove the lower bounds.

3.2. Upper bounds

The formulae of QPTL are built from the atomic propositions, boolean connectives, the temporal operators F,X and the quantifiers symbol $\exists$. We assume that the atomic propositions are drawn from the set $\mathcal{P}$. A well formed formula in QPTL is either an atomic proposition or is of the form $\neg f_1$, $f_1 \wedge f_2$, $X(f_1)$, $F(f_1)$, $\exists P(f_1)$ where f_1, f_2 are well formed formulae and P is an atomic proposition. The set of free atomic propositions in a formula is defined inductively in the obvious way. A formula without any free propositions is called a sentence. A formula (sentence) is said to be in standard form if all the propositional quantifiers appear at the beginning of the formula. A formula (or sentence) is said to be in Σ_k (Π_k) form if it is in normal form and has a quantifier prefix which starts with an existential(universal) quantifier and has (k-1) alternations of quantifiers.

An interpretation is a pair (t,i) where $i \geq 0$, and t is an ω-sequence of states, each state being a mapping from the set of atomic propositions into {True, False}. The truth of a formula f in an interpretation (t,i) (denoted by $t,i \models f$) is inductively defined.

$(t,i) \models \exists P(f_1)$ iff for some interpretation (t',i) such that such that $\forall j \geq 0$, t'_j assigns the same truth value as t_j for all the atomic propositions excepting P and $(t',i) \models f_1$.

The inductive definition for the other cases is same as give in the previous chapter.

Let f be formula in QPTL not containing any propositional quantifiers. Let

SF(f) = {g | g is a subformula of f or the negation of a subformula of f}. A $c \subseteq SF(f)$ is said to be consistent and complete iff it satisfies the following conditions:

(i) For each g, $\neg g \in SF(f)$ exactly one of them is in c;

(ii) For each $g = g_1 \wedge g_2 \in c$ iff $g_1 \in c$ and $g_2 \in c$.

Let S'_f be the set of consistent and complete subsets of SF(f). Let Tableau(f) = (S'_f, R'_f) be a directed graph such that $(c_1, c_2) \in R'_f$ iff the following condition is fulfilled: For any g,

(i) $g = Xg' \in c_1$ iff $g' \in c_2$,

(ii) $g = Fg' \in c_1$ iff $g' \in c_1$ or $g \in c_2$.

LEMMA 3.1: *In tableau* (f) *if for any* c

(i) $g = Fg' \in c$ *then for all* d *such that there is a path from* d *to* c, $g \in d$;

(ii) *If* $g = \neg Fg' \in c$ *then for all* d *such that*

there is a path from c *to* d, $g \in d$.

Proof: The above lemma is easily proved by induction. □

We say that a formula is an F-formula iff it is of the form F(g). The following Lemma easily follows from the previous one.

LEMMA 3.2: *In tableau*(f) *all the states in a strongly connected component contain the same* F-formulae. □

A finite state automaton A on infinite strings is a 5-tuple (Σ, S, M, st, H) where Σ is a finite alphabet, S is a finite set of states, $M: S \times \Sigma \rightarrow 2^S$, st is the start state and $H \subseteq 2^S$. A run of A on an input $a = (a_0, a_1, \ldots) \in \Sigma^\omega$, is an infinite sequence $s \in S^\omega$ such that $s_0 = st$ and $\forall i \geq 0 \ \ s_{i+1} \in M(s_i, a_i)$. For $s \in S^\omega$, let $in(s) = \{c \in S \mid \forall i \geq 0 \ \exists j$ such that $j \geq i \ \ s_j = c\}$. i.e.

in(s) is the set of all states that appear infinitely often in s. The automaton A is said to accept input a iff there exists a run s of A on a such that in(s)$\in$H. L(A) denotes the set of strings accepted by A.

Let f be a quantifier free formula in QPTL and $\mathcal{P}_f = \{P_1, P_2, \ldots, P_m\}$ be the set of free propositions appearing in f. Also let $\Sigma_f = \{0,1\}^m$. For any $\sigma = (\sigma_0, \sigma_1, \ldots) \in \Sigma_f^\omega$, let $\tilde{\sigma}$ denote the ω-sequence $(\tilde{\sigma}_0, \tilde{\sigma}_1, \ldots)$ where:

$\forall i \geq 0\ \tilde{\sigma}_i : \mathcal{P}_f \rightarrow \{\text{True,False}\}$ such that

$\forall j\ 1 \leq j \leq m,\ \tilde{\sigma}_i(P_j) = \text{True iff } (\sigma_i)_j = 1.$

Let $A(f) = (\Sigma_f, S_f, M_f, c_{st}, H_f)$ be a FSA on infinite strings defined as follows:

$S_f = \{c_{st}\} \cup S'_f$ where S'_f is the set of states in tableau(f);

For $c \neq c_{st}$

$M_f(c,\zeta) = \{c' \mid (c,c') \in R'_f \text{ and } \forall i\ 1 \leq i \leq m,\ P_i \in c' \text{ iff } \zeta_i = 1\}$

$M_f(c_{st},\zeta) = \{c' \mid f \in c' \text{ and } \forall i\ 1 \leq i \leq m, P_i \in c' \text{ iff } \zeta_i = 1\}$

$H_f = \{D \subseteq S'_f \mid D \text{ is non-empty and for each F-formula } g = Fg',$

such that for some d, $g \in d$, $d \in D$, there exists a d' with $g' \in d' \in D\}$.

THEOREM 3.3: $\sigma \in L(A(f))$ *iff* $\tilde{\sigma}, 0 \models f$.

Proof: ($\Rightarrow$) Assume $\sigma \in L(A(f))$. Then there exists a run $s = (s_0, \ldots)$ of A (f) on σ such that the set of states that appear infinitely often in s, is in H_f. Since s is a run, $s_0 = c_{st}$ and $s_{i+1} \in M_f(s_i, \sigma_i)$.

Claim 1: *For all* $g \in SF(f)$ *and for all* $i \geq 0$ $g \in s_{i+1}$ *iff* $\tilde{\sigma}, i \models g$.

Proof of Claim 1: By induction on the structure of g.

Basis: For $g = P$ where $P \in \mathcal{P}_f$: the result follows from the way we define M_f.

Induction:

(i) $g = \neg g_1$. $g_1 \in s_{i+1}$ iff $\tilde{\sigma}, i \models g_1$ for all $i \geq 0$.

Since either g or $\neg g$, is in s_{i+1} it follows that $g \in s_{i+1}$ iff $\tilde{\sigma}, i \models g$ for all $i \geq 0$.

(ii) $g = g_1 \wedge g_2$. obvious.

(iii) $g = Xg_1$. It can easily be seen that $g \in s_{i+1}$ iff $g_1 \in s_{i+2}$

iff $\tilde{\sigma}, i+1 \models g_1$ iff $\tilde{\sigma}, i \models g$.

(iv) $g = Fg_1$. Assume $g \in s_{i+1}$. From the way we defined acceptance it is

easily seen that for some $j \geq i+1$ $g_1 \in s_j$. Hence $\tilde{\sigma}, j\text{-}1 \models g_1$ and so $\tilde{\sigma}, i \models g$.

Assume $\tilde{\sigma}, i \models g$. Then for some $j \geq i$, $\tilde{\sigma}, j \models g_1$. Hence

$g_1 \in s_{j+1}$. By lemma 1, it follows that $g \in s_{i+1}$.

□ Q·E·D for claim 1.

continuation of proof of Theorem 3.3.

From the definition, $f \in s_1$. Hence by claim 1, $\tilde{\sigma}, 0 \models f$.

($\Leftarrow$) Assume $\tilde{\sigma}, 0 \models f$.

Let $s = (s_0, s_1, \ldots)$ be a ω-sequence of states of A (f) defined as follows: $s_0 = c_{st}$, $s_{i+1} = \{g \in SF(f) \mid \sigma, i \models g\}$. Each s_i (for $i \geq 0$) is consistent and complete, and hence is in S_f. It can easily be seen that $(s_i, s_{i+1}) \in M_f(s_i, \sigma_i)$ for $i \geq 0$. Also if $Fg_1 \in s_i$ (for $i \geq 1$), then $\exists j \geq i$, such that $g_1 \in s_j$. Due to this, the set of all states that appear infinitely often in s, is in H_f. Thus s is an accepting run of A (f) on σ. □

We use the following additional notation. Let $\mathcal{P}_f = \{P_1, P_2, \ldots, P_m\}$ be the set of free propositions in f. Then $\bar{0} \in \Sigma_f$ is such that $\forall i\ 1 \leq i \leq m\ (\bar{0})_i = 0$, and θ: $\mathcal{P}_f \rightarrow$ {True, False} such that $\forall_i\ 1 \leq i \leq m\ \theta(P_i) =$ False. $c_0 = \{g \in SF(f) \mid \theta^\omega, 0 \models g\}$.

An interpretation (t,i) is *weak* if t is of the form ($t \cdot \theta^{\omega}$), i.e. in t all propositions are false after certain instance. Let WQPTL be the logic obtained from QPTL by restricting all the interpretations to be weak and all quantifiers range over propositions which are false after certain instance. The truth of a formula is defined inductively as for QPTL but by restricting all the interpretations to be weak.

Let $FL(f) = \{\sigma \in \Sigma_f^* \mid \sigma',0 \models f \text{ where } \sigma' = \tilde{\sigma} \cdot \theta^{\omega}\}$.

It is to be observed that if $\sigma \in FL(f)$ then $\forall n \geq 0\ \sigma \cdot (\bar{0})^n \in FL(f)$, and f is true in all weak interpretations iff $FL(f) = \Sigma_f^*$. We state the following obvious lemma.

LEMMA 3.4: $FL(\neg f) = (\Sigma_f^* - FL(f))$. □

We inductively define an automation $\tilde{A}(f) = (\Sigma_f, S_f, M_f, c_{st}, H_f)$ on finite strings which accepts FL(f) where $H_f \subseteq S_f$. Observe that A(f) is an automation on infinite strings while $\tilde{A}$ (f) is on finite strings.

For a quantifier free formula f, we define, $\tilde{A}(f)$ as follows:

$\tilde{A}(f) = (\Sigma_f, S_f, M_f, c_{st}, H_f)$ where Σ_f, S_f, M_f and c_{st} are same as in A(f)

$H_f = \{ c \mid c \in S_f \text{ and } c_0 \in M_f(c, \bar{0})\}$.

THEOREM. 3.5: $\sigma \in \Sigma_f^*$ is accepted by $\tilde{A}(f)$ iff $\sigma \in FL(f)$.

Proof: The proof easily follows from theorem 3.3 and the fact that $\{c_0\} \in H_f$ in A(f). □

We extend M_f in $\tilde{A}(f)$ to the domain Σ_f^* in the natural way, i.e. for any $\sigma \in \Sigma_f^*$ $M_f(c,\sigma)$ is the set of all states reached when $\tilde{A}(f)$ is run on the input σ starting from the state c. Let f be any formula and $g = \exists P_m f$.

Define $\tilde{A}(g) = (\Sigma_g, S_g, M_g, c_{st}, H_g)$ where

where $\Sigma_g = \{0,1\}^{m-1}$, $S_g = S_f$,

$M_g(c,\delta) = \{c' \mid c' \in M_f(c,(\delta_1,\delta_2,\ldots,\delta_{m-1},i))$ for some $i \in \{0,1\}\}$,

$H_g = H_f \cup \{c \mid$ for some $n \geq 1$ $M_g(c,(\bar{0}^n)) \cap H_f \neq \emptyset\}$.

LEMMA 3.6: $FL(g) = L(\tilde{A}(g))$

Proof:

($\supseteq$): It is easily seen that $FL(g) \supseteq L(\tilde{A}(g))$.

($\subseteq$): Assume $\sigma \in FL(g)$. Then $\tilde{\sigma} \cdot \theta^{\omega},0 \models g$.

$\exists\, \gamma \in (\Sigma_f^*)$ such that length(γ) $\geq$ length(σ) and

$\forall i\ 0 \leq i < \text{length}(\sigma)$, $\forall j\ 1 \leq j \leq m\text{-}1$ $(\gamma_i)_j = (\sigma_i)_j$, and

$\forall_i$ length(σ) $\leq i \leq$ length(γ), $\forall j\ 1 \leq j \leq m\text{-}1$ $(\gamma_i)_j = 0$, and $\tilde{\gamma} \cdot \theta^{\omega},0 \models f$.

Hence $\gamma \in L(\tilde{A}(f))$. From the above it can be seen that for some $n \geq 0$,

$M_g(c_{st}, \sigma \cdot (\bar{0})^n) \cap H_f \neq \emptyset$.

Hence $M_g(c_{st},\sigma) \cap H_g \neq \emptyset$ and $\sigma \in L(A(g))$ □

If $g = \neg f$, then $\tilde{A}(g)$ is the automation that accepts the complement of the language accepted by A(f) and can be obtained in the standard way.

Let $\tilde{\Sigma}_k = \{f \mid f$ is a sentence in Σ_k and is true in all weak interpretations$\}$.

THEOREM. 3.7: $\tilde{\Sigma}_1 \in$ PSPACE *and for* $k \geq 1$ $\tilde{\Sigma}_{k+1} \in g_k$-SPACE.

Proof: Let $f = \exists_1 \forall_2 \exists_3 \ldots Q_{k+1} f_1$ where $\exists_i$ or $\forall_i$ is a sequence of existential or universal quantifiers respectively. By replacing each $\forall$ by $\neg\exists\neg$, f can be written as

$\exists_1 \neg \exists_2 \neg \exists_3 \neg \exists_4 \ldots \neg \exists_{k+1}(f_2)$. Let $f_3 = \exists_2 \neg \exists_3 \ldots \neg \exists_{k+1}(f_2)$. Thus $f = \exists_1 \neg (f_3)$.

From $\tilde{A}(f_2)$, by applying the transformation of lemma 3.6, and by using successive complementation for each negation, we can obtain $\tilde{A}(f_3)$. Since f_3 has only (k-1) negations between the quantifiers it is easily seen that $\tilde{A}(f_3)$ has at most g(k,n) states where n is the length of f. It is easily seen that $f \in \tilde{\Sigma}_{k+1}$ iff $L(\tilde{A}(f_3)) \neq (\Sigma_{f_3}^*)$. The later condition can be easily checked using space polynomial in the size of $\tilde{A}(f_3)$.

Hence $\Sigma_{k+1} \in g_k$-SPACE. □

THEOREM 3.8: *The set of true sentences of QPTL in* Σ_2, *is in* EXSPACE.

Proof: Let $f = \exists P_1 P_2 \ldots P_k \forall P_{k+1} \ldots P_{k+\ell} f_1(P_1, \ldots, P_{k+\ell})$

$= \exists P_1 P_2 \ldots P_k \neg \exists P_{k+1} \ldots P_{k+\ell}(g)$

where $g = \neg f_1(P_1 \ldots, P_{k+\ell})$.

Let $h = \exists P_{k+1} \ldots P_{k+\ell}(g)$ and

$A(g) = (\Sigma_g, S_g, M_g, c_{st}, H_g)$ where $\Sigma_g = \{0,1\}^{k+\ell}$. From A(g), we obtain

$A(h) = (\Sigma_h, S_h, M_h, C_{st}, H_h)$ where $\Sigma_h = \{0,1\}^k$, $S_h = S_g$, $H_h = H_g$ and

$M_h(c,\delta) = \{c' \mid \exists\, \delta' \in \{0,1\}^{k+\ell}$ such that for $1 \leq i \leq k$ $\delta'_i = \delta_i$, and $c' \in M_g(c,\delta')\}$.

Let $r, r' \in (S_h)^\omega$ be runs of A(h) on an input σ. We say that r is an accepting run iff in $(r) \in H_h$. Assume that in $(r') \subseteq$ in(r). In this case the following claim holds.

Claim 1: If r is not an accepting run then r' is also not an accepting run.

Proof: States in in(r) belong to a strongly connected component in tableau(g), and so each state in in(r) contains the same F-formulae. Hence if in(r') $\in H_g$ then in(r) $\in H_g$. □

A run r is maximal if there is no other run r' on σ such that in(r) $\subseteq$ in(r'). From the above claim to check that σ is not accepted by A (h) it is enough if we verify that all maximal runs are not accepting runs.

It can easily be shown that if A(h) does not accept at least one input, then there exists an input of the form $\sigma = \alpha \cdot (\beta)^{\omega}$ not accepted by A(h). Let σ be as above and $\beta = (\beta_0, \ldots, \beta_{m-1})$.

We define a directed graph G which captures all the runs of A(h) on β^{ω}, starting from different states. The nodes of G are of the form (c,i) where $c \in S_h$ and $0 \leq i < m$. There is an edge from (c,i) to (c', (i+1) **mod** m) iff $c' \in M_h(c,\beta_i)$. These are the only edges in the graph. For each infinite path $p = [(c_0,0), (c_1,1), \ldots]$ let $\pi(p) = (c_0,c_1,\ldots)$. $\pi(p)$ is a maximal run iff all the nodes of a strongly connected component of G appear infinitely often in p. Let $after(\alpha) = M_h(c_{st},\alpha)$. If we know after ($\alpha$), and the states of strongly connected components in G we can easily get all the set of states that appear infinitely often in the maximal runs. To determine the above it is not necessary to build the graph G as we show below.

Let $G' = (V,E,\ell)$ be a labelled directed graph where $V = S_h$, $(c,d) \in E$ iff there is a path from (c,0) to (d,0) in G, of the form $[(c,0), (c_2,1), \ldots, (c_{m-1}, m-1), (d,0)]$. $\ell((c,d)) = \{b \mid$ for some j, (b,j) is on a path of the above form from (c,0) to (d,0) in G$\}$.

Let C be a strongly connected component in G' and $\varphi(C) = \{c \mid$ for some $(c_1,c_2) \in E$, $c \in \ell((C_1,C_2))\}$. C is said to be *fulfilled* if for every F-formula g= Fg' such that $g \in c \in \varphi(C)$, there is a c' such that $g' \in c' \in \varphi(C)$. The following claim is easily proved from our previous remarks.

Claim 2: σ is not accepted by A(h) iff every strongly connected component in G' that is reachable from a state in after(α), is not fulfilled. □

Now we can easily give a non-deterministic $O(2^{c.length(f)})$ space bounded algorithm that checks that some $\sigma = \alpha \cdot (\beta)^{\omega}$ is not accepted by A(h). The algorithm successively guesses each letter in α and builds after (α). At some point it guesses that β starts. From the beginning of β it builds G' while simultaneously guessing β as follows. Let $G'_{i+1} = (V,E_{i+1},\ell_{i+1})$ be the partially built G' after β_i is guessed. Initially, $E_0 = \{(c,c) | c \in V\}$, $\ell_0((c,c)) = \{c\}$. After β_i is guessed G'_{i+1} is obtained from G'_i using the following equations.

$E_{i+1} = \{(c',c'') |$ $\exists$d such that $(c',d) \in E_i$ and $c'' \in M_h(d,\beta_i)\}$,

If $(c',c'') \in E_{i+1}$ then

$\ell_{i+1}((c',c'')) = \{e |$ For some d, $(c',d) \in E_i$, $c'' \in M_h(d,\beta_i)$ and $e \in \ell_i((c',d))\} \cup \{c''\}$. It is easily seen that to build G'_{i+1} it requires at most $O(2^{c.length(f)})$ space. At some point after guessing β_m it guesses that β ends and it takes G'_m as G' and it verifies that the conditions given in claim 2 is satisfied. The above procedure accepts f iff for some σ, σ is not accepted by A(h). Clearly f is true iff there is at least one such string. □

3.3. Lower bounds

In this section we show the lower bound for $\tilde{\Sigma}_k$.

THEOREM. 3.9: *Every language in* g_k-SPACE *is* log-SPACE *reducible to* $\tilde{\Sigma}_{k+1}$.

Proof: We assume that there is a procedure which given an m >0, uses space log m and outputs a formula $\varphi_{k,m}(P_x,P_y) \in \Sigma_k$ such that if $t,0 \models \varphi_{k,m}(P_x,P_y)$ then

(i) P_x is true at exactly one point in t and so is P_y and

(ii) If $t,i \models P_x$, $t,j \models P_y$ then $j = i + N_{k,m}$ where $N_{k,m} \geq g(k,m)$.

i.e. the places where P_x, P_y are true are separated by a distance of length at least $N_{k,m}$. We show later how this formula can be obtained.

Let $M = (Q,\Sigma,S,V_A,V_I)$ be a one tape deterministic turing m/c that is g(k,p(n)) space bounded. The elements in M are the set of states, the alphabet, the next move function, the accepting state and the initial state respectively. Let ID_0, ID_1, ... be a sequence of IDs that describe the computation of M on some input of length n. Let m = p(n). Without loss of generality we assume that each ID is of length $N_{k,m}$. If $N_{k,m} \geq g(k,m)$ then M uses only the initial g(k,m) cells in each ID. Using a formula f in WQPTL we express the computation of M on input 'a'.

We use the propositions P_σ for each $\sigma \in (Q \times \Sigma) \cup \Sigma$, where the elements in $Q \times \Sigma$ are the composite symbols. We use a proposition B which marks the beginning of IDs. The sequence between successive instances where B holds defines an ID. We briefly describe how we can obtain a formula that expresses the computation of M.

Let f be the conjunction of the following formulae which are informally described.

$f_1 = \forall P_x,P_y [\varphi_{k,m}(P_x,P_y) \supset f_1']$ where f_1' is quantifier free and asserts that if B is true at some point after x, then between x and y (excluding y) there is exactly one place where B is true. This condition implies that all the IDs are of length $N_{k,m}$;

f_2 asserts that each ID is a valid ID. In this the only difficult part will be to assert that each ID has exactly one compound symbol. After some thought it is easily seen that we can obtain a formula like f_1 which asserts this;

f_3 asserts that each successive ID is obtained from the previous one by one move of M

i.e. the contents of a cell in an ID depend on the contents of this cell and it's neighbor's contents in the preceding ID.

$f_3 = \forall P_x, P_y [(\varphi_{k,m}(P_x, P_y) \wedge f_3') \supset f_3'']$.

f_3', f_3'' are quantifier free. f_3' asserts that there is at least one instance before and after y where B is true i.e. x,y are points with in the computation of M. f_3'' asserts that the contents in the cell at y is related to the contents in the cells at x and its neighbors.

f_4 asserts that ID_0 is an initial ID, and that eventually a final ID appears.

If f_1, f_2, f_3 are converted into standard form then they will be in Π_k, this is because $\varphi_{k,m}$ appears only in the antecedent of an implication. Let f' be f converted into standard form. Then $f' \in \Pi_k$. Let g be the sentence obtained by introducing an existential quantification over each free variable in f'. Clearly $g \in \Sigma_{k+1}$ and is a true sentence iff "a" is accepted by M. Also g can be obtained using space O(log n). □

Let $c_0, c_1, \ldots, c_{\ell-1}$ be a sequence of binary counters each of size p. We let $v(c_i)$ denote the integer value of the counter c_i, and $c_{i,j}$ denote the j^{th} bit in the counter c_i (*0th* bit is the least significant bit). We say that the above sequence is a proper sequence of counters iff $v(c_0) = 0$, $v(c_{i+1}) = v(c_i) + 1$ for $0 \leq i < \ell-1$, and $v(c_{\ell-1}) = 2^p-1$. Clearly for a proper sequence $\ell = 2^p$. It is easily seen that the bit values in successive counters are related as follows:

For $0 \leq i < \ell-1$, $0 \leq j < p$

(A) $c_{(i+1),j} = c_{i,j}$ iff $\exists\, r < j$ such that $c_{i,r} = 0$.

We recursively define $\varphi_{k+1,m}$ in terms of $\varphi_{k,m}$ for $k \geq 1$. For this we assert that there is a proper sequence of counters each of length $N_{k,m}$ between the points x,y. We use the

proposition B whose truth values give the contents of the counters and the proposition M which marks the beginnings of counter. We require the following conditions to be satisfied.

1. P_x holds at exactly one point, say x, and P_y holds at exactly one point say y. M is true at x and y, and is false at all points before x and after y. We can easily give a quantifier free formula f_1 that asserts this condition.

2. The successive points where M is true, are separated by a distance $N_{k,m}$. We can satisfy this by requiring that for all points i,j, if i and j are separated by a distance $N_{k,m}$ then there is exactly one instance between i and j where M is true. It is easily seen that we can assert this condition by the following formula.

$f_2 = \forall P_i \forall P_j (\varphi_{k,m}(P_i,P_j) \supset f_2')$

where f_2' is quantifier free and asserts that if i,j are between x and y, then there exists exactly one instance p such that $i \leq p < j$ and M is true at p.

By this condition we can consider the truth values of B, between successive instances where M is true, to be a binary counter of size $N_{k,m}$. We consider the right most bit in a counter to be the least significant bit.

3. The proposition B is false throughout the subsequence between the 1st and 2nd instances where M is true. This asserts that the value of the first counter is 0. This is expressed by

$$f_3 = \forall P_i(f_3' \supset G(P_i \supset \neg B))$$

f_3' is quantifier free and asserts that P_i is true at exactly one instance, and this instance is in the first ID. Such an f_3' can easily be obtained.

4. The value of each succeeding counter is equal to the value of the preceding counter

incremented by 1. This can be expressed by corresponding the values of bits which are in the same position in successive counters. Such positions are those between x and y, which are separated by a distance $N_{k,m}$. The following formula expresses the above condition.

$f_4 = \forall P_i \forall P_j ((\varphi_{k,m}(P_i,P_j) \wedge f_4') \supset f_4''))$ where f_4' and f_4'' are quantifier free. f_4' asserts that i,j are between x and y. f_4'' asserts that if there is a less significant bit position than i, in the counter of i and B is false in this position, then the truth values of B at i and j are equal, otherwise they are different. We can easily obtain f_4' and f_4''.

5. In the last counter, that is the one before y B is true at all points in the counter. We can easily obtain a quantifier free formula, f_5 that expresses this condition.

6. In all counters other than the last counter there is at least one position where B is false. This condition guarantees that 5 does not hold in any other counter. The following formula expresses this condition.

$f_6 = \forall P_i \forall P_j ((\varphi_{k,m}(P_i,P_j) \wedge f_6') \supset f_6'')$

where f_6', f_6'' are quantifier free. f_6' asserts that i,j are the beginnings of successive counters and j is strictly before y. f_6'' asserts that there exists a point between i and j where B is false.

Let $f' = \bigwedge_{1 \le i \le 6} f_i$, and f be the resulting formula when f' is converted into standard form. Since $\varphi_{k,m} \in \Sigma_k$, it is easily seen that $f \in \Pi_k$.

Let $\varphi_{k+1,m}(P_x,P_y) = \exists B \exists M (f)$. Clearly $\varphi_{k+1,m} \in \Sigma_{k+1}$

We describe how to obtain $\varphi_{1,m}$. We use m propositions $Q_0, Q_1, \ldots, Q_{m-1}$ which are existentially quantified. The truth values of these propositions define a binary counter at any point. We assert that the sequence of these counters starting from x, form a proper

sequence of counters. We also assert that there is exactly one point between x and y, which is just before y where all the propositions are true. It is not difficult to see how this formula can be obtained. We can obtain $\varphi_{1,m}$ such that it's length is $O(m^2)$. Clearly $N_{1,m} = 2^m$.

THEOREM. 3.10: $t,0 \models \varphi_{k+1,m}(P_x,P_y)$ iff P_x is true at exactly one point , say x; P_y is true exactly one point say y; and $y = x + N_{k,m}$ where $N_{k,m}$ is given as follows:

$N_{1,m} = 2^m$,

$N_{k,m} = N_{k-1,m} \cdot 2^{(N_{k-1,m})}$ for $k \geq 2$ □

The above theorem can easily be proved by induction. It is easily seen that length of $\varphi_{k,m}$ is $O(m^2)$, and that $\varphi_{k,m}$ can be obtained in space (log m) recursively. It is clear that $N_{k,m} \geq g(k,m)$.

THEOREM. 3.11: *The set of true sentences of* QPTL in Σ_2 is EXSPACE-complete.

Proof: Follows from theorems 3.8 and 3.9 . □

Chapter 4

Automatic Verification of finite state concurrent programs: A Practical approach

4.1. Introduction

In the traditional approach to concurrent program verification, the proof that a program meets its specifications is constructed by hand using various axioms and inference rules in a deductive system such as temporal logic ([MP81], [HO80], [OL80]). The task of proof construction is in general quite tedious, and a good deal of ingenuity may be required to organize the proof in a manageable fashion. Mechanical theorem provers have failed to be of much help due to the inherent complexity of even the simplest logics.

We argue that proof construction is unnecessary in the case of finite state concurrent systems and can be replaced by a model theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic. The global state graph of the concurrent system can be viewed as a finite Kripke structure, and an efficient algorithm can be given to determine whether a given structure is a model of a particular formula - i.e. to determine if the program meets its specification. The algorithm, which we call a *model checker*, is similar to the global flow analysis algorithms used in compiler optimization and has complexity linear in both the size of the structure and the size of the specification. When the number of global states is not excessive (i.e. not more

than a few thousand) we believe that our technique may provide a useful new approach to the verification of finite state concurrent systems.

Our approach is of wide applicability since a large class of concurrent programming problems have finite state solutions, and the interesting properties of many such problems can be specified in propositional temporal logic. For example, many network communication protocols (e.g. the Alternating Bit Protocol [BSW69]) can be modeled at some level of abstraction by a finite state system. A typical requirement for such systems is that every transmitted message must ultimately be received; this can easily be expressed in the logic we use.

Our specification language is a propositional, branching-time temporal logic called *Computation Tree Logic* (CTL) and is based on the logical systems described in [EC80], [BMP81], and [CE81]. Since our goal is to specify concurrent systems we must be able to assert that a correctness property only holds on fair execution sequences. It follows from the results of ([EC80], [EH83]) that CTL cannot express such a property. The alternative of using a linear time logic is ruled out because any model checker for such a logic must have high complexity ([SC82]). We overcome this problem by moving fairness requirements into the semantics of CTL. Specifically, we change the definition of our basic modalities so that only fair paths are considered. Our previous model checking algorithm is modified to handle this extended logic without changing its complexity.

This chapter is organized as follows: Section 4.2 contains the syntax and semantics of our logic. In section 4.3 we describe the basic model checking algorithm and illustrate its use to establish absence of starvation for a solution to the mutual exclusion problem. An extension of the model checking algorithm which only considers *fair computations* is given

in section 4.4. Section 4.5 describes an experimental implementation of the extended model checking algorithm and shows how it can be used to verify the correctness of the Alternating Bit Protocol. In section 4.6 we consider extensions of our logic that are more expressive and investigate the complexity of model checkers for these logics. The chapter concludes with a discussion of related work and remaining open problems.

4.2. The Specification Language.

The syntax for CTL is given below. AP is the underlying set of *atomic propositions.*

1. Every atomic proposition $p \in AP$ is a CTL formula.
2. If f_1 and f_2 are CTL formulae, then so are $\neg f_1$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A[f_1 \, U \, f_2]$, and $E[f_1 \, U \, f_2]$.

The symbols $\wedge$ and $\neg$ have their usual meanings. X is the *nexttime* operator; the formulae AXf_1 (EXf_1) intuitively means that f_1 holds in every (in some) immediate successor of the current program state. U is the *until* operator; the formula $A[f_1 U f_2]$ ($E[f_1 U f_2]$) intuitively means that for every computation path (for some computation path), there exists an initial prefix of the path such that f_2 holds at the last state of the prefix and f_1 holds at all other states along the prefix.

We define the semantics of CTL formulae with respect to a labeled state-transition graph. Formally, a CTL structure is a triple $M = (S, R, P)$ where

1. S is a finite set of states.
2. R is a binary relation on S ($R \subseteq S \times S$) which gives the possible transitions between states and must be total, i.e. $\forall x \in S \exists y \in S \, [(x,y) \in R]$.
3. P is an assignment of atomic propositions to states i.e. $P : S \rightarrow 2^{AP}$.

A *path* is an infinite sequence of states $(s_0, s_1, s_2, ...)$ such that $\forall i \, [(s_i, s_{i+1}) \in R]$. For

any structure M = (S,R,P) and state $s_0 \in S$, there is an *infinite computation tree* with root labeled s_0 such that $s \rightarrow t$ is an arc in the tree iff $(s,t) \in R$.

We use the standard notation to indicate truth in a structure: $M, s_0 \models f$ means that formula f holds at state s_0 in structure M. When the structure M is understood, we simply write $s_0 \models f$. The relation $\models$ is defined inductively as follows:

$s_0 \models p$ iff $p \in P(s_0)$.

$s_0 \models \neg f$ iff not$(s_0 \models f)$.

$s_0 \models f_1 \wedge f_2$ iff $s_0 \models f_1$ and $s_0 \models f_2$.

$s_0 \models AXf_1$ iff for all states t such that $(s_0,t) \in R$, $t \models f_1$.

$s_0 \models EXf_1$ iff for some state t such that $(s_0,t) \in R$, $t \models f_1$.

$s_0 \models A[f_1 \, U \, f_2]$ iff for all paths $(s_0, s_1,...)$,
$\exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_j \models f_1]]$.

$s_0 \models E[f_1 \, U \, f_2]$ iff for some path $(s_0, s_1,...)$,
$\exists i[i \geq 0 \wedge s_i \models f_2 \wedge \forall j[0 \leq j < i \rightarrow s_i \models f_1]]$.

4.3. Model Checker

Assume that we wish to determine whether formula f is true in the finite structure M = (S, R, P). We design our algorithm so that when it finishes, each state will be labelled with the set of subformulae true in the state. We let label(s) denote this set for state s. Consequently, $M, s \models f$ iff $f \in$ label(s) at termination. In order to explain our algorithm we first consider the case in which each state is currently labelled with the **immediate** subformulae of f which are true in that state.

We will use the following primitives for manipulating formulas and accessing the labels associated with states:

- arg1(f) and arg2(f) give the first and second arguments of a two argument formula f such as $A[f_1 \, U \, f_2]$.
- labelled (s, f) will return true (false) if state s is (is not) labelled with formula f.
- add_label(s, f) adds formula f to the current label of state s.

Our state labelling algorithm (**procedure** label_graph (f)) must be able to handle seven cases depending on whether f is atomic or has one of the following forms: $\neg f_1$, $f_1 \wedge f_2$, AXf_1, EXf_1, $A[f_1 \, U \, f_2]$, or $E[f_1 \, U \, f_2]$. We will only consider the case in which $f = A[f_1 \, U \, f_2]$ here since all of the other cases are either straightforward or similar. For the case $f = A[f_1 \, U \, f_2]$ our algorithm uses a depth first search to explore the state graph. The bit array marked[1: nstates] is used to indicate which states have been visited by the search algorithm. The algorithm also uses a stack ST to keep track of those states which require additional processing before the truth or falsify of f can be determined. The boolean procedure stacked(s) will determine (in constant true) whether state s is currently on the stack ST.

```
begin
    ST := empty_stack;
    for all s ∈ S do marked(s) := false;
    L : for all s ∈ S do
    if ¬ marked(s) then au(f,s,b)
end
```

The recursive procedure au(f,s,b) performs the search for formula f starting from state s. When au terminates, the boolean result parameter b will be set to true iff $s \models f$. The annotated code for procedure au is shown below:

```
procedure au(f,s,b)
begin
```

{If s is marked and stacked, return false (see lemma 4.1). If s is already labelled with f, then return true. Otherwise, if s is marked but neither stacked nor labelled, then return false.}

```
if marked(s) then
    begin
        if stacked(s) then
            begin
                b := false;
                return
            end ;
        if labelled(s,f) then
            begin
                b := true;
                return
            end;
        b := false;
        return
    end;
```

{Mark state s as visited. Let f = A[f_1 U f_2]. If f_2 is true at s, f is true at s; so label s with f and return true. If f_1 is not true at s, then f is not true at s; so return false. }

```
marked(s) := true;
if labelled(s,arg2(f)) then
    begin
        add_label(s,f);
        b := true;
        return
    end
else if ¬labelled(s,arg1(f)) then
    begin
        b := false;
        return
    end;
```

{Push s on stack ST. Check to see if f is true at all successor states of s. If there is some successor state s1 at which f is false, then f is false at s also; hence remove s from the stack and return false. If f is true for all successor states, then f is true at s; so remove s from the stack, label s with f, and return true.}

```
    push(s,ST);
    for all s1 ∈ successors(s) do
        begin
            au (f,s1,b1);
            if ¬b1 then
                begin
                    pop(ST);
                    b := false;
                    return
                end
        end;
    pop(ST);
    add_label(s,f);
    b := true;
    return
end of procedure au.
```

To establish the correctness of the algorithm we must show that

$$\forall s \left[\text{labelled}(s,f) \leftrightarrow s \models f\right]$$

holds on termination. Without loss of generality we consider only the case in which f has the form $A[f_1 \, U \, f_2]$. We further assume that the states are already correctly labelled with the subformulae f_1 and f_2. The first step in the proof is an induction on depth of recursion for the procedure au. Let I be the conjunction of the following eight assertions:

I1. All states are correctly labelled with the subformulae f_1 and f_2: $\forall s\left[\text{labelled}(s,f_i) \leftrightarrow s \models f_i\right]$ for i = 1,2.

I2. The states on the stack form a path in the state graph: $\forall i \left[1 \leq i < \text{length}(ST) \rightarrow (ST(i), ST(i+1)) \in R\right]$.

I3. The current state parameter of au is a descendant of the state on top of the stack: $(\text{Top}(ST), s) \in R$.

I4. f1 $\wedge$ $\neg$ f2 holds at each state on the stack : $\forall i \left[1 \leq i \leq \text{length}(ST) \rightarrow ST(i) \models f1 \wedge \neg f2\right]$.

I5. Every state on the stack is marked but unlabelled : $\forall i \left[1 \leq i \leq \text{length}(ST) \rightarrow \text{marked}(ST(i)) \wedge \neg \text{labelled}(ST(i), f) \right]$.

I6. If a state is labelled with f, then it also marked and f is true in that state: $\forall s \left[\text{labelled}(s,f) \rightarrow \text{marked}(s) \wedge s \models f \right]$.

I7. If a state is marked but neither labelled with f nor on the stack, then f must be false in that state: $\forall s \left[\text{marked}(s) \wedge \neg \text{labelled}(s,f) \wedge \neg \exists i \left[1 \leq i \leq \text{length}(ST) \wedge s = ST[i] \right] \rightarrow s \models \neg f \right]$.

I8. ST_0 records the contents of the stack: $ST = ST_0$.

We claim that if I holds before execution of au(f, s, b), then I will also hold on termination of au; Moreover, the boolean result parameter b will be true iff f holds in state s. In the standard Hoare triple notation for partial correctness assertions the inductive hypothesis would be

$$\{I\} \text{ au } (f, s, b) \{I \wedge (b \leftrightarrow s \models f)\}.$$

Once the inductive hypothesis is proved, the correctness of our algorithm is easily established. If the stack is empty before the call on au, we can deduce that both of the following conditions must hold:

a. $\forall s \left[\text{marked}(s) \rightarrow [\text{labelled}(s, f) \rightarrow s \models f] \right]$ (from I1).

b. $\forall s \left[\text{marked}(s) \rightarrow [\neg \text{labelled}(s,f) \rightarrow s \models \neg f] \right]$ (from I2, I3).

It follows that

$$\forall s \left[\text{marked}(s) \rightarrow [\text{labelled}(s, f) \leftrightarrow s \models f] \right].$$

Because of the **for** loop L in the calling program for au, every state will eventually be marked. Thus, when loop L terminates $\forall s \left[\text{labelled}(s, f) \leftrightarrow s \models f \right]$ must hold.

Proof of the inductive hypothesis is straightforward but tedious and will be left to the reader. The only tricky case occurs when the state s is marked and on the stack. In this situation the procedure au simply sets b to false and returns. To see that this is the correct action, we make use of the following observation:

LEMMA 4.1: *Suppose there exists a path* $(s_1, s_2, ..., s_m, s_k)$ *in the state graph such that* $1 \leq k \leq m$ *and* $\forall i[\, 1 \leq i \leq m \rightarrow s_i \models \neg f_2]$, *then* $s_k \models \neg A[f_1 \, U \, f_2]$. □

Assuming that the states of the graph are already correctly labelled with f_1, and f_2, it is easy to see that the above algorithm requires time O(card(S) + card(R)). The time spent by one call of procedure au excluding the time spent in recursive calls is a constant plus time proportional to the number edges leaving the state s. Thus, all calls to au together require time proportional to the number of states plus the number of vertices since au is called at most once in any state.

We next show how handle CTL formulas with arbitrary nesting of subformulas. Note that if we write formula f in prefix notation and count repetitions, then the number of subformulae of f is equal to the length of f. (The length of f is determined by counting the total number of operands and operators.) We can use this fact to number the subformulae of f. Assume that formula f is assigned the integer i. If f is unary i.e. f= (op f_1) then we assign the integers i+1 through i + length(f_1) to the subformulae of f_1. If f is binary i.e. f = (op f_1 f_2) then we assign the integers from i + 1 through i + length(f_1) to the subformulae of f_1 and i + length(f_1) through i + length(f_1) + length(f_2) to the subformulae of f_2. Thus, in one pass through f we can build two arrays nf[1 : length(f)] and sf[1 : length(f)] where nf[i] is the i^{th} subformula of f in the above numbering and sf[i] is the list of the numbers assigned to the immediate subformulae of the i^{th} formula. For example, if f = (AU (NOT X) (OR Y Z)), then nf and sf are given below:

```
nf[1] (AU (NOT X) (OR Y Z))     sf [1]   (2 4)
nf[2] (NOT X)                   sf [2]     (3)
nf[3] X                         sf [3]     nil
nf[4] (OR Y Z)                  sf [4]   (5 6)
nf[5] Y                         sf [5]     nil
nf[6] Z                         sf [6]     nil
```

Given the number of a formula f we can determine in constant time the operator of f and the number assigned to its arguments. We can also efficiently implement the procedures "labelled" and "add_label". We associate with each state s a bit array L[s] of size length(f). The procedure add_label(s,fi) sets L[s][fi] to true, and the procedure labelled(s,fi) simply returns the current value of L[s][fi].

In order to handle an arbitrary CTL formula f we successively apply the state labelling algorithm described at the beginning of this section to the subformulas of f, starting with simplest (i.e. highest numbered) and working backwards to f:

```
for fi := length(f) step -1 until 1 do
        label_graph (fi);
```

Since each pass through the loop takes time O(size(S) + card(R)), we conclude that the entire algorithm requires O(length(f) · (card(S) + card(R))).

THEOREM 4.2: *There is an algorithm for determining whether a CTL formula f is true in state s of the structure* M = (S, R, P) *which runs in time* O(length(f)·(card(S) + card(R))). □

We illustrate the model checking algorithm by considering a finite state solution to the *mutual exclusion problem* for two processes P_1 and P_2. In this solution each process is always in one of three regions of code:

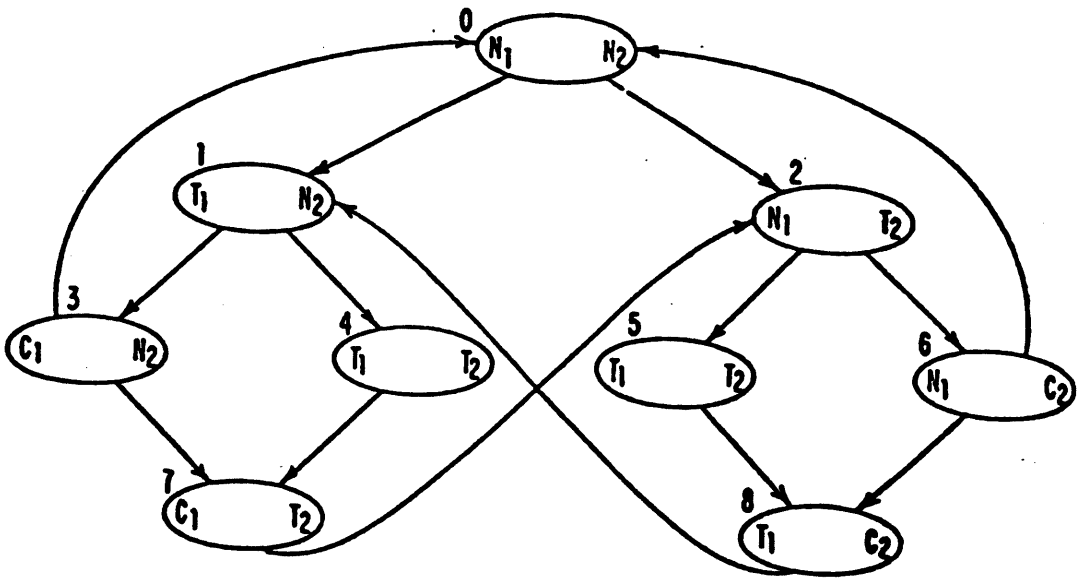


Figure 4-1: Global state transition graph for two process mutual exclusion problem

N_i	the Noncritical region,
T_i	the Trying region,
or C_i	the Critical region.

A *global* state transition graph for this solution is shown in figure 4.1 . Note that we only record transitions between different regions of code; moves entirely within the same region are not considered at this level of abstraction.

In order to establish *absence of starvation* for process 1 we consider the CTL formula $T_1 \rightarrow AFC_1$ or, equivalently, $\neg T_1 \vee AFC_1$, where $AFp \equiv A[\text{true } U\ p]$ means that p occurs at

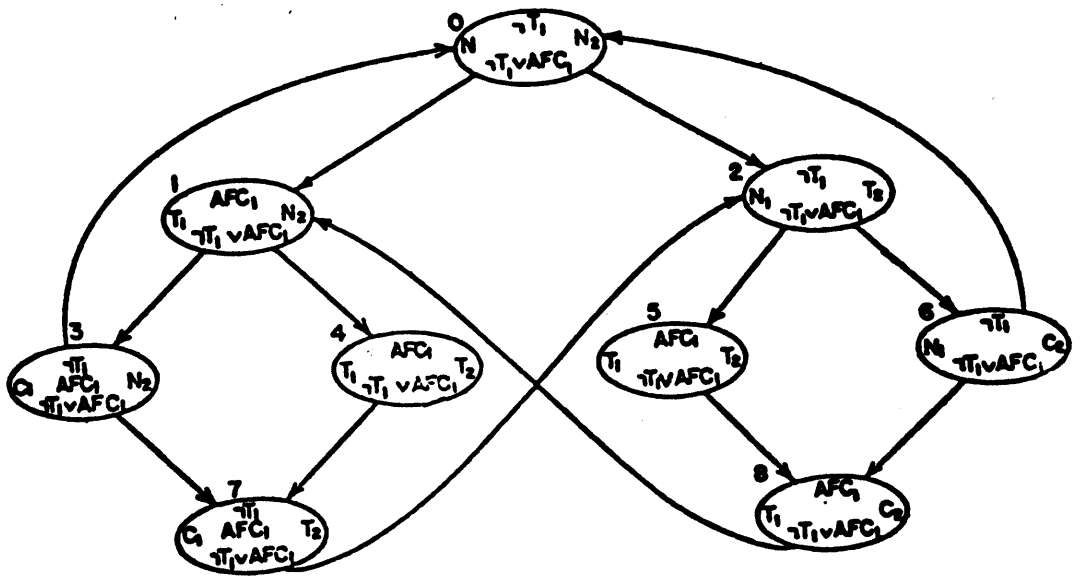


Figure 4-2: Global state transition graph after termination of model checking algorithm

some point on all execution paths. In this case the set of subformulae contains $\neg T_1 \vee AFC_1$, $\neg T_1$, T_1, AFC_1 and C_1. The states of the global transition graph will be labelled with these subformulae during execution of the model checking algorithm. On termination every state will be labelled with $\neg T_1 \vee AFC_1$ as shown in figure 4.2. Thus, we can conclude that $s_0 \models AG(T_1 \rightarrow AFC_1)$ where $AGp \equiv \neg E[true\ U\ \neg p]$ means that p holds globally on all computation paths. It follows that process 1 cannot be prevented from entering its critical region once it has entered its trying region.

4.4. Introducing Fairness into CTL

Frequently, in verifying concurrent systems we are only interested in the correctness of fair execution sequences. For example, with a system of concurrent processes we may wish to consider only those computation sequences in which each process is executed infinitely often. When dealing with network protocols where processes communicate over imperfect (or lossy) channels we may also wish to restrict the set of computation sequences; in this case the *unfair* execution sequences are those in which a sender process continuously transmits messages without any reaching the receiver. Since we are considering only finite state systems, each of these notions of fairness requires that some collection of states be repeated infinitely often in every fair computation. It follows from [EH83] that correctness of fair executions cannot be expressed in CTL. In fact, CTL cannot express the property that some proposition Q should eventually hold on all fair executions.

In order to handle fairness and still obtain an efficient model checking algorithm we modify the semantics of CTL. The new logic, which we call CTL^F, has the same syntax as CTL. But a structure is now a 4-tuple (S, R, P, F) where S, R, P have the same meaning as in the case of CTL, and F is a collection of subsets of S i.e. $F \subseteq 2^S$. A path p is fair iff the following condition holds:

for each $c \in F$, there are infinitely many instances on p at which some state in c appears.

CTL^F has exactly the same semantics as CTL except that all path quantifiers range over fair paths.

An execution of a system Pr of concurrent processes is some interleaving of the execution steps of the individual processes. We can model a system of concurrent processes by a structure (S, R, P) and labelling function $L:R \rightarrow Pr$. S is the set of global states of the system, R is the single step execution relation of the system, and for each transition in R, L

gives the process which caused the transition. By duplicating each state in S at most card(Pr) times, we can model the concurrent system by a structure (S*, R*, P*, F), where each state in S* is reached by the execution of at most one process, and F is a partitioning of S* such that each element in F is the set of states reached by the execution of one process; thus card(F) = card(Pr). The fair paths of the above structure are exactly the fair execution sequences of the system of concurrent processes. A similar approach can be used to model network protocols (see section 5).

We next extend our model checking algorithm to CTL^F. We introduce an additional proposition Q, which is true at a state iff there is a fair path starting from that state. This can easily be done, by obtaining the strongly connected components of the graph denoted by the structure. A strongly connected component is *fair* if it contains at least one state from each c_i in F. We label a state with Q iff there is a path from that state to some node of a fair strongly connected component. As usual we design the algorithm so that after it terminates each state will be labelled with the subformulae of f_0 true in that state.

We consider the two interesting cases where $f \in sub(f_0)$ and either $f = E[g\ U\ h]$ or $f = A[g\ U\ h]$. We assume that the states have already been labelled with the immediate subformulae of f by an earlier stage of the algorithm.

(i) $f = E[g\ U\ h]$: f is true in a state iff the CTL formula $E[g\ U\ (h \wedge Q)]$ is true in that state, and this can be determined using the CTL model checker. A state s is labeled with f iff f is true in that state.

(ii) f = A[g U h] : It is easy to see that A[g U h] = ¬ (E[¬h U (¬g ∧ ¬h)] ∨ EG(¬h)). For a state s we can easily check if s ⊨ E[¬h U (¬g ∧ ¬h)] using the previous technique. To check if s ⊨ EG(¬h) we use the following procedure. Let G_R be the graph corresponding to the above structure. From G_R eliminate all nodes v such that h ∈ label(v) and let G'_R be the resultant labeled graph. Find all the strongly connected connected components of G'_R and mark those which are fair. If s is in G'_R and there is a path from s to a fair strongly component of G'_R then s ⊨ EG(¬h); otherwise s ⊨ ¬ EG(¬h). As in (i), s is labeled with f iff f is true in s.

If n = max(card(S), card(R)), m = length(f) and p = card(F), then it can be shown that the above algorithm takes time $O(n \cdot m \cdot p)$.

4.5. Using the Extended Model Checker to Verify the Alternating Bit Protocol

In this section we consider a more complicated example to illustrate *fair paths* and to show how the Extended Model Checking (EMC) system might actually be used. The example that we have selected is the *Alternating Bit Protocol* (ABP) originally proposed in [BSW69]. This algorithm consists of two processes, a *Sender process* and a *Receiver process*, which alternately exchange messages. We will assume (as in [QS81]) that messages from the Sender to the Receiver are *data messages* and that messages from the Receiver to the Sender are *acknowledgments*. We will further assume that each message is encoded so that garbled messages can be detected. Lost messages will be detected by using time-outs and will be treated in exactly the same manner as garbled messages (i.e. as error messages).

Ensuring that each transmitted message is correctly received can be tricky. For example, the acknowledgment to a message may be lost. In this case the Sender has no

choice but to resend the original message. The Receiver must realize that the next data message it receives is a duplicate and should be discarded. Additional complications may arise if this message is also garbled or lost. These problems are handled in the algorithm of [BSW69] by including with each message a control bit called the *alternation bit.*

In the EMC system finite-state concurrent programs are specified in a restricted subset of the CSP programming language [Ho78] in which only boolean data types are permitted and all messages between processes must be *signals.* CSP programs for the Sender and Receiver processes in the ABP are shown in figures 4.3 and 4.4 . To simulate garbled or lost messages we systematically replace each message transmission statement by a (nondeterministic) alternative statement that can potentially send an error message instead of the original message. Thus, for example,

Receiver ! mess0 would be replaced by

[True → Receiver ! mess0
□
True → Receiver ! err]

A global state graph is generated from the state machines of the individual CSP processes by considering all possible ways in which the transitions of the individual processes may be interleaved. Since construction of the global state graph is proportional to the product of the sizes of the state machines for the individual processes, various (correctness preserving) heuristics are employed to reduce the number of states in the graph. Explicit construction of the global state machine can be avoided to save space by dynamically recomputing the successors of the current state. The global state graph for the ABP is shown in the figure 4.5 .

Once the global state graph has been constructed, the algorithm of section 4 can be used to determine if the program satisfies its specifications. In the case of the ABP we require that every data message that is generated by the Sender process is eventually accepted by the Receiver process:

$$AG[gen\_dm0 \rightarrow AX[A[\neg (gen\_dm0 \vee gen\_dm1) \, U \, acc\_dm0]] \wedge$$

$$AG[gen\_dm1 \rightarrow AX[A[\neg (gen\_dm0 \vee gen\_dm1) \, U \, acc\_dm1]]$$

This formula is not true of the global state graph shown in figure 4.5 because of infinite paths on which a message is lost or garbled each time that it is retransmitted. For this reason, we consider only those fair paths on which the initial state occurs infinitely often. With this restriction the algorithm of section 4 will correctly determine that the state graph of figure 4.5 satisfies its specification.

As of October 1982, most of the programs that comprise the EMC system have been implemented. The program which parses CSP programs and constructs the global state graph is written in a combination of C and lisp and is operational. An efficient top-down version of the model checking algorithm of section 3 has also been implemented and debugged. The extended model checking algorithm of section 4 (which only considers fair paths) has been implemented in Lisp and is currently being debugged.

```
*[ gen – dm0;
   RCV!dm0;
   *[ RCV ? am0 → exit;
        □
      RCV ? am1 → RCV ! dm0;
        □
      RCV ? err → RCV ! dm0;
   ]
   gen – dm1;
   RCV ! dm1;
   *[ RCV ? am1 → exit;
        □
      RCV ? am0 → RCV ! dm1;
        □
      RCV ? err → RCV ! dm1;
   ]
]
```

Figure 4-3: Sender Process(SND)

```
*[ *[ SND ? dm0 → exit;
        □
      SND ? dm1 → SND ! am1;
        □
      SND ? err → SND ! am1;
   ]
   acc – dm0;
   SND ! am0;
   *[ SND ? dm1 → exit;
        □
      SND ? dm0 → SND ! am0;
      SND ? err → SND ! am0;
   ]
   acc – dm1;
   SND ! am1;
]
```

Figure 4-4: Receiver Process(RCV)

(Note: dm stands for data message; am stands for acknowledgement message.)

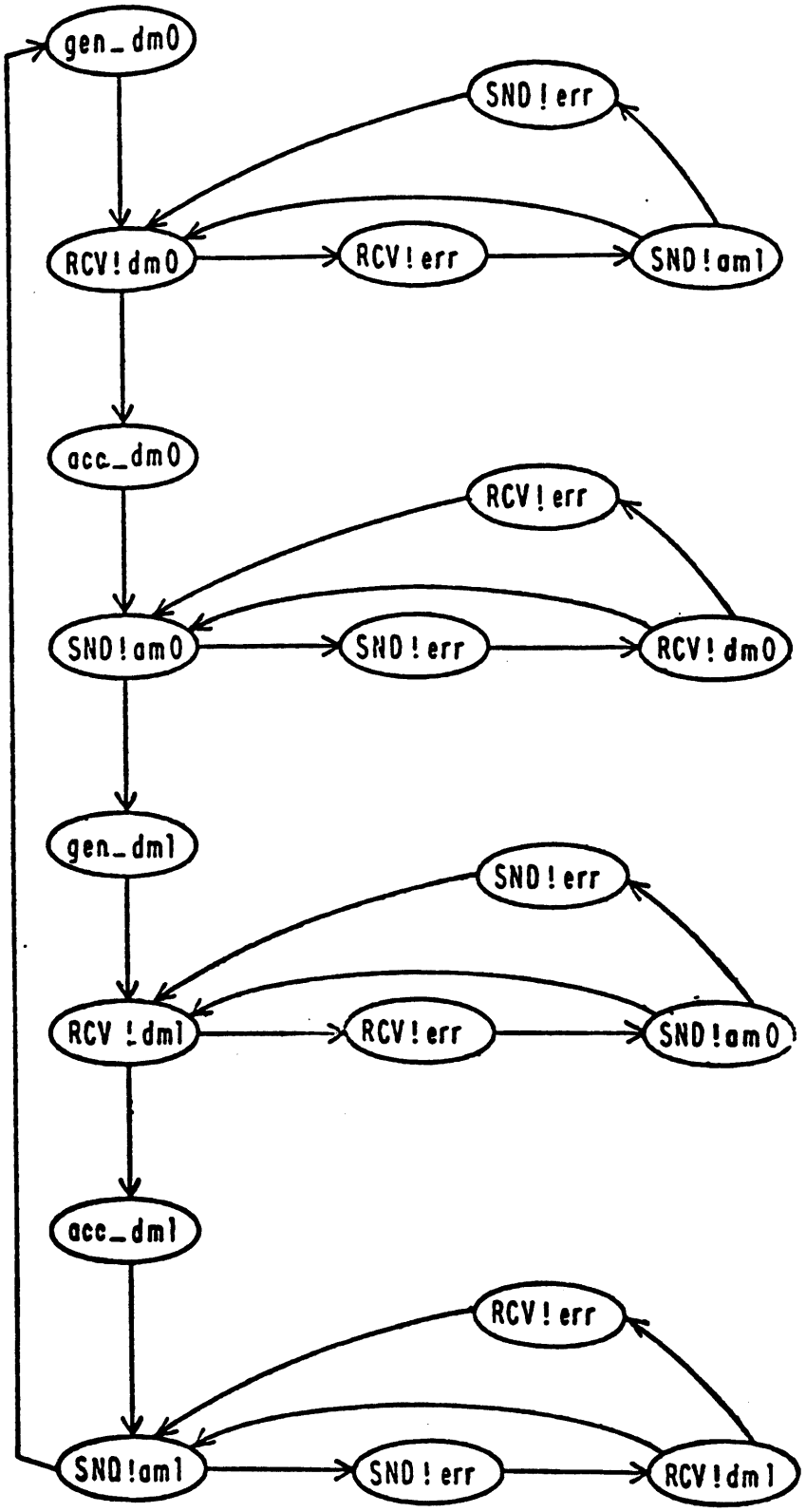


Figure 4-5: Global state transition graph for alternating bit protocol.

4.6. Extended Logics

In this section we consider logics which are more expressive than CTL and investigate their usefulness for automatic verification of finite state concurrent systems. CTL severely restricts the type of formula that can appear after a path quantifier. In CTL* we relax this restriction and allow an arbitrary formula of linear time logic to follow a path quantifier. We distinguish two types of formulae in giving the syntax of CTL*: state formulae and path formulae. Any state formulae is a CTL* formula.

<state-formula>::= <atomic proposition> |
<state-formula> ∧ <state-formula> |
¬ <state-formula> |
E(<path-formula>)

<path-formula>::= <state-formula> |
<path-formula> U <path-formula> |
¬<path-formula> |
<path-formula> ∧ <path-formula> |
X <path-formula> |
F<path-formula>

We use the abbreviation Gf for ¬F¬f and A(f) for ¬E¬(f). We interpret state formulae over states of a structure and path formulae over paths of a structure in a natural way. The truth of a CTL* formula in a state of a structure is inductively defined. A formula of the form E(<path formula>) is true in a state iff there is a path in the structure starting from that state on which the path formula is true. The truth of a path formula is defined in much the same way as for a formula in linear temporal logic if we consider all the immediate state - subformulae as atomic propositions [EH83]. BT^* will denote the subset of the above logic in which path formulae only use the F operator. CTL^+ will denote the subset in which the temporal operators X, U, F are not nested.

Fairness can be easily handled in CTL*. For example, the following formula asserts that on all fair executions of a concurrent system with n processes, R eventually holds:

$$A((GFP_1 \wedge GFP_2 \wedge ...GFP_n) \rightarrow FR)$$

Here P_1, P_2,...P_n hold in a state iff that state is reached by execution of one step of process P_1, P_2,...P_n, respectively.

THEOREM 4.3: *The model checking problem for* CTL* *is* PSPACE-*complete.* □

Proof Sketch: We wish to determine if the CTL* formula f is true in state s of structure M. Let g be a subformula of f of the form E(g') where g' is a path formula not containing any path quantifiers. For each such g we introduce an atomic proposition Q_g. Let f' be the formula obtained by replacing each such subformula g in f by Q_g. We modify M by introducing the extra atomic-propositions Q_g. Each Q_g is true in a state of the modified structure iff g is true in the corresponding state in M. The latter problem can be solved in polynomial space using the algorithm given in Chapter 2. f is true at state s in M iff f' is true in state s in the modified structure. We successively repeat the above procedure, each time reducing the depth of nesting of the path quantifiers.

It is easily seen that the above procedure takes polynomial space. Model checking for CTL* is PSPACE-hard because model checking for formulas of the form E(g'), where g' is free of path quantifiers, is shown to be PSPACE-hard in Chapter 2. □

THEOREM 4.4: *The model checking problem for* BT^* *(*CTL^+*)*
is both NP-*hard and co*-NP-*hard, and is in* Δ_2^P. □

Proof Sketch: The lower bounds follow from the results in Chapter 2. In Chapter 2 it was shown that the model checking problem for formulas of the form F(g'), where g' is free of path quantifiers and uses the only temporal operator F, is in NP. Using this result and a procedure like the one in the proof of previous theorem it is easily seen that the model checking problem for BT^* is in Δ_2^P. A similar argument can be given for CTL^+. □

We believe that the above complexity results justify our approach in section 4.4 where fairness constraints are incorporated into the semantics of the logic in order to obtain a polynomial-time model checking algorithm.

4.7. Conclusion

Much research in protocol verification has attempted to exploit the fact that protocols are frequently finite state. For example, in [Za80] and [Si] (global-state) *reachability tree* constructions are described which permit mechanical detection of system deadlocks, unspecified message receptions, and non-executable process interactions in finite-state protocols. An obvious advantage that our approach has over such methods is flexibility; our use of temporal logic provides a uniform notation for expressing a wide variety of correctness properties. Furthermore, it is unnecessary to formulate protocol specifications as reachability assertions since the model checker can handle both safety and liveness properties with equal facility.

The use of temporal logic for specifying concurrent systems has, of course, been extensively investigated ([MP81], [HO80], [OL80]). However, most of this work requires that a proof be constructed in order to show that a program actually meets its specification. Although this approach can, in principle, avoid the construction of a global state machine, it is usually necessary to consider a large number of possible process interactions when establishing non-interference of processes. The possibility of automatically synthesizing finite state concurrent systems from temporal logic specifications has been considered in [CE81] and [MW81]. But this approach has not been implemented, and the synthesis algorithms have exponential-time complexity in the worst case.

Perhaps the research that is most closely related to our own is that of Quielle and

Sifakis ([QS81], [QS82]), who have independently developed a system which will automatically check that a finite state CSP program satisfies a specification in temporal logic. The logical system that is used in [QS81], is not as expressive as CTL, however, and no attempt is made to handle fairness properties. Although fairness is discussed is [QS82], the approach that is used is much different from the one that we have adopted. Special temporal operators are introduced for asserting that a property must hold on fair paths, but neither a complexity analysis nor an efficient model checking algorithm is given for the extended logic.

Chapter 5

A Multiprocess Network Logic with Spatial and Temporal Operators

5.1. Introduction

. One of the fundamental models of parallel computation is a collection of synchronous processors with fixed inter-connections. For example, the iterative linearly connected, mesh connected, and multidimensional arrays of [Ko69] and [Co69], the shuffle exchange networks of [St71] and ultracomputer of [Sc80], and the cube connected cycle networks of [PV79].

Parallel algorithms for such networks are difficult to formally describe and prove correct. For example, the systolic algorithms of [KL78] are not formally proved correct in that paper; instead informal "picture proofs" are presented.

An informal description of a program or algorithm for a fixed connection network would likely make reference to the spatial relationships between neighboring processes and the properties holding for all processes, as well as the transformations over time. Indeed, natural English allows expressions of spatial modal operators such as *everywhere, somewhere, across such and such connection*, as well as temporal modal operators such as *until,eventually, hereafter*, and *nexttime*. However, natural English cannot suffice for formal semantics. This paper proposes a formal logic allowing use of these modal operators in the context of a fixed connection network.

Previous program logics contained only temporal modal operators [Pn77], [MP81] or modal operators for the effect of program statements [FL79]. Temporal logic has been used to reason about parallel programs; however it is impractical to use this logic to reason about large number of processes operating synchronously and communicating through fixed connections. Our use of spatial as well as temporal modal operators is a new idea. (Note: our spatial modal operators differ in an essential way from the modal operators of dynamic logic; see Section 5.2). This combination of temporal and spatial modal operators allows us to formally reason about computations on networks with complex connections.

The contribution of this chapter is more than simply the definition of our logic. We also describe applications and investigate the computational complexity of decision procedures.

Section 5.2 defines the logic. Section 5.3 describes some interesting applications of our logic to routing on the shuffle exchange network, to the firing squad problem on a linear array, and to systolic computations on arrays. We felt these examples to multiprocess networks illustrate the general applicability.

Section 5.4 investigates the problem of deciding validity of formulae of our logic. We show the set of valid formulas is Π_1^1-complete. However, in practice we are generally only interested in deciding validity of a proportional formula with respect to a given finite network. We show that given a finite network and a formula, the problem of deciding if the formula is valid in all models over the given network, is PSPACE-complete.

5.2. Definitions and Notation

First we define the propositional version of the logic. At the end of this section we briefly describe the first order version of this logic.

5.2.1. Networks

Let L be a countable set of symbols, which we call *links.* A *network* G = (P,E) contains a countable set of *processes* P and a partial mapping E: $L \times P \rightarrow P$. For each process $p \in P$ and label $\ell \in L$, $E(\ell,p)$ is (if defined) the process connected to p by *link* ℓ. For example, a square grid network might have links *up, down, left,* and *right.* The links are different from the atomic programs of PDL due to the restrictions given in the next page.

5.2.2. Syntax of the Logic

We distinguish as *temporal* modal operators the symbols F, G, U, X; for readability purpose sometimes we use the mnemonics *eventually, hereafter, until,* and *nexttime* respectively for the above mentioned temporal operators. The spatial modal operators are *somewhere, everywhere,* and any symbol in the set of links L, which we assume contains none of the previously mentioned modal operators.

Let $\mathcal{P}$ be the infinite set of *atomic* propositions. The well formed formulae are inductively defined as follows: An atomic proposition is a well-formed formula; if f_1, f_2 are well formed formulae then so are $\neg f_1$, $f_1 \wedge f_2$, Ff_1, Gf_1, Xf_1, f_1 U f_2, *somewhere*(f_1), *everywhere*(f_1), $\ell(f_1)$ where $\ell \in L$.

5.2.3. Semantics

A *Model* $\mathcal{M}$ is a 5-tuple (S,Ψ,Δ,G,π) where:

(i) S is the set of *states*,

(ii) $\Psi : S \rightarrow 2^{\mathcal{P}}$,

(iii) Δ: $(L \cup \{nexttime\}) \times S \rightarrow S$, is a partial function,

(iv) $G = (P,E)$ is a network, and

(v) π: $S \rightarrow P$.

Thus for each state $s \in S$, $\Psi(s)$ is the set of atomic formulas which are true in s, and $\pi(s)$ is the process associated with state s. Also $\Delta(nexttime, s)$ is the state occurring in the next time instance if the current state is s , and $\Delta(\ell,s)$ is the current state of the process connected to process $\pi(s)$ by link ℓ.

We extended Δ as a partial mapping to the domain $(L \cup \{nexttime\})^* \times S$ so that for all $s \in S$, $\Delta(\varepsilon,s) = s$ where ε is the empty string , and $\Delta(\ell_1 \cdot \ell_2 , s)$ is defined iff $\Delta(\ell_1, s)$ and $\Delta(\ell_2, \Delta(\ell_1, s))$ are defined and in this case $\Delta(\ell_1 \cdot \ell_2, s) = \Delta(\ell_2, \Delta(\ell_1, s))$. Similarly we also extend E as a partial mapping to the domain $L^* \times P$.

A model $\mathcal{M}$ is *proper* iff the following five conditions are satisfied:

R1: For each link $\ell \in L$ and each state $s \in S$, $\Delta(\ell \cdot nexttime,s) = \Delta(nexttime \cdot \ell,s)$ (thus *nexttime* commutes with respect to each link; this presumes the processes are synchronous).

R2: For each state $s \in S$, $\Delta(nexttime,s)$ is defined and $\pi(s) = \pi(\Delta(nexttime,s))$ (thus the name of each process is invariant over time).

R3: For each state $s \in S$ and link $\ell \in L$, $E(\ell,\pi(s))$ is defined iff $\Delta(\ell,s)$ is defined and in this case, $E(\ell,\pi(s)) = \pi(\Delta(\ell,s))$.

R4: For any $\alpha,\alpha' \in L^*$ and state $s \in S$ if $E(\alpha,\pi(s))$, $E(\alpha',\pi(s))$ are defined and $E(\alpha,\pi(s)) = E(\alpha',\pi(s))$ then $\Delta(\alpha,s) = \Delta(\alpha',s)$. (Thus the relationship between the states of two processes is independent of the particular paths of links over which they are connected.)

R5: If $\pi(s_1)=\pi(s_2)$ then for some $i\geq 0$ $\Delta(nexttime^i,s_1)=s_2$ or $\Delta(nexttime^i,s_2)=s_1$.

Hereafter, we consider only proper models.

Let us fix the model $\mathcal{M}$. We define truth of a formulae at a given state $s \in S$ by structural induction on the formula.

For each atomic proposition $Q \in \mathcal{P}$, $s \models Q$ iff $Q \in \Psi(s)$;

For any f_1, f_2

$s \models f_1 \wedge f_2$ iff $s \models f_1$ and $s \models f_2$;

$s \models \neg f_1$ iff not $(s \models f_1)$;

$s \models$ *nexttime* f_1 iff $\Delta(nexttime, s) \models f_1$;

$s \models$ *eventually* f_1 iff $\exists k \geq 0$ $\Delta(nexttime^k,s) \models f_1$;

$s \models$ *hereafter* f_1 iff $\forall k \geq 0$, $\Delta(nexttime^k,s) \models f_1$;

$s \models f_1$ *until* f_2 iff $\exists k \geq 0$ $\Delta(nexttime^k,s) \models f_2$ and $\forall i, 0 \leq i < k$, $\Delta(nexttime^i,s) \models f_1$;

$s \models \ell f_1$ iff $\Delta(\ell,s)$ is defined and $\Delta(\ell,s) \models f_1$;

$s \models$ *somewhere* f_1 iff $\exists\ \alpha \in L^*$, such that $\Delta(\alpha,s)$ is defined and $\Delta(\alpha,s) \models f_1$;

$s \models$ *everywhere* f_1 iff $\forall \alpha \in L^*$ ($\Delta(\alpha,s)$ is defined $\Rightarrow \Delta(\alpha,s) \models f_1$).

We let $\models_{\mathcal{M}}$ denote truth with respect to a given model $\mathcal{M}$.

5.2.4. Decision Problems

A formula f is said to be *satisfiable* (*valid*) if $s \models_{\mathcal{M}} f$ for some (all, respectively) model $\mathcal{M}$ and some (all) state s. Given a network G, a formula f is said to be G-*satisfiable* (G-*valid*) if $s \models_{\mathcal{M}} f$ for some (all) models $\mathcal{M}$ with network G and some(all) state s.

5.2.5. Extensions to First Order Logic

The first order version of this logic consists of the additional symbols like local variables, global variables, constant symbols, function and relation symbols, and the universal quantifier $\forall$. A term is defined as in the case of first order predicate calculus. An atomic formula is an atomic proposition or of the form $R(t_1 t_2 \ldots t_k)$ where R is k-ary relation symbol (R can be equality in which case $k = 2$) and $t_1, t_2, \ldots, t_k$ are terms. The additional requirement for the set of formulae is that if f is a formula and x is a global variable so is $\forall x$ (f). A model $\mathcal{M}$ is a 5-tuple $(\Sigma, S, \Delta, G, \pi)$ where $\Sigma = (D, \alpha, \beta)$ in which D is a countable domain from which the variables take values, α interprets relation and function symbols, β is mapping associating with each global variable and constant symbol a value from the domain; S is the set of states where each state is a mapping that associates a truth value with each atomic proposition and a value from D with each local variable; Δ, G, π are the same as in the propositional case. A proper model should satisfy the same conditions as for propositional case, modified in a natural way. We consider only proper models. Truth of an atomic formula in a state of a model is defined as in the case of first order predicate calculus; and truth of a formula in a state of a model is defined inductively as in the propositional version with the following addition; $s \models_{\mathcal{M}} \forall x$ f iff for each $c \in D$, $s \models_{\mathcal{M}_c} f$ where $\mathcal{M}_c$ is exactly same as $\mathcal{M}$ except that the global variable x is given the value c in $\mathcal{M}_c$. Satisfiability and validity of formulae are defined as usual.

5.3. Applications

This section gives some examples of the use of our logic to various multiprocess network applications.

5.3.1. Routing on a Shuffle-Exchange Network

A Shuffle-Exchange network $G=(P,E)$ where $P=\{0,1\}^n$ and $E: \{exchange, shuffle\} \times P \rightarrow P$ is defined as follows:

$E(exchange, (a_{n-1}, a_{n-2}, \ldots, a_0)) = (a_{n-1}, a_{n-2}, \ldots, \bar{a}_0)$

$E(shuffle, (a_{n-1}, a_{n-2}, \ldots, a_0)) = (a_0, a_{n-1}, \ldots a_1)$

for all $a_{n-1}, a_{n-2}, \ldots, a_0 \in \{0,1\}$.

Intuitively, the exchange edge connects processes p_1 and p_2 if all the bits of p_1 and p_2 are the same excepting the least significant bits which are distinct. The shuffle edge connects two processes p_1 and p_2, if p_2 is obtained by one cyclic shift of bits in p_1.

The routing problem in this network is to route a packet present at some process to a given destination traversing only along the shuffle and exchange edges.

We capture the name of a process by the atomic propositions $A_{n-1}, A_{n-2}, \ldots, A_0$. The formula f_0 asserts that the name of process is invariant over time;

$$f_0 = \bigwedge_{0 \le i < n} (\textit{hereafter}\ A_i \vee \textit{hereafter}\ \neg A_i)$$

f_1, f_2 assert that exchange and shuffle edges are properly connected.

$$f_1 = \bigwedge_{1 \le i < n} (A_i \leftrightarrow \textit{exchange}\ A_i) \wedge (A_0 \leftrightarrow \textit{exchange}\ \neg A_0)$$

$$f_2 = \bigwedge_{0 \le i < n} (A_i \leftrightarrow \textit{shuffle}\ A_{(i-1) \bmod n})$$

The presence of the packet at any process will be indicated by the atomic proposition X, and the destination by $D_{n-1}, D_{n-2}, \ldots, D_0$. We assume that the name of the destination

travels with the message. Let g_0 assert that X is true in at most one place at any time. It is not difficult to see that this can easily be expressed. g_1 asserts that the name of the destination travels with the packet.

$g_1 = X \bigwedge_{0 \le i < n} (D_i \supset \textit{hereafter everywhere}\,(X \supset D_i)) \wedge$

$\quad (\neg D_i \supset \textit{hereafter everywhere}\,(X \supset \neg D_i))$

g_2 asserts that the packet travels along the shuffle or exchange edges only.

$g_2 = X \supset \textit{nexttime}\,(X \vee \textit{shuffle} X \vee \textit{exchange}\, X)$

The main correctness property is g_3 which asserts that the packet reaches its destination eventually.

$g_3 = \textit{eventually somewhere}(X \bigwedge_{0 \le i < n} (A_i \leftrightarrow D_i))$

Let r be a formula which describes the actual routing algorithm. Then

$(\textit{hereafter everywhere}\,(r \wedge f_0 \wedge f_1 \wedge f_2 \wedge g_0 \wedge g_1 \wedge g_2)) \supset g_3$

is a valid formula iff the algorithm correctly routes packets.

Next we describe a specific routing algorithm for the shuffle-exchange network and derive the corresponding formula r for its semantics. The packet will be routed in n stages, where for $i = 0, \ldots, n\text{-}1$ if at the start of the i-th stage the packet is located at a process whose lowest order address bit is not the value of D_i, then the packet traverses the *exchange* link and reaches the $i+1$ stage. In either case, the packet next traverses a *shuffle* link.

To define a formula r for this routing algorithm, it is useful to introduce propositional variables $S_0, \ldots, S_{n-1}$ and require that only unique S_i be true at any process, and that S_i be invariant on traversing an *exchange* link but that $S_{(i+1) \bmod n}$ be true on traversing a *shuffle* link. Thus we let

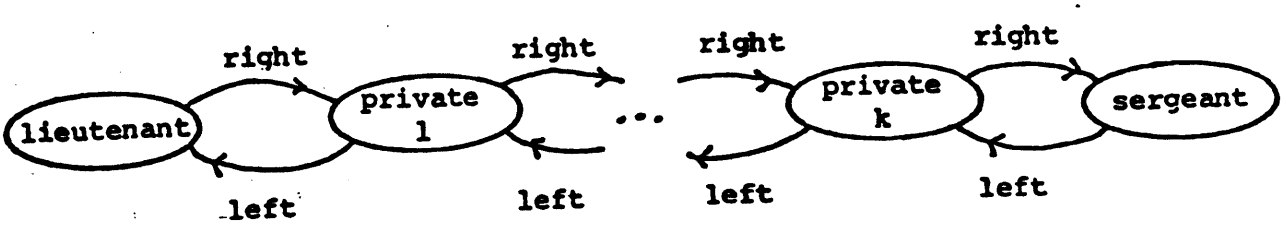
$$r_0 = \bigvee_{0 \le i < n} [(S_i \wedge \neg S_0 \wedge \ldots \wedge \neg S_{i-1} \wedge \neg S_{i+1} \wedge \ldots \wedge \neg S_{n-1})$$
$$\wedge (\textit{nexttime exchange}\ S_i) \wedge (\textit{nexttime shuffle}\ S_{(i+1) \bmod n})]$$

The formula for the semantics of this routing algorithm is given by r, where

$$r = r_0 \wedge [(X \wedge \bigvee_{0 \le i < n} (A_i \leftrightarrow \neg D_i))$$
$$\supset \bigwedge_{0 \le i < n} \{S_i \wedge ((A_0 \leftrightarrow D_i) \supset$$
$$\textit{nexttime exchange}\ X) \wedge ((A_0 \leftrightarrow \neg D_i) \supset \textit{nexttime shuffle}\ X)\}]$$

5.3.2. The Firing Squad Problem for a Linear Array

We briefly describe the problem and show how it's correctness can be specified in our logic. A solution to the firing squad problem consists of a linear array of deterministic finite state processes as shown in the following figure. The next state of each process is a function of it's present state and the states of it's neighbors. All the privates are identical processes. The problem is to obtain the programs for the lieutenant, the sergeant and the privates so that whenever the lieutenant is in a designated initial state, then eventually all the processes simultaneously enter a special state called the firing state, and none of them enters this state before this time. The solution should work for linear arrays of all sizes.



We assume that all processes have the state set Q={0,1,2, . . . , m}, and the state 0 is the initial state of each process. State 1 is the specific state into which the lieutenant enters

to start the operation. State m is the firing state. All the privates are identical. We use atomic propositions $P_0, P_1, \ldots, P_m$ to indicate the state of a process (P_i is true at a place iff the corresponding process is in state i at that instance). Now we assert the operation of the system as follows.

(i) "I" asserts that each process is in at most one state at any instance of time.

$$I = \textit{everywhere hereafter}[\ \bigwedge_{0 \le i < j \le k} (P_i \supset \neg P_j)]$$

(ii) f_0 asserts that the moves of the lieutenant is according to its next move partial function

$\delta_0 : Q^2 \rightarrow Q$.

$$f_0 = \textit{everywhere}[\neg \textit{left}(\text{true}) \supset \{(P_0 \vee P_1) \wedge \textit{hereafter} \bigwedge_{i,j} ((P_i \wedge \textit{right}\, P_j) \supset \textit{nexttime}\, P_{\delta_0(i,j)})\}]$$

Note that $\neg$*left*(true) is true only on the lieutenant, the left most processor.

(iii) Similarly let f_1, f_2 be the formulae that define the moves of all privates and the sergeant respectively. The positions of privates is identified by the truth of the formula

(*left*(True) $\wedge$ *right*(True)).

Note that the position of the sergeant is identified by the formula $\neg$*right*(True).

(iv) Let g_0 be the formula that asserts that if any process (other than the lieutenant) and all its neighbors are in state 0 then it remains in state 0 in the next step. It is easily seen that this can also be asserted.

Now we assert that if all the above conditions are met and at any time the lieutenant enters the state 1 then all processes will eventually enter the firing state simultaneously at some future instance, and none of them will be in the firing state before that instance. This is captured by the formula g.

$$g = (I \wedge f_0 \wedge f_1 \wedge f_2 \wedge g_0) \supset \textit{hereafter}[\textit{somewhere}(\neg \textit{left}(\text{true}) \wedge P_1) \supset$$
$$\{(\neg \textit{somewhere}\, P_m)\ \textit{until}\ (\textit{everywhere}\, P_m)\}]$$

g is valid on all models with linear arrays as networks iff the given solution to the firing squad problem is correct. A similar construction can be given for the firing squad problem over any given network.

5.3.3. Systolic Arithmetic Computations

The systolic algorithms of [KL78] are not formally proved correct in their paper; instead they present informal "picture proofs". Our logic is thus particularly useful here when extended to first order formulae (as described in Section 2.5).

We consider an interesting example of a network for matrix-vector multiplication due to ([KL78],[Le81]). The matrix is an infinite band matrix of bandwidth (n+1). The network architecture is shown in figure 5.1 .

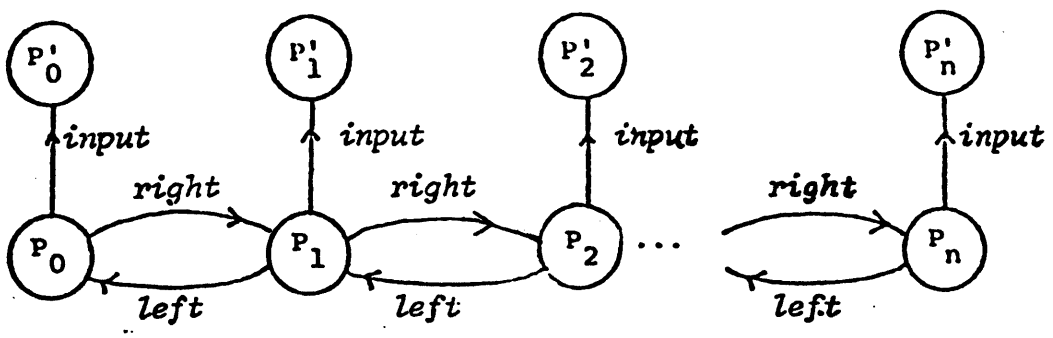


Figure 5-1:

The main processors are $P_0, P_1, \ldots, P_n$. The processors $P'_0, P'_1, \ldots, P'_n$ are the input processors, each of them contains a variable Z. The values of Z in P'_i change with time and they represent the values of the i^{th} diagonal of the matrix. This variable takes the successive values of the i^{th} diagonal at alternate time instances and takes value zero at the intermediate time instances. Each processor P_i has two variables X,Y. The values of the variable X in P_0 over time represent the input vector. This variable takes the successive values of the input vector at alternate time instances and takes value zero at the intermediate time instances. The values of X move right with each time instance. The details of the relative timings are given in [Le81] and the reader is referred to this.

Thus,

$$g_1 = \mathit{left}(\text{true}) \supset \forall\alpha(\mathit{left}(X = \alpha) \leftrightarrow \mathit{nexttime}(X = \alpha))$$

asserts that the value of X at the next time instance in processor P_i $(i > 0)$, is the present value of X in the process left to P_i.

At each step $P_i(i > N)$ computes its value of Y to be the sum of the previous value of Y in process P_{i+1}, plus the product of X in P_i times Z in P'_i. This is captured by

$$g_2 = \mathit{right}(\text{true}) \supset \forall\alpha\forall\beta(\mathit{right}(Y = \alpha) \wedge \mathit{nexttime\ input}(Z = \beta)$$
$$\supset \mathit{nexttime}(Y = \alpha + X \cdot \beta))$$

At each step P_n computes its value of Y to be the product of the value of X in P_n and the value of Z in P'_n. This can also be easily asserted by the formula

$$g_3 = [\neg \mathit{right}(\text{True}) \wedge \mathit{input}(\text{true})] \supset \forall\alpha\forall\beta[(X = \alpha \wedge \mathit{input}(Z = \beta)) \supset \mathit{nexttime}(Y = \alpha \cdot \beta)].$$

(note that $\neg \mathit{right}(\text{True}) \wedge \mathit{input}(\text{true})$ holds only for processor P_n)

The steady state correctness property at P_0 can thus be expressed in our logic as

$$\mathit{hereafter\ everywhere}(g_1 \wedge g_2 \wedge g_3) \supset \mathit{hereafter}\ h$$

where

$$h = [\neg left(True) \wedge input(\text{true})] \supset$$
$$\forall \alpha_0 \ldots \alpha_n \forall \beta_0 \ldots \beta_n [(\bigwedge_{0 \le i \le n} nexttime^{2i}(X = \alpha_i) \wedge$$
$$nexttime^{n+i} right^{n-i} input\,(Z = \beta_i))$$
$$\supset nexttime^{2n}(Y = \sum_{0 \le i \le n} \alpha_i \cdot \beta_i)]$$

5.4. Decidability and Complexity Issues

In this section we consider issues of decidability and complexity of different versions of our logic. Recall that a formula is said to be satisfiable iff there exists a model and a state at which the formula is true. A formula is said to be valid if it is true in all states of all models. We say that a formula is satisfiable (valid) on finite networks if the formula is true in some (all) model with finite networks.

THEOREM 5.1: *The set of satisfiable formulae of multiprocessor network logic is* Σ_1^1 -complete *and the set of valid formulae is* Π_1^1-complete.

Proof sketch: First we show that the set of satisfiable formulae is a Σ_1^1-complete set. From this result it can easily be shown that the set of valid formulae is Π_1^1-complete.

We consider a deterministic Turing machine M on infinite strings. M has one read only infinite input tape, and an infinite work tape. An infinite string is input to M on its input tape. M never halts. M is said to accept an input if during its computation it goes into any of a set of final states infinitely often. The set of encodings of all Turing machines that accept at least one input, is shown to be Σ_1^1-complete in [SCFG82] . We reduce this set to the set of satisfiable formulae. An ID of M is the part of input is seen thus far, the contents of the work tape, the position of the head on the work tape. We define a sequence of IDs of

M during its computation on an input and express this sequence using a formula in the logic. We also assert that in this sequence, final IDs (IDs having a final state) appear infinitely often. Thus given an encoding of a Turing machine we obtain a formula that is satisfiable iff the Turing machine accepts at least one input. The details can easily filled up by the reader. □

Let $\mathcal{M} = (S, \Psi, \Delta, G, \pi)$ be a model where G=(P,E) is a is a finite network. Let φ:P→S. φ is said to be consistent with $\mathcal{M}$, if $\pi(\varphi(p))=p$ for all p∈P, and for all p_i, p_j if $p_j = E(\ell,p_i)$ for some $\ell \in L$, then $\varphi(p_j) = \Delta(\ell, \varphi(p_i))$. Let $\Phi = \{\varphi \mid \varphi$ is consistent with $\mathcal{M}\}$, and let *next*: $\Phi \rightarrow \Phi$ be such that for all $\varphi \in \Phi$ and for all p, $next(\varphi)(p) = \Delta(nexttime, \varphi(p))$. $\mathcal{M}$ is said to be *ultimately periodic* with starting index ℓ and period m, if for all $\varphi \in \Phi$, $next^i(\varphi) = next^{i+m}(\varphi)$ for all $i \geq \ell$. For any formula f, let SF(f) be the set of subformulae of f, and for any $\varphi \in \Phi$, let $[\varphi]$:P → $2^{SF(f)}$ such that $[\varphi](p)=\{g \mid g \in SF(f)$ and $\varphi(p) \models g\}$. We require a technical lemma characterizing satisfiability. This lemma can be proved on the same lines as the corresponding lemma for PTL given in Chapter 2.

LEMMA 5.2: f *is satisfiable in a model over a finite network iff* f *is satisfiable over an ultimately periodic model over a finite network.* □

THEOREM 5.3: *The set of formulae that are satisfiable in a model over a finite network is* Σ_1^0-complete,*and the set of valid formulae in models over finite networks is* Π_1^0-complete.

Proof: As in the previous theorem, we can reduce the halting problem of Turing machines over finite strings to the set of satisfiable formulae in a model over a finite network. We give a Turing machine M which accepts the above set. M guesses a finite

network and an ultimately periodic model over this network. It next verifies that f is satisfiable in this model. M halts only on the input formulae that are satisfiable in a model over a finite network. □

THEOREM 5.4: *The following problem is* PSPACE-*complete. Given a finite network* G, *and a formula* f, *is* f *satisfiable in a model over the network* G ?

Proof: The PSPACE-hardness of the problem follows from the PSPACE-hardness of satisfiability for PTL given in Chapter 2. We give a polynomial space bounded Turing machine M that checks if f is satisfiable in a model over the network G. M guesses $[\varphi]$, and verifies for consistency and that $f \in [\varphi](p)$ for some $p \in P$. At each subsequent instance M guesses $[next(\varphi)]$ and checks that it is consistent with φ. It continues this each time keeping $[\varphi]$ and $[next](\varphi)$. At a certain instance it guesses the beginning of the period and saves the corresponding $[\varphi]$. It continues the previous process, each time guessing either $[next(\varphi]$ or guessing that it is the end of the periodic part. In the latter case it takes $[next(\varphi)]$ to be the saved value at the beginning of the period. Each time M guesses $[next(\varphi)]$ it verifies that $[\varphi]$ is consistent with $[next(\varphi)]$. M also verifies that certain formulae are fulfilled in the periodic part. M clearly uses space polynomial in the size of G and the size of f. □

5.5. Conclusion

. We have proposed a logic to reason about computations of multiprocessor networks. We feel that our logic will be useful to specify the semantics and prove correctness of multiprocess networks. No such formal system for multiprocessor networks had been proposed previously. We have examined the application of our logic to some diverse multiprocess network problems, and presented some results in decidability and complexity of our logic.

All the applications we presented are synchronous systems. However, our logic can also be used for asynchronous distributed systems.

Chapter 6

Characterization and Axiomatization of Message Buffers in Temporal Logic

6.1. Introduction

Exchange of information between executing processes is one of the primary reasons for process interaction. Many distributed systems implement explicit message passing primitives to facilitate intercommunication. Typically, a process executes a *write* command to pass a message to another process, and the target process accepts the message by executing a *read* command. The semantics of *write* and *read* may differ considerably depending on the methods used for storing or buffering messages that have been sent but not yet accepted by the receiving process.

Because message passing systems are so widely used, it is important to develop formal techniques for reasoning about them. In this chapter we investigate the possibility (impossibility) of using linear temporal logic to characterize the semantics of different message buffering mechanisms.

Specifically, we consider FIFO buffers (*queues*), LIFO buffers (*stacks*) and unordered buffers (*bags*). The set of distinct messages that can be written into the buffer is called the *message alphabet*. We specify a message buffer as the set of all valid infinite input/output message sequences. Thus, characterizing a message buffer in temporal logic consists of obtaining a formula that is true exactly on these sequences. For *unbounded* buffers, we

show that it is *impossible* to obtain such a formula in first order linear temporal logic that is *independent* of the underlying interpretation (i.e. message alphabet). Nor is it possible to obtain such a formula in propositional linear temporal logic (PTL) when the message alphabet is finite. It is possible ,however, to give a formula in first order linear temporal logic that gives a domain-independent characterization of bounded buffers. In fact, if the message alphabet is finite, then such a formula can be expressed in PTL. Although such bounded message buffers can be characterized using ω-regular expressions (or monadic second order theory of one successor), it is not obvious that they can be expressed in PTL since this logic is provably less expressive than ω-regular expressions [Wo81]. We show how we can characterize bounded buffers elegantly in QPTL with one level of existential quantification.

We also consider the problem of axiomatizing the various types of message buffers. A model of a message buffer is an infinite sequence of states denoting a series of legal read/write operations on the buffer. The theory of a message buffer is the set of all PTL formulae which are true in all models of the buffer. Since bounded buffers over finite alphabet can be characterized in PTL and since PTL has a complete axiom system it can easily be shown that bounded buffers are axiomatizable in PTL. We show that, in general, unbounded FIFO buffers are not axiomatizable. Surprisingly, it is possible to axiomatize unbounded LIFO buffers and unbounded unordered buffers; in fact, the theories of these buffers are decidable.

This chapter is organized as follows: Section 6.2 defines the additional notation that we use in the remainder of this chapter. In section 6.3 we specify precisely those properties of message buffers that we would like to capture in temporal logic. Section 6.4 shows that bounded buffers can be characterized in the logic and describes how uninterpreted auxiliary

proposition symbols can be added to simplify this construction. In section 6.5 we prove that it is impossible to give a domain independent characterization of unbounded message buffers in first order temporal logic. We also show that unbounded FIFO message buffers are not axiomatizable in PTL while unbounded LIFO and unordered buffers are axiomatizable. The chapter concludes in section 6.6 with a summary and discussion of our results.

6.2. Definitions

In the previous chapters we defined the syntax and semantics of PTL . Here we define a restricted version of the first order temporal logic. The language of this logic includes variables, function symbols, relation symbols and the universal quantifier in addition to the symbols in the propositional version of the logic. The *type* of the language is a tuple which gives the function symbols, the relation symbols with their arities. The variables are partitioned into two groups: local variables whose values depend on the current state and global variables whose values are state independent. Atomic formulae have the same syntax as in the usual first order case. The set of well formed formulae is the smallest set containing the atomic formulae and closed under universal quantification over global variables, boolean connectives, and the above temporal operators.

A *model* T is a triple (Δ,α,s) where Δ is the *domain*; α assigns meanings to the function symbols, relation symbols, and global variables; and s is a *ω-sequence* of states. A *state* assigns values from Δ to the local variables and truth values to the atomic propositions. An interpretation in this case is a pair $\langle T,i\rangle$ where T is a model and $i \geq 0$ specifies the present state. Truth of an atomic formula in an interpretation is defined as in the usual first order case; truth of a composite formula is defined as in the case of propositional temporal logic with the following addition: $T,i \models \forall x(f)$ iff for each $c \in \Delta$ $T_c,i \models f$ where T_c is T with the meaning assigned to the global variable x changed to the value c.

6.3. What Are Message Buffers?

We characterize a message buffer by the set of legal *read/write* sequences allowed on the buffer. A *write* operation writes a message into the buffer; a *read* operation reads a message from the buffer and deletes it. At most one read or write operation is permitted at any instant of time. In the case of bounded buffers a write request will be rejected when the buffer is full; similarly, a read request on an empty buffer will be rejected. Rejected read/write requests are not included in the sequences of legal operations characterizing the buffer. We consider below three types of message buffers: FIFO buffers *(queues)*, LIFO buffers *(stacks)*, and unordered buffers *(bags)*. In FIFO buffers the earliest written message in the buffer is the output for a read request; with LIFO buffers the latest written message in the buffer is used; and with unordered buffers any message present in the buffer is output. We also require that each physical message written into the buffer is ultimately read; this is the *liveness property of buffer behavior.*

Let Σ be the message alphabet and $\mathcal{P}_\Sigma$ be the set of atomic propositions $\{R_\sigma \mid \sigma \in \Sigma\} \cup \{W_\sigma \mid \sigma \in \Sigma\}$. Let $\mathcal{P} \supseteq \mathcal{P}_\Sigma$ be the set of atomic propositions in the language.

$ST = \{\varphi \mid \varphi: \mathcal{P} \rightarrow \{True,False\}$ such that $\varphi(P) = True$ for at most one P in $\mathcal{P}_\Sigma\}$

We consider each member of ST to be a state; if $R_\sigma(W_\sigma)$ true in a state, then it indicates that the message σ is read (written) from (into) the buffer in that state.

Let $t \in ST^* \cup ST^\omega$ and $i_0 < i_1 < \ldots$ be all the instances at which some messages $\sigma_0,\sigma_1,\ldots$ are read from the buffer, i.e., $t_{i_k}(R_{\sigma_k}) = True$ for $k \geq 0$. Then $\pi_r(t)$ denotes the sequence $(\sigma_0,\sigma_1,\ldots)$. Similarly, we define $\pi_w(t)$. Let $t^{(i)}$ denote the sequence $(t_0,t_1,\ldots,t_i)$, then $nb_i = length(\pi_w(t^{(i)})) - length(\pi_r(t^{(i)}))$ is the number of messages in the buffer just after the instance i.

$FS_{\Sigma,k}$ is the set of all infinite sequences of states which denote legal series of read/write operations on a FIFO buffer of size k. $LS_{\Sigma,k}$ and $US_{\Sigma,k}$ are the corresponding sets of sequences for LIFO and unordered buffers respectively. Unbounded buffers will be denoted in this scheme of notation by $k = \infty$.

For $k \geq 0$ and $k = \infty$

$FS_{\Sigma,k} = \{t \in ST^{\omega} \mid$ for all $i \geq 0$, $0 \leq nb(i) \leq k$ and $\pi_r(t^{(i)})$ is a prefix of $\pi_w(t^{(i)})$ and $\pi_r(t) = \pi_w(t)\}$.

$LS_{\Sigma,k} = \{t \in ST^{\omega} \mid$ for all $i \geq 0$, $0 \leq nb(i) \leq k$ and if for some $\sigma \in \Sigma$, $t, i \models W_{\sigma}$ then there exists $j > i$ such that $t,j \models R_{\sigma}$, $nb(j-1) = nb(i)$ and $\forall \ell\ i \leq \ell \leq j-1$ $nb(\ell) \geq nb(i)\}$

$US_{\Sigma,k} = \{t \in ST^{\omega} \mid$ for all $i \geq 0$, $0 \leq nb(i) \leq k$ and for all $\sigma \in \Sigma$, the number of writes of the message σ up to $i \geq$ the number of reads of the message σ up to i, and for infinitely many i, $nb(i) = 0\}$

In the case of both LIFO and unordered buffers we require that the buffer should become empty infinitely often, in order to satisfy the liveness requirement.

For a finite alphabet Σ, a formula f in PTL *characterizes* a FIFO message buffer of size k (unbounded FIFO buffer) if $\forall t \in ST^{\omega}$ $t,0 \models f$ iff $t \in FS_{\Sigma,k}$ ($t \in FS_{\Sigma,\infty}$).

Similarly we define what it means to characterize LIFO and unordered buffers in PTL.

Let L be a language of first order linear temporal logic of type τ with local variables read-val, write-val and with mutually exclusive atomic propositions R,W. Let $T = (\Sigma,\alpha,s)$

be a model of type τ. In any state s_i, if $s_i(R)$ = True then it signifies the reading of message s_i(read-val) in that state. Similarly s_i(write-val) denotes the message written if $s_i(W)$ = True. With s, we associate any sequence t defined as follows:

For all $i \geq 0$ $t_i : \mathcal{P} \rightarrow$ {True,False} such that for all $\sigma \in \Sigma$,

$t_i(R_\sigma)$ = True iff $s_i(R)$ = True and s_i(read-val) = σ;

$t_i(W_\sigma)$ = True iff $s_i(W)$ = True and s_i(write-val) = σ.

A first order linear temporal formula f of type τ is a *(domain independent) characterization* of a FIFO buffer of size k (unbounded FIFO buffer) if for all T = (Σ,α,s) of type τ T,0 $\models$ f iff t $\in$ $FS_{\Sigma,k}$ (t $\in$ $FS_{\Sigma,\infty}$). Similar definitions hold for LIFO and unordered buffers.

A model of a message buffer is an infinite sequence of states denoting a legal series of read/write operations on the buffer, as given above. The theory of a message buffer is the set of all PTL formulae which are true in all interpretations (t,i) where t is a model of the buffer. We say that a message buffer is axiomatizable if there exists a recursive set of axioms from which the formulae in the theory of the buffer can be deduced using some inference rules.

6.4. Characterizing Bounded Buffers

In this section we characterize bounded buffers over a finite alphabet using propositional linear temporal logic; we also give domain independent characterizations in first order linear temporal logic. We let fb_k, ℓb_k, ub_k denote formulae in propositional temporal logic characterizing FIFO, LIFO, and unordered message buffers of size k over the finite message alphabet Σ. First we describe how to obtain the formulae for buffer size = 1 and 2, and show how it can be extended to the general case.

Let Σ be a finite message alphabet, and $\mathcal{P}_\Sigma = \{R_\sigma \mid \sigma \in \Sigma\} \cup \{W_\sigma \mid \sigma \in \Sigma\}$ be the set of atomic propositions. Throughout this section we use the following abbreviations:

$$W = \bigvee_{\sigma \in \Sigma} W_\sigma$$

$$R = \bigvee_{\sigma \in \Sigma} R_\sigma$$

$$Ex = \bigwedge_{\sigma_1 \neq \sigma_2} \neg (R_{\sigma_1} \wedge R_{\sigma_2}) \wedge \bigwedge_{\sigma_1 \neq \sigma_2} \neg (W_{\sigma_1} \wedge W_{\sigma_2}) \wedge \neg(W \wedge R)$$

$$I = G(Ex)$$

'I' asserts that at any instant at most one operation occurs on the buffer, and *reads, writes* are mutually exclusive.

In the case of buffer size = 1 the buffer behavior is as follows:

1. The *writes* and *reads* occur alternately;
2. The message read in each *read* operation is the message written by the previous *write* operation. Thus, $fb_1 = I \wedge f_a \wedge f_b$ where

$$f_a = G(W \supset X(\neg W \, U \, R)) \wedge G((R \wedge X(F \, R)) \supset X(\neg R \, U \, W));$$

$$f_b = G(\bigwedge_{\sigma \in \Sigma} (R_\sigma \supset (\neg W \, S \, W_\sigma))).$$

It is easily seen that f_a and f_b assert properties (1) and (2), respectively.

Intuitively, the operation of a buffer of size = 2 can be described as follows. Initially, *writes* and *reads* occur alternately; whenever a *read* occurs the buffer becomes empty, and after each *write* the buffer will have exactly one message. This continues until two writes occur successively without a *read* operation in between, and the buffer becomes full

(formula ℓ_2 expresses this). Subsequently, *reads* and *writes* will again begin to alternate. After each *read* the buffer will have one message and after each *write* operation the buffer becomes full. This may continue forever, or until two *reads* occur successively without a *write* in between, making the buffer empty (r_2 expresses this); now the previous sequence repeats. This behavior is common for FIFO, LIFO and unordered buffers of size = 2. The formula ℓ_2, r_2 are given below:

$$\ell_2 = W \wedge (\neg R \, U \, W)$$

$$r_2 = R \wedge (\neg W \, S \, R)$$

In the remainder of this section we will frequently use the formula alt(p,q,c) given below:

$$alt(p,q,c) = [(g \, U \, c) \vee G(g \wedge \neg c)] \wedge [(\neg c \, U \, p) \supset (\neg q \, U \, p)]$$

where

$$g = (p \supset X(\neg p \, U \, q)) \wedge (q \supset [X(\neg q \, U \, p) \vee X(\neg q \, U \, c)])$$

The first conjunct in alt(p,q,c) asserts that either there is a future instant at which c occurs and until this instant p,q occur alternately, or throughout the future p,q occur alternately without c occurring anywhere. The second conjunct asserts that if p occurs then it occurs before q. Thus, the previous intuitive description of the behavior of the buffer of size 2 is captured by the formula bv given below.

$$bv = alt(W,R,\ell_2) \wedge G[\ell_2 \supset X \, alt(W,R,r_2)] \wedge G[r_2 \supset X \, alt(W,R,\ell_2)]$$

bv asserts that ℓ_2,r_2 occur alternately with alternating *read* and *writes* occurring in between. Any *read* after ℓ_2 but before the next r_2 is on a full buffer, while any *read* after an r_2 but before the next ℓ_2 is on a buffer containing one message. The formulas read-on-full, read-

on-single given below characterize *reads* on a full buffer and *reads* on a buffer with one message, respectively.

$$\text{read-on-full} = R \wedge (\neg r_2 \, S \, \ell_2)$$

$$\text{read-on-single} = R \wedge [(\neg \ell_2) \, S \, r_2 \vee \neg(\text{True} \, S \, \ell_2)]$$

For FIFO buffers, a *read* on a full buffer reads the message written by the *write* before the previous *write.*

$$fb_2 = I \wedge bv \wedge g \wedge h \quad \text{where}$$

$$g = G(\text{read-on-full} \supset \bigwedge_{\sigma} (R_\sigma \supset [\neg W \, S \, (W \wedge Y(\neg W \, S \, W_\sigma))])),$$

$$h = G(\text{read-on-single} \supset \bigwedge_{\sigma} [R_\sigma \supset (\neg W \, S \, W_\sigma)]).$$

The formula on the left side of '$\supset$' in g is true when *reads* occur on a full buffer, while the formula on the right side asserts that the message read at these instances is the message written by the last but one *write* operation. 'h' asserts that *read* operations on a buffer containing a single message, read the message written by the previous *write* operation.

THEOREM 6.1: *For any infinite sequence of states* t, t,0 $\models fb_2$ *iff* $t \in FS_{\Sigma,2}$. □

Let $t \in LS_{\Sigma,2}$. If t,i $\models r_2$, then there exists j < i such that t,j $\models \ell_2$. The message read at the instance i is the message written at the instance j. If t,i $\models R$ and t,i $\models \neg r_2$, then the message read at the instance i is the message written in the previous write operation. These properties are expressed by g' and h' respectively.

$$g' = G(r_2 \supset \bigwedge_{\sigma \in \Sigma} [R_\sigma \equiv \neg \ell_2 \, S \, (\ell_2 \wedge W_\sigma)])$$

$$h' = G((\neg r_2 \wedge R) \supset \bigwedge_{\sigma \in \Sigma} [R_\sigma \equiv \neg W \, S \, W_\sigma])$$

Let $b_2 = I \wedge bv \wedge g' \wedge h'$

THEOREM 6.2: *For any infinite sequence of states* t, t,0 $\models \ell b_2$ *iff* t $\in LS_{\Sigma,2}$. □

Let t $\in US_{\Sigma,2}$. Then for every $\sigma \in \Sigma$, for all $i \geq 0$ the number of messages of value σ written into the buffer up to the instance i is greater than or equal to the number of messages of value σ read from the buffer up to the instance i, and they do not differ by more than 2. For a given σ, we can obtain a formula bv_σ asserting the above property by replacing R by R_σ,W by W_σ in bv.

Let $ub_2 = I \wedge bv \wedge \bigwedge_{\sigma \in \Sigma} bv_\sigma$

The following theorem can be easily proved:

THEOREM 6.3: *For any infinite sequences of states* t, t,0 $\models ub_2$ *iff* t $\in US_{\Sigma,2}$. □

We have shown how the buffer behaviors can be expressed for buffer size = 2. We show below how we can express the buffer properties for arbitrary buffer sizes.

As before we use formulae with parameters ex: f(d) denotes a formula with parameter d which can be substituted for. Frequently we use the formula ALT(h_1,h_2,g(d),c). This formula is slightly different from the formula 'alt' we used before. It asserts that

either

there is a future instance i at which c holds and from the present up to i

(i) the instances at which h_1,h_2 are true occur alternately (i.e., between every two instances at which h_1 is true there is an instance where h_2 is true, and vice versa) with h_1 occurring first and h_2 occurring last and

(ii) whenever h_1 holds, at the immediate next instance g(h_2) holds and whenever h_2

holds, $g(h_1)$ holds at the next instance if this is not the last occurrence of h_2 before c, otherwise g(c) holds at the next instance. Also at i $g(h_1)$ holds

or there is no instance in future at which c is true and (i),(ii) hold throughout in the future.

It is easily seen that ALT can be expressed in PTL.

Let $t = (t_0,t_1,...) \in ST^{\omega}$. We say that the subsequence $(t_i,t_{i+1},...,t_j)$ is a RW pattern of a k buffer if the number of reads and writes in the subsequence are equal and at any point in the sequence the number of reads is no more than the number of writes up to that point and they do not differ by more than k.

We use the following intermediate formulae:

$NF_k(c)$: $t,i \models NF_k(c)$ iff $\exists\, j \geq i$ such that $t,j \models c$ and $(t_i,t_{i+1},...,t_{j-1})$ has RW pattern of a k buffer. In case $t,i \models c$, then $t,i \models NF_k(c)$.

L_{k+1}: Let d be the number of messages in the buffer before the i^{th} instance. $t,i \models L_{k+1}$ iff $\exists\, j > i$ such that the buffer has $d+k+1$ messages at j and at all instances between i and j (including i, j) the buffer has at least $d+1$ messages. This is shown in Figure 6.1.

We will obtain a formula for L_{k+1} $(k \geq 2)$ in terms of NF_k. Let $t,i \models L_{k+1}$ and d be the number of messages before the instance i, and j be as given in the definition of L_{k+1}. Let $m > i$ be the earliest instant such that the buffer has at least $d+2$ messages throughout between m and j. This is shown in Figure 6.1.

It is clear that $t,m \models L_k$ and, throughout between i and m the buffer has at least $d+1$ messages and at most $d+k$ messages. Hence $(t_i,t_{i+1},...,t_{m-1})$ has RW pattern of a k-1 buffer.

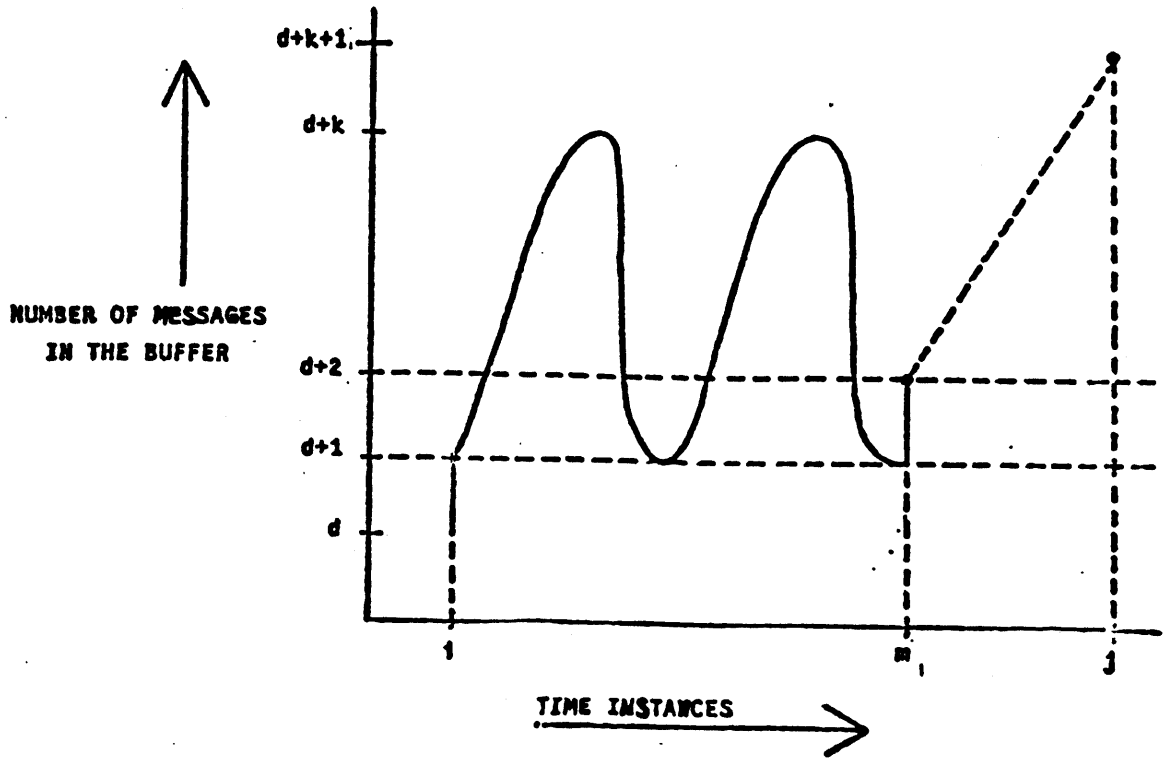


Figure 6-1:

Hence $L_{k+1} = W \wedge X(NF_{k-1}(L_k))$ for $k \geq 2$

$L_2 = W \wedge X(\neg R \, U \, W)$

$L_1 = W$

We also use the following formulae:

$NB_k(c)$: It is the dual of $NF_k(c)$. $t,i \models NB_k(c)$ iff $\exists\, j \leq i$ such that $t,j \models c$ and $(t_{j+1},\ldots,t_i)$ has RW pattern of a k buffer. $NB_k(c)$ speaks about the past while $NF_k(c)$ speaks about the future.

R_{k+1}: It is the dual of L_{k+1}. Let d be the number of messages in the buffer just after the j^{th} instance. Then $t,i \models R_{k+1}$ iff $\exists\, j < i$ such that the buffer has $d+k+1$ messages just before the j^{th} instance and at all instances between j and i-1 (including both) the buffer has at least $d+1$ messages.

Similar to L_{k+1}, R_{k+1} can be defined in terms of NB_{k-1} and R_k. L_k denotes that there is a future instance at which the buffer will have $d+k$ messages if it has d messages just

before the present instance. Similarly R_k denotes that there was an instance in the past when the buffer had d+k messages, and at present the buffer has d messages.

Now we define $NF_k(c)$ in terms of L_k, R_k and NF_{k-1}. Let $t,i \models NF_k(c)$ and let d be the number of messages before the i^{th} instance. Then $\exists\, j \geq i$ such that $t,j \models c$ and $(t_i,t_{i+1},\ldots,t_{j-1})$ has RW pattern of a k buffer. In this case one of the following two conditions holds.

(i) Between i and j, the buffer never has d+k messages in it, that is $(t_i,t_{i+1},\ldots,t_{j-1})$ has RW pattern of a k-1 buffer; i.e., $t,i \models NF_{k-1}(c)$.

(ii) Between i and j, the buffer has d+k messages in it at least once. In this case it is easily seen that between i and j, L_k,R_k occur alternately with L_k occurring first, R_k occurring last; between successive L_k and R_k, the sequence has RW pattern of a k-1 buffer; between i and the first L_k as well as between the last R_k and j the sequence has RW pattern of a k-1 buffer.

Thus $t,i \models ALT(L_k,R_k,NF_{k-1}(d),c)$.

Hence, $NF_k(c) = NF_{k-1}(c) \vee (ALT(L_k,R_k,NF_{k-1}(d),c)$

$NF_1(c) = Alt(W,R,True,c)$

It is to be observed that $NF_k(c)$, L_k are defined mutually inductively. Similarly we can define $NB_k(c)$.

Above we gave a formula $NF_k(c)$ to express the RW pattern of a k buffer up to an instance where c holds. We can extend this easily to express the behavior of a k buffer forever. For an infinite sequence to be the behavior of a FIFO or LIFO or unordered k buffer, it has to satisfy two properties, (a) the RW pattern denoted by the sequence should be that of a k buffer and (b) the messages read by the read operations should match with the

messages written by the write operations. Below we describe how to express the property (b) for FIFO, LIFO and unordered buffers of size k.

6.4.1. Expressing Bounded FIFO Buffers

We briefly sketch how we can express a FIFO buffer of size k. Let i,j be integers such that $j < i$ and m is the maximum integer so that $t,j \models L_m$, and there is no instance between j and i at which R_m is true. It can easily be seen from figure 6.1 that if the buffer has d messages before j, then it will have at least $d+1$ messages at every instance between j and i. Thus every instance like j increases the number of messages present in the buffer at the instance i. It can easily be shown that the number of messages in the buffer at the instance i is equal to the number of instances like j, present before i. Using this property we can obtain a formula f_c which is true at an instance iff the number of messages in the buffer just before that instance is c. We can also express that the message read by a read operation is the message written by the c^{th} previous write operation. Thus we can obtain a formula which expresses the correspondence between messages read from the buffer and messages written into the buffer.

6.4.2. Expressing Bounded LIFO Buffers

Let m be the maximum integer such that $t,i \models R_m$. Also let $j < i$ be such that $t,j \models L_m$ and at no other instance between j and i L_m holds. If $nb(j-1) = d$, then at every instance between j and i, the buffer has at least $d+1$ messages. Hence the message read at i is the message written at j. Since at every read operation R_m holds for some $0 \leq m \leq k$, we can easily obtain a formula which expresses the correspondence between messages read from the buffer and messages written into the buffer.

6.4.3. Expressing Bounded Unordered Buffers

In case of unordered buffers the message read can be any message present in the buffer at that instance. We express the above property as follows: For each $\sigma \in \Sigma$, we assert that the number of messages of value σ written into the buffer is always greater than or equal to the number of messages σ read from the buffer and that they differ by no more than k. This property can be expressed. It is easily seen that the above property together with the property (a) given above expresses an unorder buffer of size k.

All the formulae fb_k, ℓb_k, ub_k are in propositional linear temporal logic and are dependent on the message alphabet Σ. By making the following changes we can convert them into formulae in first order linear temporal logic that give domain independent characterizations of buffers of size k.

(i) Replace all R_σ by ((read-val = σ) $\wedge$ R) and W_σ by ((write-val = σ) $\wedge$ W)

(ii) Replace all $\bigwedge_\sigma$ (conjunctions over σ) by $\forall\sigma$.

It can easily be proved that the resulting formulae give domain independent characterizations of buffers of size k.

Below we show how we can characterize bounded message buffers more elegantly in QPTL(introduced in chapter 3) i.e. if quantification over propositions is allowed. We use only one level of existential quantification. A FIFO buffer of size 2 can be considered as two FIFO buffers each of size 1 in tandem as shown in figure 6.2 .

External writes come into the left buffer while external reads are from the right buffer. Whenever the left buffer is full and the right buffer is empty the message in the left buffer is internally read and is written into the right buffer. We consider this internal reading and writing to be occurring simultaneously and capture it by the propositions I_σ for $\sigma \in \Sigma$.

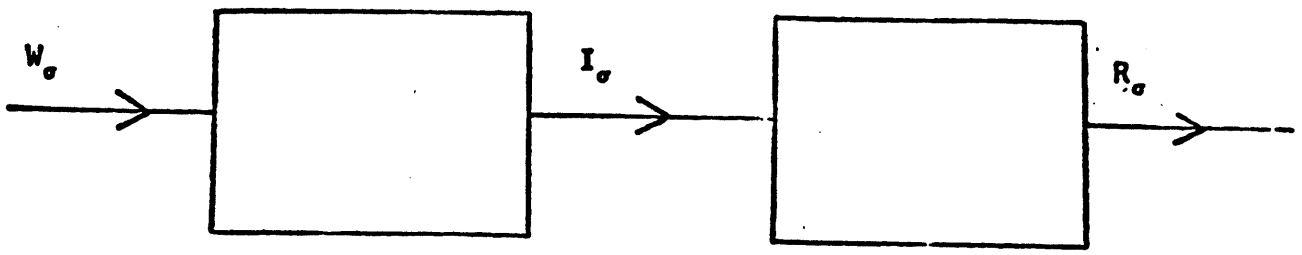


Figure 6-2:

Let $fb_1(\vec{W}, \vec{R})$ be the formula characterizing a buffer of size 1, where $\vec{W}$, $\vec{R}$ indicate vectors of propositions. The sequence of operations on the left buffer is characterized by $fb_1(\vec{W}, \vec{I})$, and the sequence of operations on the right buffer is characterized by $fb_1(\vec{I}, \vec{R})$. Let

$$fb_2 = \exists \vec{I} \{fb_1(\vec{W}, \vec{I}) \wedge fb_1(\vec{I}, \vec{R})\}$$

LEMMA 6.4: $s,0 \models fb_2$ *iff* $s \in FS_{\Sigma,2}$. □

For the general case of a buffer of size k we use a somewhat more complicated approach with k existentially quantified propositions $P_0, P_1, \ldots, P_k$. We will assert that P_j is true at an instance i iff the buffer has j messages before the operation of the i^{th} instance.

$$h' = G[\bigwedge_{0 \le \ell < m \le k} \neg(P_\ell \wedge P_m) \wedge \bigwedge_{0 \le \ell < k} ((P_\ell \wedge W) \supset X P_{\ell+1}) \wedge$$
$$\bigwedge_{0 < \ell \le k} ((P_\ell \wedge R) \supset X P_{\ell-1}) \wedge (P_0 \supset \neg R) \wedge (P_k \supset \neg W)] \wedge P_0$$

The first clause asserts that no more than one P_ℓ is true at any instance, the second clause asserts that if P_ℓ is true at an instance and the operation is a write operation then at the next instance $P_{\ell+1}$ is true, the third clause asserts similar property for read operation, the last two clauses assert that there are no writes on a full buffer and no reads on an empty buffer.

Let

$$fb_k = \exists \vec{P} \{I \wedge h' \wedge G(\bigwedge_{0<\ell\leq k} (P_\ell \supset \bigwedge_\sigma (R_\sigma \supset DC(\sigma,\ell))))\}$$

where $DC(\sigma,\ell)$ asserts that the ℓ^{th} previous write is the message σ. It is easily seen that fb_k characterizes FIFO buffers of size k.

THEOREM 6.5: $t,0 \models fb_k$ *iff* $t \in FS_{\Sigma,k}$. □

Let

$$\ell b_k = \exists \vec{P} \{I \wedge h' \wedge G(\bigwedge_{0<\ell\leq k} (P_\ell \supset \bigwedge_\sigma [R_\sigma \supset (\neg P_{\ell-1} S(W_\sigma \wedge P_{\ell-1}))])\}$$

The last clause asserts that the message read at any instance when the buffer has ℓ messages is same as the message written at the last instance when the buffer has ℓ-1 messages. The following theorem can be easily proved.

THEOREM 6.6: $t,0 \models \ell b_k$ *iff* $t \in LS_{\Sigma,k}$. □

Similarly we can obtain a formula for unordered buffers.

6.5. Characterizing Unbounded Buffers

Let $\mathcal{P}$ be a finite set of atomic propositions and $s = (s_0,s_1,...)$ be an infinite sequence of states where each state is a mapping from $\mathcal{P}$ into {True,False}. Let f be a formula in propositional temporal logic and SF(f) denote the set of subformulae of f. It is easily seen that card(SF(f)) ≤ length (f). For $i \geq 0$ let $[i]_{s,f} = \{g \in SF(f) \mid s,i \models g\}$.

LEMMA 6.7: *Let* $0 \leq i \leq j$ *be such that* $[i]_{s,f} = [j]_{s,f}$ *Then* $s,0 \models f$ *iff* $s',0 \models f$ *where* $s' = (s_0,s_1,...,s_i,s_{j+1},s_{j+2},...)$. □

THEOREM 6.8: *Unbounded message buffers (unordered, FIFO or LIFO) cannot be characterized in propositional linear temporal logic.*

The above theorem can be proved by a simple argument using the previous lemma. □

THEOREM 6.9: *There is no domain independent characterization of unbounded message buffers (unordered, FIFO or LIFO) in first order linear temporal logic.*

Proof. Suppose there is a formula f of type τ in first order temporal logic, which is a domain independent characterization of an unbounded buffer on models of type τ. Consider any model of type τ with finite domain. Then f characterizes unbounded message buffers in this model. Since the domain of this model is finite, we can replace all universal quantifiers by finite conjunctions, and by some other trivial changes we can obtain a formula f' in propositional temporal logic characterizing unbounded buffers over this domain. But this contradicts the Theorem 6.8. □

We have proved that it is impossible to give a *domain independent* characterization of unbounded message buffers. However, there are *partially interpreted temporal logics* in which unbounded message buffers can be characterized. Assume that there are two local variables write-history, read-history such that at any instance write-history contains the sequence of messages written into the buffer, while read-history contains the sequence of of messages read from the buffer. Then the following formula fb_∞ characterizes the behavior of an unbounded FIFO buffer:

$fb_\infty = g \wedge h$ where

g = G(read-history $\leq$ write-history),

h = G($\forall$ hist (hist = write-history $\supset$ F read-history = hist))

where $\leq$ is interpreted as the prefix relation, $\forall$ is interpreted as quantification over the set of all finite sequences of the message alphabet. 'g' asserts that the sequence of messages read from the buffer is a prefix of the sequence of messages written into the buffer; 'h' asserts that each message written into the buffer is ultimately read from the buffer. It can easily be shown that the above logic is undecidable.

6.6. Axiomatization of Message Buffers

Axiomatization of message buffers in PTL is a weaker notion than expressiveness. We show below that in general unbounded FIFO buffers are not axiomatizable. We also show that unbounded LIFO buffers and unbounded unordered buffers are axiomatizable though they are not characterizable in PTL.

THEOREM 6.10: *Bounded FIFO, LIFO and unordered buffers over any finite alphabet Σ are axiomatizable in PTL.*

Proof. Let fb_k be the formula in PTL characterizing the FIFO buffer of size k over a finite alphabet Σ.

Let fb_k = True S ($fb_k \wedge \neg$Y True). For any t and $i \geq 0$, t,i $\models \neg$Y(True) iff $i = 0$. Hence for any t and $i \geq 0$, t,i $\models fb_k$ iff t,0 $\models fb_k$; i.e., if $t \in FS_{\Sigma,k}$. Let A be any consistent and complete axiomatization for PTL. Then A $\cup$ $\{fb_k\}$ is a consistent and complete axiomatization for FIFO buffers of size k over Σ. Similarly we can give an axiomatization for bounded LIFO and unordered buffers over a finite alphabet. □

THEOREM 6.11: For any Σ such that card $\Sigma \geq 2$, the theory of unbounded FIFO buffers over Σ is not axiomatizable, and this set is Π_1^1-complete.

Proof. We prove below that for $\Sigma = \{0,1\}$ the theory of unbounded FIFO buffers is Π_1^1-complete. From this it automatically follows that this theory is not axiomatizable.

We first prove that the set of PTL formulae satisfiable over some model of an unbounded FIFO buffer over $\{0,1\}$ is Σ_1^1-complete. We consider a deterministic turing m/c on infinite strings with one read only infinite input tape and one work tape. This turing m/c works exactly like an ordinary turing m/c, but it takes infinite input strings and it never halts. It accepts an input string by going through the final state infinitely often. Let $M = (\Delta,Q,\delta,q_f)$ be such a turing m/c where Δ is the alphabet (including both input alphabet and tape alphabet), Q is the set of states, δ: $Q \times \Delta \times \Delta \rightarrow Q \times \Delta \times \{\text{left,right}\}$, q_f is the final state. After each step the input head of M moves right by one cell. If $\delta(q,\sigma_1,\sigma_2) = (q',\sigma'_2,\text{left})$, then whenever M is in state q and sees the symbols σ_1,σ_2 on the input and work tapes respectively, then M moves into state q', writes σ'_2 on the work tape and moves its head left, and it moves its input head right by one cell. We show below that given the encoding of M we can recursively obtain a formula f_M in PTL such that f_M is satisfiable on an unbounded FIFO buffer over $\{0,1\}$ iff M accepts at least one input.

Let $c = (Q \times \Delta) \cup \Delta$ be the set of composite symbols. A partial id of M, is a sequence of values from c, containing exactly one symbol from $(\sigma \times \Delta)$. A partial id denotes the contents of the work tape and the head position on the work tape and the state of finite control in the usual way. We like to assert that there is a ω-sequence of partial ids, so that each succeeding partial id is obtained from the previous partial id by one move of M for

some value of input character read by the input head, and there are infinitely many partial ids in this sequence containing a symbol of the form (q_f, σ). We call such a sequence an accepting sequence. Any such sequence denotes an accepting computation of M, and for every accepting computation of M there is such a sequence.

We fix a unary encoding of symbols from c using the character $0 \in \Sigma$. An encoding of a partial id is a sequence of encoding of the symbols in it separated by a single 1. An encoding of a sequence of partial ids, is the sequence of encodings of the partial ids separated by two consecutive 1's. In the following a symbol denotes the encoding of the value of the symbol. f_M asserts that there is an accepting sequence of partial ids of M as follows: An encoding of the initial partial id followed by two consecutive 1's is written into the buffer and during this period nothing is read from the buffer.

After writing of the initial id, reading and writing of symbols occurs alternately. Whenever a symbol is read, it is the symbol of the previous id. Each symbol written into the buffer is the value of the symbol in the new id assuming some input symbol on the input tape. f_M can express this because the value of a symbol in a new id depends only on the contents of that cell and its neighbors in the previous id, and the assumed value of input character. f_M makes sure that the assumed value of input character is the same throughout an id. f_M makes sure that two consecutive 1's are written at the end of each id. Finally f_M asserts that there are infinitely many places where a symbol of the form (q_f, σ) is written into the buffer. It is clearly seen that f_M is satisfiable on a model of an unbounded FIFO buffer over {0,1} iff M accepts at least one input.

Now we give a reduction in the other direction. Given any formula f in PTL we obtain a finite state automaton M'_f on infinite strings which accepts exactly those sequences of t

such that t,0 $\models$ f (each symbol in t is a mapping from the set of atomic propositions in f into {True,False}). From M'_f we obtain a TM M_f, which operates as follows. M_f takes each symbol in its input to be an encoding of a function assigning truth values to the set of atomic propositions. M_f simulates M'_f on the input, and at the same time it makes sure that the values of the propositions R_0,R_1,W_0,W_1 denotes a valid FIFO buffer behavior. M_f accepts an input iff M'_f accepts it and the input sequence denotes a valid FIFO buffer behavior. It is easily seen that M_f accepts at least one input iff f is satisfiable on a model of an unbounded FIFO buffer over {0,1}.

It can easily be shown that the set of encodings of TM's on infinite strings that accept at least one input, is Σ_1^1-complete. Hence the set of formulae in PTL that are satisfiable on a model of an unbounded FIFO buffer is Σ_1^1-complete. From this it follows that set of formulae, not satisfiable on any model of an unbounded FIFO buffer over {0,1} is Π_1^1 -complete. Hence the set of valid formulae is Π_1^1-complete. □

Let FS = $\bigcup_{k>1} FS_{\Sigma,k}$. Then the theory of finite FIFO message buffers is the set of all PTL formulae true in all interpretations over the models in FS. It can easily be shown that this theory is also not axiomatizable and that it is Π_1^0-complete.

Sometimes it is more realistic to consider models of FIFO buffers which are recursive; i.e., models for which we can recursively determine the truth value of an atomic proposition at any point on the model. For this case also, it can be shown that the theory of these models is Π_3^0- complete.

The degenerate case in which the message alphabet has a single element is also interesting since it corresponds to processes that communicate using signals.

THEOREM 6.12: *The theory of unbounded FIFO buffers over Σ has a single element, is decidable.*

Proof. We say that an infinite sequence of states t is ultimately periodic with starting index ℓ and period p if $\forall\, i \geq \ell\ t_i = t_{i+p}$. We can easily prove that a formula f is satisfiable on a $t \in FS_{\Sigma,\infty}$ iff there exists a $t' \in FS_{\Sigma,\infty}$ such that t' is ultimately periodic with starting index $2^{c.|f|}$ and period $2^{c.|f|}$ for some constant c and f is satisfiable on t'. From this we can easily get a decision procedure for satisfiability of f in $FS_{\Sigma,\infty}$ is decidable. Indeed we can get a decision procedure that uses space polynomial in the length of the input. □

THEOREM 6.13: *The theory of unbounded LIFO buffers over a finite alphabet is decidable.*

Proof. For each formula f in PTL we can obtain a finite state automaton M_f on infinite strings such that M_f accepts exactly those sequences t such that $t,0 \models f$ (each state in t is a mapping from the set of atomic proposition appearing in f into {True,False}). From M_f we can obtain a push down automata P_f operating on infinite strings. P_f uses its stack to make sure that the sequence of read/write operations represented by the input string is a legal series of read/write operations on the buffer, while at the same time the finite state control of P_f makes state transitions exactly as M_f . P_f accepts an infinite string iff its finite state control goes through any of a set of final states infinitely often. P_f accepts an input t iff $t \in LS_{\Sigma,\infty}$ and $t,0 \models f$. Thus f is satisfiable on a $t \in LS_{\Sigma,\infty}$ iff P_f accepts some input. The latter problem can be reduced to a finite set of questions regarding whether an ordinary push down automaton (on finite strings) accepts any string. Hence the problem of satisfiability of a formula on a sequence in $LS_{\Sigma,\infty}$ is decidable. □

THEOREM 6.14: *For a finite* Σ, *satisfiability of a formula on a model of an unbounded unordered message buffer over* Σ *is decidable iff reachability problem for vector addition systems with states of dimension* card(Σ) is decidable.

Proof. Let G = (V,E,L) be a vector addition system with states of dimension k, where (V,E) is a directed graph, and L: $E \rightarrow N^k$ where N is the set of integers. A configuration is a pair (s,a) where $s \in V$, $a \in N^k$. We say that a configuration (t,b) is reachable from (s,a) iff there exists a sequence of configurations (called a path) (s_1,a_1), (s_2,a_2),...,(s_n,a_n) such that $(s_1,a_1) = (s,a)$, $(s_n,a_n) = (t,b)$, and $\forall i\ 1 \leq i \leq n\ \forall j\ 1 \leq j \leq k\ a_{ij} \geq 0$ and for $i < n$ $(s_i,s_{i+1}) \in E$ and $a_{i+1} = a_i + L(s_i,s_{i+1})$. Let k = card($\Sigma$).

We reduce reachability problem to satisfiability problem. G is a vector addition system as given above and it is required to determine if (t,b) is reachable from (s,a). Let $\Sigma = \{\sigma_1,\sigma_2,...,\sigma_k\}$. We give a formula f such that for some $t \in US_{\sigma,\infty}$, $t,0 \models f$ iff (t,b) is reachable from (s,a). We use propositions P_u for each $u \in V$. f asserts the following properties:

(i) For each i, $1 \leq i \leq k$ initially a_i number of messages of value σ_i are written into the buffer; immediately after this P_s is true.

(ii) The propositions P_u (for $u \in V$) are mutually exclusive. For $u \neq t$ if P_u is true at any instance i then the next proposition to be true in future at instance j will be P_v where $(u,v) \in E$, and if $(c_1,c_2, ...c_k) = L(u,v)$ then between i and j, $\forall \ell\ 1 \leq \ell \leq k$ if c_ℓ is positive (negative) then $|c_\ell|$ number of messages of value σ_ℓ are written into (read from) the buffer.

(iii) If P_t is true at any instance, either (ii) holds or the following condition is satisfied. Immediately after P_t is true, $\forall \ell\ 1 \leq \ell \leq k$ b_ℓ number of messages of value σ_ℓ are read from the buffer, and after this all propositions are false forever.

(iv) There is a future instance from where all propositions will be false forever.

Since we required that for any $t \in US_{\Sigma,\infty}$, infinitely often the buffer should be empty it easily follows that $t,0 \models f$ iff (t,b) is reachable from (s,a) in the above vector addition system. Assume reachability problem is decidable.

Let f be a formula. We wish to determine if $\exists\, t \in US_{\Sigma,\infty}$ such that $t,0 \models f$. From f we can easily obtain a finite state automaton on infinite strings M_f, which accepts by going through a final state infinitely often, and such that M_f accepts exactly the set of strings t such that $t,0 \models f$. From M_f we can obtain a vector addition system $G = (V,E,L)$, in which V is the set of states of M_f, $(s_1,s_2) \in E$ iff there is an α (α is a tuple denoting a function that assigns truth values to propositions) such that there is a transition in M_f from s_1 to s_2 on input α, and $L(s_1,s_2) = (a_1,a_2,...,a_k)$ where $\forall i\ 1 \le i \le k$,

$a_i = 1$ if w_{σ_i} is true in α,

$a_i = -1$ if R_{σ_i} is true in α

$a_i = 0$ otherwise.

Let q_I, q_f be the initial and final states of M_f, and $\bar{0} = (0,0,...0)$.

The following is easily seen:

There is a $t \in US_{\Sigma,\infty}$ such that $t,0 \models f$ iff there is a $q \in V$ such that

(i) $(q,\bar{0})$ is reachable from $(q_I,\bar{0})$ in G and,

(ii) $(q,\bar{0})$ is reachable from $(q,\bar{0})$ by passing through q_f

(ii) is not a direct reachability problem; however, we can put it as a reachability

problem as follows: Introduce another copy of G, call it G', and introduce a transition from q_f in G to q'_f in G', which is labelled with $\bar{0}$. Now (ii) is satisfied in G iff $(q',\bar{0})$ is reachable from $(q,\bar{0})$ in the new vector addition system.

Since we assumed reachability is decidable, we can easily decide if there is a q satisfying (i) and (ii). □

6.7. Conclusion

We have examined the possibility of using linear temporal logic to express the semantics of different message buffering systems. We have shown that it is possible to characterize bounded message buffers but not unbounded ones. We have also considered axiomatization of the theory of various message buffer systems; unbounded FIFO buffers are, in general, not axiomatizable, while unbounded LIFO and unordered buffers are axiomatizable.

Chapter 7

Distributed Implementation Of CSP

7.1. Introduction

Communicating Sequential Processes (CSP) was introduced in [Ho78] as an appropriate Programming Language for Distributed Systems. The original semantics of CSP did not require *fairness* in the selection of processes waiting to establish communication. However, in practice such a restriction may be highly desirable. In this chapter we introduce a formal model for CSP implementations and prove simple lower bounds on the time complexity for establishing fair communication. We also present a number of new algorithms that ensure different fairness properties and are near optimal in special cases.

The processes in the CSP language do not share global memory, but instead communicate by message passing primitives using the following syntax:

P ? x (input message from process P into variable x),

P ! x (output x to process P).

Communication occurs when one process names another as destination for output, and the second process names the first as the source for input. In this case, the value to be output is copied from the first process to the second. This type of synchronization is called a *rendezvous* and is used as the basis for synchronization mechanism in the ADA language also. Note that there is no automatic buffering of messages that have been sent but not

received; therefore, a process executing an output command will be delayed until the destination process is ready to receive and vice versa.

Much of the power and elegance of CSP comes from the fact that input and output statements can occur within the guards of guarded commands. For example, a server process which is willing to receive requests from any of two user processes executes a command of the following form:

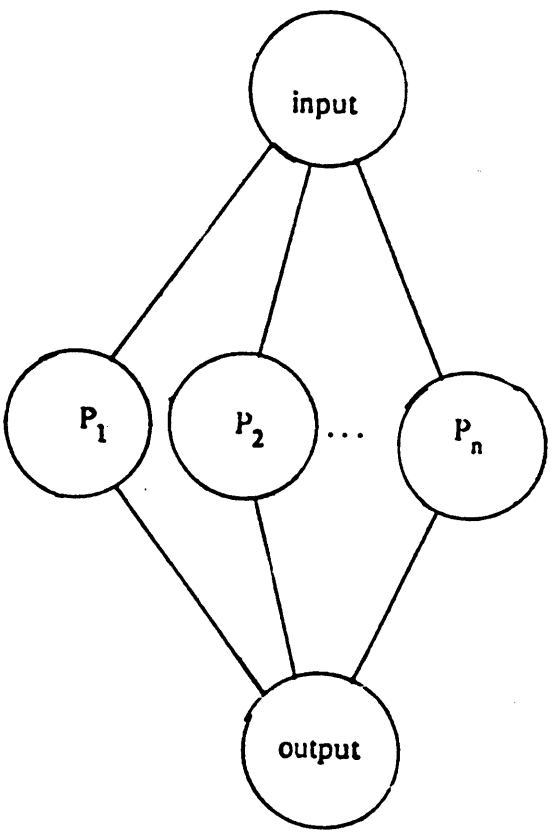
```
[
 user1 ? request → ...
   □
 user2 ? request → ...
]
```

In this case whenever the server process enters the above alternative command, it waits until either $user_1$, or $user_2$ is ready to send a request and accepts a request from one of them. After this it executes the statements following the corresponding guard. If both the user processes are ready to send a request then it chooses one of them arbitrarily. It is possible to have a sequence of boolean expressions in front of a guard with at most one i/o command. In this case, whenever the server process enters an alternative command then all the boolean expressions in a guard are evaluated, if any of them is false then the guard fails and the corresponding communication is not allowed. A failed guard is ignored. Thus we see that, in general the set of communications that are enabled depends on the state of the server process and so is dynamic. The following is an example where output statements in guards are convenient. There are two identical server processes P_1, P_2. Each user whenever it needs a service executes the following alternative command.

```
[
 P1 ! request → ...
   □
 P2 ! request → ...
]
```

The semantics of the above command is the same as in the previous case. Thus we see that a particular user does not have to wait for a particular server. The same alternative command can have both input and output guards.

Fairness was not required in the original semantics of CSP. However fairness in communication is highly desirable in many cases. Consider the following example:



The input process reads a sequence of transactions from a terminal. It sends all transactions of type i to the transaction processor P_i. The output process periodically executes an alternative command where in it is ready to receive processed transactions from any of the transaction processors. The output process prints the processed transactions. Each transaction processor P_i periodically executes an alternative command where in it is ready to receive the next input transaction, or is ready to send an already processed output transaction to the output process. In this problem we require that whenever P_i wants to send a processed transaction then the output process should eventually receive it. Thus fairness in communication is a requirement here. We assume that each CSP process is non-terminating and periodically executes an alternative command with i/o guards.

We introduce a formal model for this and consider two different fairness properties; *weak fairness* and *strong fairness.* For example: in weak fairness we require that computations in which two processes are *willing* to communicate with each other throughout the future but in which the two processes never establish communication, should not be possible. We consider algorithms for distributed schedulers to ensure the different fairness properties. In this model neighboring schedulers can talk to each other using shared variables. We define the *time complexity* for ensuring the fairness property, as in [Ly80]. We give simple *global algorithms* for ensuring the fairness properties. In these the scheduler processes are able to send arbitrary information to one another. Next we consider algorithms in which *interaction* between neighboring schedulers is restricted (i.e. a scheduler can request another for communication and the other scheduler can grant or deny the request). For these algorithms we prove an $O(\gamma^2)$ lower bound on the time complexity for ensuring weak fairness where γ is the chromatic number of the *communication graph.* We present a near optimal algorithm for the case when the communication graph is a complete

graph. We consider algorithms in which the interaction between neighboring schedulers is improved (in addition to the previous interaction a scheduler can withdraw a request). In this model we present better algorithms for weak fairness, and for strong fairness.

In [Be80], [BS83], [Si79] algorithms for distributed implementation of CSP were considered. but their notion of fairness (if any) is weaker than ours. An algorithm for weak fairness is presented in [Sc], but no lower bounds are presented, and our new algorithms have better complexity. Some probabilistic algorithms are presented in [Sp81], [RS81], for weak fairness. However the model used in these is slightly more general than our model.

This chapter is organized as follows: In section 2 we introduce the formal model, define the weak/strong fairness properties and the complexity of implementing fairness. In section 3 we present simple global algorithms for ensuring the fairness properties. In section 4 we consider algorithms with restricted interaction. In section 5 we consider algorithms that permit more interaction between the scheduler processes.

7.2. Formal Model and Definitions

7.2.1. Notation

A *Distributed Synchronization System* (DSS) is a triple (G, P, Var) where G = (V,E) is an undirected graph, called the *Communication Graph.* Each node in V denotes a CSP process and each edge denotes a possible communication between a pair of CSP processes. Note that this graph can be syntactically determined from any given CSP program. P: V $\rightarrow$ **Programs** is a function that associates a *scheduler process* with each node in V. We denote the nodes in V by integers, and the scheduler process P(i) by P_i. Var is the set of variables which consists of *local variables* and *shared variables.* The local variables can only be accessed by one process.

A process is a set of *actions*. Each action is of the form: (s_{old}, v,$value_{old}$,$value_{new}$,s_{new}). This action denotes that whenever the process is in state s_{old} and the value of v is $value_{old}$ then it changes the value of v to $value_{new}$ and goes into the new state s_{new} in one operation. The states of a process are drawn from an infinite set of states. We require that each process should be *deterministic* i.e. in every state, all the actions of the process in that state should be on the same variable and there should be exactly one action in a given state for each possible present value of the variable. The only local variables present at node i are c_{ij}, $port_{ij}$ corresponding to each $\{i,j\} \in E$. These variables take values 0 or 1. Intuitively $c_{ij} = 1$ indicates that the CSP process at node i is willing to communicate with the one at node j. If $port_{ij} = port_{ji} = 1$ then it indicates that the CSP processes at node i,j are in communication. We require that each P_i should satisfy the following constraints:

1. c_{ij} can only be accessed by P_i, and it is a read only variable for P_i i.e, the old and new values of c_{ij} in any action of P_i should be identical;

2. The only update by P_i on the variable $port_{ij}$ is from 0 to 1.

Each shared variable can be accessed by only one pair of adjacent processes i.e. a variable v is accessible by only two processes P_i, P_j such that $\{i,j\} \in E$. Only one process can update the variable, but it can be read by both.

We assume that the shared variables are implemented as follows: Consider a logical shared variable v that is shared between two adjacent processes P_i, P_j. Assume that P_i can update v while P_j can only read this variable. There are two actual variables v', v" corresponding to this v. v' is at the node i, while v" is at node j. In addition to the scheduler processes, there are two channel processes M_{ij}, M_{ji} corresponding to each edge { i,j } in G. Whenever P_i updates v', M_{ij} reads this value in one action and writes this value into v" in another action.

Let **Act-var** = $\{ c_{ij}, port_{ij} \mid \{i,j\} \in E \} \cup \{ v', v'' \mid$ for each shared variable v $\}$,

Processes = $\{ P_i \mid i \in V \} \cup \{ M_{ij}, M_{ji} \mid \{i,j\} \in E \}$

A configuration (or id) of a DSS is pair of the form (S, Val) where S: **Processes** → States, Val: **Act-var** → **Values** where S(p) denotes the state of process p, Val(v) denotes the value of the variable v and should be from the appropriate domain. A computation $\mathbb{C}$ is an ω-sequence of ids $id_0, id_1, id_2, \ldots, id_n, id_{n+1}, \ldots$ that satisfies the following criterion:

1. id_0 is the initial id.

2. id_{n+1} is obtained from id_n by one of the following moves:

(a) Process moves - the transition from id_n to id_{n+1} may involve one action of more than one process, but the actions of different processes should be on different variables. These processes moves also include the moves of the channel processes.

(b) Oracle moves - An oracle move occurs simultaneously at nodes i,j if $port_{ij} = port_{ji} = 1$ in id_n. The oracle move resets $port_{ij}$,$port_{ji}$ to 0 ; and it gives arbitrary new values to the the variables in the set $\{c_{ik} \mid \{i,k\} \in E\} \cup \{c_{jk} \mid \{j,k\} \in E\}$. An oracle move models the completion of communication between the CSP processes at i,j and the reevaluation of the guards by both of them. By this we are assuming that the computation by the CSP processes between successive entries into an alternative command takes zero time.

In the initial id, the values of the variables $port_{ij}$ is 0. A DSS may have a restricted set of initial ids.

7.2.2. Correctness and Fairness

The correctness criterion is that in any computation if two processes are in communication at any instant then both of them should be willing to communicate with each other at that instant and at any instant a process can communicate with at most one other process. This is captured by the following invariance requirement:

For every $\{i,j\} \in \mathbf{E}$

$$port_{ij} \supset (c_{ij} = 1) \wedge (c_{ji} = 1) \wedge \bigwedge_{k \neq j} (\neg port_{ki} \wedge \neg port_{ik}) \bigwedge_{k \neq i} (\neg port_{kj} \wedge \neg port_{jk})$$

A DSS is said to satisfy *weak fairness* iff the following type of computation is not allowed by it.

$id_0, id_i, \ldots, id_n, \ldots, id_\ell, \ldots$

In the above computation $c_{ij} = c_{ji} = 1$ in every id after id_n, but in no id after id_n i,j establish communication ; that is in no id after id_n $port_{ij} = port_{ji} = 1$. Intuitively weak fairness requires that if two processes are willing to communicate with each other for sufficiently long time then they eventually establish communication.

A DSS is said to satisfy *strong fairness* iff the following type of computations are avoided by it.

$id_0, id_1, \ldots, id_n, \ldots, id_\ell, \ldots$

In the above computation, in every id after id_n $c_{ij} = 1$, and infinitely often $c_{ji} = 1$ but i,j never establish communication after id_n. The above definition is symmetrical with respect to i,j. It is to be observed that any DSS that ensures strong fairness also ensures weak fairness.

7.2.3. Complexity

A timed computation is a pair ($\mathcal{C}$, t) where $\mathcal{C}$ is a computation and t: N → **Reals** (where N is the set of positive integers) such that t is positive and monotonic i.e. $t(i) \geq 0$ and if $i > j$ then $t(i) > t(j)$. Intuitively, t gives timing to each step of the computation.

We use the following time parameters:

T_c - upper bound on communication time,

T_m - upper bound on message passing time,

T_p - upper bound on process step time.

We say that a timed computation ($\mathcal{C}$,t) obeys the above time parameters iff it satisfies the following conditions:

For all $m > \ell \geq 0$, $i,j \geq 0$

1. If $port_{ij} = port_{ji} = 1$ in every id_k such that $\ell \leq k < m$ then $t(m) - t(\ell) \leq T_c$;

2. Let v',v" be the two images of a shared variable v that can be updated by P_i and read by P_j. Assume id_ℓ is obtained by a move of the channel process M_{ij} which read the variable v', and let id_m be the id obtained by an action of M_{ij} that wrote into v", the previously read value. Then $t(m) - t(\ell) \leq T_m$;

3. If id_ℓ is obtained by one move of P_i, and id_m is the earliest next id obtained by a move of P_i, then $t(m) - t(\ell) \leq T_p$.

The meaning of the above time parameters is intuitively obvious. Note that we are only considering upper bounds, thus allowing each process to be arbitrarily faster than the other. From here onwards we fix the above time parameters and only consider timed

computations that obey the above parameters. We say that μ is the complexity of a DSS D iff μ is the minimum value satisfying the following condition:

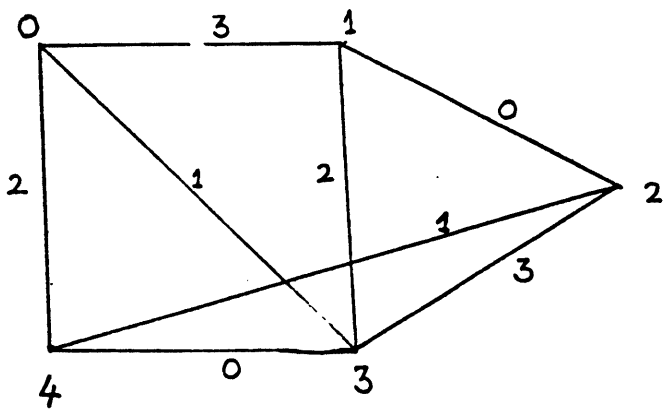
In every timed computation (C,t) of D and for every $\{i, j\} \in E$, if ℓ,m are two instances such that $t(m) - t(\ell) \geq \mu$ and in every id between id_ℓ, id_m $c_{ij} = c_{ji} = 1$, then i, j establish communication in some id between id_ℓ,id_m.

We let $\mu(D)$ denote the complexity of D. Clearly it is a function of the time parameters T_c,T_m,T_p. We also assume that T_p is negligible compared to T_c,T_m.

7.3. Global Algorithms

We present a simple global algorithm that ensures weak fairness. Let d be the maximum degree of any node in G. We color the edges of G with h colors i.e.for each edge e we associate a color c(e) such that if e_1,e_2 are edges incident on a vertex then $c(e_1) \neq c(e_2)$. From graph theory [Be73], it is known that there is a coloring such that the number of colors $h \leq (d+1)$. Let the colors be drawn from the set $\{0,1,\ldots, h-1\}$.

Ex:



We designate a scheduler at a particular node as the controlling scheduler. The whole

algorithm proceeds in successive rounds. At the beginning of the ℓ^{th} round, the controlling scheduler sends messages to all other schedulers denoting the beginning of the round. Let k $= \ell$ **mod** h. When P_i receives this message it performs the following procedure: If there is an edge $e=\{i,j\}$ with color k, then P_i, P_j talk to one another and if both are willing to communicate then they establish communication. Otherwise P_i sends an acknowledgement to the controlling scheduler. If communication along e is established then once the communication is complete then both P_i,P_j send acknowledgements to the controlling scheduler. When all the acknowledgements are received the controlling scheduler starts the next round.

In the above algorithm, we can use some protocol to merge acknowledgements at each node. We can use echo algorithms as in [Ch] to send acknowledgements. The details are easy to work out. It is easily seen that this algorithm ensures weak fairness. A careful analysis shows that the complexity of the above algorithm $= h \cdot T_c + k \cdot D \cdot d^2 \cdot T_m$, where D is the diameter of G and k is a constant. The above algorithm does not guarantee strong fairness. However we can easily obtain a similar algorithm which uses dynamic priorities on edges, that ensures strong fairness.

We can indeed easily obtain many different such global algorithms that use a central scheduler. All these algorithms have the following disadvantages:

- Reliability will be poor because these algorithms are controlled by a centralized scheduler.

- If at any instance the set of (edges connecting neighboring processes which are willing to communicate with each other), is sparse then these algorithms take too long time.

- The complexity of these algorithms is dependent on the diameter of the graph which is an undesirable feature.

7.4. Local Algorithms with restricted interaction.

7.4.1. Lower bounds for weak fairness

We consider a class of algorithms in which the interaction between neighboring processes is limited. For each edge {i,j}, there are two shared variables v_{ij},v_{ji}. v_{ij} can be updated by P_i, while v_{ji} can be updated by P_j. These variables take the following values:

R - Request

G - Granted

D - Denied

N - Null

We restrict the interaction between neighboring processes as follows:

P_i sets v_{ij} to "R" ;

P_j sets v_{ji} to "G" or "D"; /* After seeing the request of P_i*/

P_i sets v_{ij} to "N"; /* After seeing the answer of P_j */

P_j sets v_{ji} to "N" /* After seeing P_i reset it's variable */

We require that P_i set v_{ij} to "R" only when $c_{ij} = 1$ i.e. when i is willing to communicate with j. Also when P_j sets v_{ji} to "G" then it is committed for communication i.e. in the next action it sets $port_{ji}$ to 1. If P_j sets v_{ji} to "D" then it does not commit to communicate until

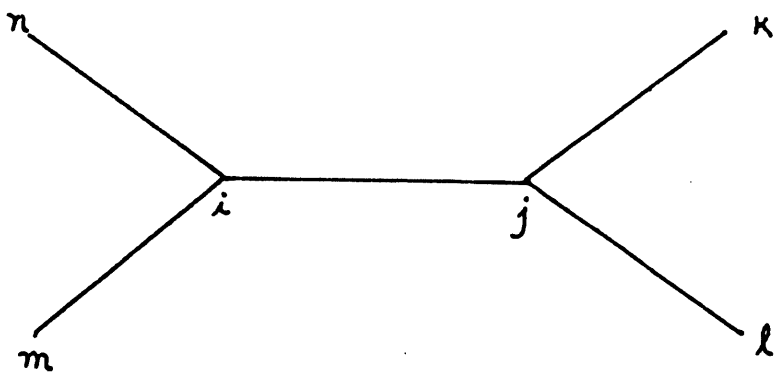
the next interaction. If both P_i, P_j set their respective variables to "R" at the same time then they both should commit to communicate immediately. All these restrictions can be defined more formally.

It can easily be shown that with the above restricted interaction processes cannot pass any information between them.

THEOREM 7.1: *If* D *is any* DSS *with restricted interaction that ensures weak fairness then* $\mu(D) \geq (\gamma(\gamma-1)/2)\cdot(T_c + 2T_m)$ *where* γ *is the chromatic number of* G.

Proof Sketch:

Consider two nodes i,j in the communication graph as shown below:



We say that i *immediately waits on* j in the configuration ID_t iff j is in communication with one of its neighbors other than i and in all computations from ID_t in which $c_{ij} = c_{ji} = 1$ throughout, i does not communicate with any other node until it established communication with j.

Claim 1: There exists computations in which i immediately waits on j infinitely often or j immediately waits on i infinitely often.

Proof: Assume the contrary. We can generate computations in which $c_{ij} = c_{ji} = 1$ throughout and the following happens: Whenever P_i requests P_j for communications at that instance j is in communication with one of its neighbors. Since i does not immediately wait on j, after some time it establishes communication with a different neighbor (other than j). And a similar situation occurs whenever P_j requests P_i. Thus i,j never establish communication, though each of them is willing to communicate with the other throughout the future. This violates weak fairness requirement. □

One of the protocols by which i immediately waits on j is that P_i sets v_{ij} to "R" and waits until P_j replies. Because of the restricted interaction, the only other protocols are trivial modifications (ex. P_i and P_j have at most a fixed number of interactions in which in all interactions except the last one P_j denies the request, and in the last interaction P_j may grant the request). We say that i *waits on* j in ID_t iff j immediately waits on another of it's neighbors in ID_t and in all computations starting in ID_t in which through out the computation $c_{ij} = c_{ji} = 1$, i does not establish communication with any other process until it establishes communication with j.

Claim 2: There exists computations in which i waits on j infinitely often or vice versa.

Proof: Assume the contrary. From Claim 1, assume j immediately waits on some neighbor k infinitely often in some computation and so does i immediately wait on some neighbor n in some other computation. Because of the asynchrony we can prove that there is a computation in which j immediately waits on k and i immediately waits on n infinitely often. Now we can generate a computation as follows. Whenever P_i requests P_j then j is immediately waiting on k. At these instances P_j cannot come out of its waiting phase on P_k and establish communication with i (ex: in the simple protocol P_j sets v_{jk} to "R" and waits

for the reply of P_k which may grant the request. In this case if P_j withdraws the request, then the sequence of events may be as follows. P_k sees the request of P_j, after this P_j withdraws the request and establishes communication with i, then P_k commits for communication with j by granting its request. This clearly causes incorrect computation.) Hence from our hypothesis P_j always rejects the request of P_i. Similarly whenever P_j requests P_i at that instance i is immediately waiting on n and this request gets rejected. Thus i,j never establish communication. This violates weak fairness requirements. □

By similar arguments we can show that dynamic arbitrary long waiting chains form. Clearly these waiting chains have to be acyclic, otherwise deadlocks occur. Because the processes cannot pass information around, the only way to avoid deadlocks is as follows. There should be an a priori defined RW relation as follows: For each edge { i,j }; i RW j or j RW i. i waits on j only if i RW j. Also there should not be any cycles of length greater than 2 in the RW relation.

Let the RW-graph be the directed graph corresponding to the RW relation. Let $e_0, e_1, \ldots, e_k$ be a sequence of edges along a path in the RW-graph. Consider a computation in which a waiting chain along this path forms and communication along these edges occur in sequence. Thus by the time communication along e_0 is established, it will have to wait for k communications to occur in sequence and this takes $k \cdot (T_c + 2\,T_m)$ time period.

Let the RW'-graph be a directed graph obtained from RW-graph by keeping, only one directed edge for every cycle of length 2. Clearly the RW'-graph is acyclic and is obtained by directing all the edges of G in one direction. Let i be any node which has directed edges leaving it to the vertices; $j_0, j_1, \ldots, j_k$. Now consider a computation in which $c_{ij_\ell} = c_{j_\ell i} = 1$ (for $0 \leq \ell \leq k$) throughout the computation. i can only wait along one edge at a time. This

can easily be seen , otherwise correctness will be violated. Let $e_\ell = (i,j_\ell)$ (for $0 \le \ell \le k$). Now i has to wait on one of the edges last before it waits on other edges at least once. Without loss of generality let e_k be this edge. When i waits along e_0, a waiting chain along the longest path starting with e_0 can form. Let m_ℓ be the length of the maximum path along e_ℓ. Thus when i waits along e_ℓ it takes at least $m_\ell \cdot (T_c + 2\,T_m)$ time before it completes the communication along e_ℓ. Hence by the time communication along e_k is established it takes at least time $\sum_{0 \le \ell \le k} m_\ell \cdot (T_c + 2T_m)$.

Using simple graph theory [Be73] we can easily prove the following claim.

Claim 3: IF G' is an acyclic directed graph obtained by directing the edges of G, then there is a vertex i in G' such that there are $(\gamma\text{-}1)$ outgoing edges from i (say $e_0, e_1, \ldots, e_{(\gamma-1)}$) such that the length of the maximum path along e_ℓ is $(\gamma\text{-}1\text{-}\ell)$. □

From claim 3, it follows that there is a computation such that to establish communication along one of the edges it takes time $\ge \sum_{0 \le \ell \le (\gamma-1)} (\gamma\text{-}\ell\text{-}1) \cdot (T_c + 2T_m) =$ $(\gamma(\gamma\text{-}1)/2) \cdot (T_c + 2T_m)$.

Hence we see that $\mu(D) \ge (\gamma(\gamma\text{-}1)/2) \cdot (T_c + 2T_m)$. □

The lower bound given by theorem 7.1 may not be significant for some cases (where γ is very small). An obvious lower bound is $d \cdot (T_c + 2T_m)$. In these cases this may be the better lower bound.

7.4.2. Algorithms for weak fairness

The following algorithm was proposed in [Sc] for ensuring weak fairness:

Let c: $V \rightarrow \{ 0,1, \ldots, (h-1)\}$ be a vertex coloring of G using h colors, i.e. for each edge $\{i, j\}$ in G, $c(i) \neq c(j)$. The RW relation is defined as follows: i RW j iff $\{i, j\}$ is an edge in G and $c(i) < c(j)$. Clearly the RW relation is acyclic.

The protocol at node i

OUT: Array of all j such that i RW j;

IN: Array of all j such that j RW i;

D_o: Out degree of node i, in the RW-graph;

D_i: In degree of node i, in the RW-graph;

Current: Variable used to point into the array OUT;

```
current : = 0;
Loop Forever
    flag : = TRUE;  pointer : = current; found : = FALSE;
    While flag
        j : = OUT [pointer];
        If c_ij then
            current : = pointer,
            flag : = FALSE,  found : = TRUE
        else
            pointer: = (pointer + 1) mod D_o
            If pointer = current then flag : = FALSE
    end of while

    If ¬found then go to Answering-Phase;
    j : = OUT[current];
    v_ij : = "R";
    await  v_ji ≠ "N";
```

Temp := v_{ji} ;

v_{ij} := "N";

Await v_{ji} = "N";

If temp = "G" **then** establish communication with j;

Answering - Phase:

For ℓ := 1 **Step** 1 **until** D_i **do**

j := IN [ℓ];

If v_{ji} = "R" **then**

if c_{ij} = 0 **then** answer negatively to j

Else

begin

v_{ij} := "G";

establish communication with j

end

End for

End main for loop

The above algorithm works as follows: P_i alternates between an asking phase and an answering phase. During an asking phase it requests along an outgoing edge. It waits until this request is answered. If the request is granted then it establishes communication along this edge. After this it goes into answering phase. In this phase it goes through all waiting requests in some order. During this phase if it is willing to communicate on an in coming edge it grants the request and establishes communication along this edge, otherwise it denies the request. A proof of correctness of the above algorithm is presented in [Sc]. It is also shown that the above algorithm has complexity

$$\mu \leq h \cdot d^2 \cdot (T_c + 2T_m).$$

If we use minimum coloring then $h = \gamma \leq (d+1)$. Hence $\mu \leq d^2(d+1) \cdot (T_c + 2T_m)$.

Thus the complexity of the above algorithm depends only on the degree d, and is independent of the size of the graph. We say that an algorithm is *real time* iff the complexity of the algorithm is a function of the maximum degree d and is independent of the size of the graph.

For complete graphs we give an algorithm that is near optimal. Let G be a complete graph. Define the RW relation as follows: (i RW j) iff $i < j$. We modify the answering phase of the previous algorithm as follows:

We initialize the IN array at node i as follows. If $k < j < i$ then j appears before k in the array IN.

Answering-Phase at node i:

For $\ell := 0$ **step** 1 **until** $\ell = D_i$ **do** FLAG[ℓ]: = FALSE **od**;

ind : = TRUE;

while Ind **do**

t : = FALSE;

For ℓ: = 0 **Step** 1 **until** D_i **do**

j : = IN[ℓ];

case

v_{ji} = "R" $\wedge$ c_{ij} = 1 $\wedge$ $\neg$ FLAG[ℓ]:

Establish communication with j;

FLAG [ℓ] : = TRUE;

t : = TRUE;

v_{ji} = "R" $\wedge$ c_{ij} = 0 $\wedge$ $\neg$FLAG[ℓ]:

Answer negatively to j;

v_{ji} = "R" $\wedge$ FLAG[ℓ]:

Answer negatively to j;

$v_{ji} \neq$ "R": ;

End case

End for

If $\neg$t **then** Ind : = FALSE;

End of while

The answering phase given above works as follows: In the 'for' loop P_i makes a sweep

of waiting requests in the order of the IN array. During a sweep it grants a request along an edge iff it is willing to communicate and has not established communication along this edge previously in the current answering phase. It continues these sweeps until after a sweep in which it has not established communication along any edge. It is easily seen that P_i makes at most $(D_i + 1)$ sweeps and establishes communication at most once along an in coming edge, during an answering phase. Thus we see that the time taken for an answering phase $\leq$ $D_i \cdot T + k D_i (D_i + 1) \cdot T_p$ where $T = (T_c + 2T_m)$. Since T_p is much smaller than the other time parameters,we approximate the above time to $(D_i + 1) \cdot T = i \cdot T$(since G is a complete graph $D_i = (i-1)$).

Consider a computation in which $c_{ij} = c_{ji} = 1$ throughout. We say that a request from i to j is successful if the request is granted. Let t_{ij} denote the time between the instances when i made a successful request and j granted the request. If the request of i to j is made when j is in the answering phase then the request will be answered with in time $j \cdot T$. However if the request of i is made while j is waiting on it's request to k then i's request will be answered within time $(t_{jk} + (j-i) \cdot T)$ (this is because there at most (j-i-1) items before i in the IN array at node j). The following recurrence inequality can easily be seen.

$$t_{ij} \leq \max \{ j \cdot T, [(j-i) \cdot T + \max_{k<j}(t_{jk})] \}$$

where $T = (T_c + 2T_m)$.

By induction we can easily show that $t_{ij} \leq (2n-i) \cdot T$. A request will be rejected if it occurs in the same answering phase of j as the previous one. Thus there may be several requests before it is granted. However in this case the difference in time between the first and last rejected requests is bounded by the time taken by j to complete one answering phase and this is $j \cdot T$. By taking into consideration the time taken to answer requests at

node i we can show that the worst case time μ_{ij} taken for establishing communication along the edge {i,j} is bounded as follows:

$$\mu_{ij} \leq ((2n-i)\cdot T + j\cdot T)\cdot(n-i) + (n-i)\cdot i\cdot T$$

From the above analysis the following theorem is easily proved

THEOREM 7.2: For the above algorithm the complexity $\mu \leq 2n^2 \cdot (T_c + 2T_m)$ □

For a complete graph $\gamma = n$ and hence we see that the complexity of the above algorithm has the same order as the lower bound proved in theorem 7.1. Thus it is a near optimal algorithm.

7.5. Algorithms which permit more interaction among Processes

In the previous section we considered algorithms with restricted interaction between schedulers. Due to the restriction, processes cannot withdraw requests. In this section we consider algorithms with improved interaction. We introduce two more shared variables w_{ij}, w_{ji} for each edge {i,j}. w_{ij} can only be updated by P_i. These variables take binary values. In the following algorithms schedulers use these variables to withdraw requests. Briefly, whenever P_i wants to withdraw its request to P_j it sets w_{ij} to 1. When P_j grants the withdrawal it sets w_{ji} to 1.

7.5.1. An algorithm using preemption of requests

Each scheduler process P_i maintains priorities of edges incident on it. It gives unique priorities to them which are dynamically updated. To maintain the priorities it keeps a queue Q_i of edges incidents on it. If edge e_1 is before e_2 in Q_i then e_1 has higher priority then e_2.

As in the previous algorithm it uses an a priori defined RW relation. In the beginning Q_i contains all the in coming edges (at node i in the RW-graph) before all the outgoing edges.

Protocol at node i:

We say that an edge e = {i,j} is ready iff $c_{ij} = 1$ and (e is an outgoing edge in the RW-graph or e is in coming edge on which there is a request)

Loop

If there is no ready edge **then** go to end-of-loop;

1. Get a ready edge e which has highest priority;

2. **If** e is an in comming edge **then** /*there is a request along e*/

   begin

3. Deny all other waiting requests of lower priority;

4. Put e at the end of Q_i;

   Establish Communication along e;

5. During the communication Ok all withdrawals;

   go to end-of-loop;

   end

6. **If** e is an outgoing edge **then**

   Begin /*let e be {i,j}*/

7. Deny all request of lower priority;

8. Request for communication along e;

9. **While** there is no reply along e **do**

10. Go through all waiting requests as follows:

11. **If** there is a request on $e' = \{i,k\}$ such that $c_{ik} = 0$ **then** deny the request;

12. **If** there is a request on a ready edge $e' = \{i,k\}$ with higher priority than e **then** request j for withdrawal by setting w_{ij} to 1;

13. **If** there is a withdrawal request along $e' = \{i,k\}$ such that e' has lower priority than e **then** Ok the withdrawal by setting w_{ik} to 1

```
        end while
      end
  /* the request along e is answered */

14.   If the request is granted i.e. v_ji = "G" then
      Begin

15.     Deny all waiting requests along edges e' such that
      e' has lower priority then e or e' is not ready;
16.     Grant all withdrawal requests;
      /* All ready requests along e' such that e' has
      higher priority than e and such that there is no
      withdrawal request along e' are kept waiting */
17.     Establish communication along e,
      during the communication grant all withdrawal requests;
      go to end-of-loop

      end
18.   If the request is denied i.e. v_ji = "D"
      then Put e at end of Q_i,
           go to end-of-loop;
19.   If the withdrawal request is granted without denial
      then go to end-of-loop;
      end
end-of-loop:
End of loop
```

We informally describe the algorithm below. The main features of the algorithm are that it uses dynamic priorities and that it allows preemption of requests. At the beginning of the loop, P_i picks up a ready edge of highest priority. If this is an in coming edge in the

RW-graph then it establishes communication and it denies all other requests of lower priority (new requests of higher priority might have arrived after the choice). During communication it okays all withdrawal requests. If the edge is an outgoing edge it does the following: it denies all lower priority requests. It requests for communication along this edge. While it is waiting for the reply, it checks for pre-emption. If there is a request on a higher priority edge it requests for withdrawal of previous communication request. It denies all requests along edges on which it is not willing to communicate. If there is a withdrawal request along an edge with lower priority it okays the withdrawal request. The other parts of the algorithm are self explanatory.

Let h be the number of edges in a longest path in the RW-graph.

LEMMA 7.3: Every communication request (or withdrawal request) is answered within time at most $(T_c + 2h \cdot T_m)$.

Proof: The only situation when a withdrawal request from P_i to P_j is not immediately answered is, when the withdrawal request arrives while P_j is executing in the loop at statement 9. Even in this case, the withdrawal request is not immediately answered only if P_j has requested along an edge of lower priority than {i,j}. And in this case P_j also requests for the withdrawal of it's communication request to the other process. Thus if a withdrawal request is not immediately answered, there must have formed a chain of withdrawal requests along some directed edges. This chain can be of length at most h. Hence it takes time at most $2h \cdot T_m$ before a withdrawal request is answered.

Assume there is a communication request from i to j. If the request to j arrives when P_j is outside the while loop at statement 9, then it will be answered after at most one

communication i.e. within time T_c. If the request arrives while P_j is executing the loop at statement 9, then a chain of requests forms. Let $i, j, j_0, \ldots, j_k, j_{k+1}, j_{k+2}$ be the chain of requesting processes. It is easily seen that the worst case occurs when there are no withdrawal requests along the chain and the request of j_k to j_{k+1} arrives just when communication along the edge $\{j_{k+1}, j_{k+2}\}$ has begun. After the communication is over j_{k+1} answers j_k which in turn immediately answers j_{k-1} and so on. Since the length of the chain can be at most h, it takes time almost $(T_c + 2h \cdot T_m)$ before the request of i is answered. □

THEOREM 7.4: The above algorithm ensures weak fairness and it has complexity $\mu \leq 2d^2 \cdot (T_c + h \cdot T_m)$.

Proof: Consider an edge e = (i,j) in the RW-graph such that $c_{ij} = c_{ji} = 1$ for sufficiently long time starting from the instance 't'. Assume that e has least priority at this instance at i as well as at j. If P_i chooses an edge e' at statement 1, then at that instance e' must be having higher priority than e. If e' is an in coming edge then after the communication along e' it will have lesser priority than e. Assume e' is an outgoing edge. If the request along e' is not withdrawn then after the request is answered it is given priority lesser then e. Assume the request along e' is withdrawn. Then this must have been due to a request along an in coming edge e" of higher priority than e'. In this case i immediately establishes communication along e" and gives it the least priority. Thus we see that after a request along e is answered, some edge which has higher priority than e before the request will have lesser priority after the request is answered. From this it follows that within d iterations of the main loop of P_i, P_i will request along e and this request will not be withdrawn. If this request is denied by P_j then it must have been the case that j established communication along an edge of higher priority than e at j (which will immediately be given

lesser priority than e at j) . Hence after at most d such requests e will have highest priority at j and communication will get established along this edge. Thus the complexity is bounded by the time required to make d^2 iterations of the main loop in P_i. The time required to make one iteration is at most equal to the sum of the time required for getting an answer to a request and the time for one communication (answering the other withdrawal or communication requests takes time which depends only on T_p and is negligible). Thus the complexity $\mu \leq 2d^2 \cdot (T_c + h \cdot T_m)$. □

As before we can get RW-graph such that $h \leq (d+1)$. In this case, $\mu \leq 2d^2 \cdot T_c + 2d^2(d+1) \cdot T_m$. Clearly this algorithm has better complexity than the one given in the previous section.

Even though the above algorithms guarantee weak fairness, in general they do not guarantee strong fairness. Indeed any algorithm that uses an a priori defined RW relation as above does not guarantee strong fairness in general. Assume G has cycles. Then there exist a pair of nodes i,j such that i RW j but not vice versa. Now consider a computation in which $c_{ij} = 1$ throughout the future and $c_{ji} = 1$ infinitely often: whenever i requests j for communication at that instance $c_{ji} = 0$ and j cannot keep the request of i waiting indefinitely and hence j rejects the request of i. But c_{ji} may be 1 when there is no request from i and i,j never establish communication.

7.5.2. An algorithm for strong fairness

We present a new algorithm that ensures strong fairness. In this algorithm there is no constraint on which process should request which other process, as in the previous algorithm. Instead any process can request any other process for communication.

The algorithm is a slight modification of the previous one. Each scheduler process P_i keeps two queues of edges Q'_i, Q''_i. Q'_i is used so that requests for communication are made in a fair way and it is maintained exactly as in the previous algorithm. Q''_i defines the priorities among edges used in the algorithm. Whenever there is a communication along the edge $e=\{i,j\}$ then e is given least priority at both i,j (i.e. e is placed at the end of Q''_i, Q''_j). Apart from the above change the modified algorithm has also the following changes. At line 1, the earliest ready edge on Q'_i is picked. This ensures that communication requests along the edges are made in a fair manner. At statement 2, it is checked if there is a request along e, in this case statements 3 through 5 are executed, otherwise the control goes statement 6.

At any instant of time let '>' be a relation defined among the edge of G as follows: Let $e_1 = \{i,j\}$, $e_2=\{i,k\}$. Then $e_1 > e_2$ iff e_1 has higher priority then e_2 at node i. Since the priorities are dynamic, so is the relation >. We require that in the beginning all Q''_i are initialized in such a way that > is acyclic.

LEMMA 7.5: In the above algorithm > is always acyclic.

Proof. The only situation when the priorities (i.e. Q''_i) are updated is whenever a communication is established along an edge. After the communication, both the processes on either side of the edge give least priority to that edge while preserving the relative priorities of the other edges. This update clearly preserves the acyclicity of >. □

Let L be the length of the longest path in G.

LEMMA 7.6: Any request is answered within time $(T_c + 2\,L \cdot T_m)$.

Proof: First we have to prove that every request gets eventually answered. The only problem comes when there is a circular chain of requests. In this case we want to show that deadlocks do not occur.

Let $i_0, i_1, \ldots, i_{k-1}, i_0$ be a sequence of nodes along a cyclic chain of requests i.e. for $0 \leq j < k$, i_j has a communication request to $i_{(j+1) \bmod k}$. Let $e_j = \{i_j, i_{(j+1) \bmod k}\}$. Since $>$ is an acyclic relation among the edges, it is easily seen that there exists at least one vertex i_j on this cycle such that $e_{(j-1) \bmod k}$ has higher priority then e_j at this node. Thus i_j will send a withdrawal request to $i_{(j+1) \bmod k}$. This guarantees that the cycle gets broken.

Waiting chains of length at most L can form. Now the time bound can be proved as in lemma 7.3. □

THEOREM 7.7: The above algorithm guarantees weak fairness and has complexity $\mu \leq d^2 \cdot (T_c + L \cdot T_m)$. The algorithm also guarantees strong fairness.

Proof: The weak fairness and the complexity can be proved as in theorem 7.4. It is to be observed that in the complexity bound there is no constant factor 2 as in theorem 7.4. This is because if $c_{ij} = c_{ji} = 1$ for a time period $d \cdot (2T_c + 2L \cdot T_m)$ and if i,j did not establish communication then each of them must have requested the other at least once during this time (instead of only one of them requesting as in the previous algorithm). The strong fairness property is easily seen from the fact that any process can request any other process. □

The main disadvantage of the above algorithm is that it's complexity depends on the size of the graph.

7.6. Conclusions

In this chapter we have considered the problem of achieving different fairness properties in communication among CSP processes. For a natural class of algorithms we have proved a lower bound on the time complexity for ensuring weak fairness. For special cases we presented near optimal algorithms. We also presented interesting new algorithms for ensuring weak and strong fairness properties. The algorithms we gave for strong fairness are not real time. This makes us conjecture that there are no real time algorithms that ensure strong fairness. The algorithms we presented can also be used for distributed implementation of other formal concurrent systems like Millner's CCS [Mi78].

Chapter 8

Conclusions

In this thesis we have addressed some important theoretical problems in the design and verification of distributed systems. In chapter 2 we examined the complexity of decision procedures for satisfiability of different versions of temporal logics. We gave a polynomial space bounded decision procedure for the full Propositional Linear Temporal Logic and we presented a decision procedure in NP for the logic that uses only the F(eventuality) operator. These results justify the use of temporal logic in program verification instead of first order language of linear order since it is known that the later logic is non-elementary. In chapter 2 we also considered the problem of automatic verification of finite state concurrent programs using specifications given in PTL. These results show that there may not be efficient algorithms for this problem. An important problem for future research is to investigate restricted versions of this logic for which there are efficient algorithms for automatic verification of finite state concurrent programs. In chapter 3 we extended PTL to QPTL by allowing quantifiers over propositions. We showed that the set of true sentences of this logic which are in normal form with a quantifier prefix that has one alternation of quantifiers, is EXSPACE-complete. We showed that for a weaker version of this logic (WQPTL) there is a tight space complexity hierarchy with the number of quantifier alternations for the set of true sentences that are in normal form. WQPTL is expressively equivalent to the well known logic WS1S(*Weak Monadic Second-order Theory of One Successor*). However WS1S is not known to exhibit such a nice hierarchy.

In chapter 4, we considered a branching time temporal logic for verifying concurrent systems. We modified the semantics of this logic so that only fair computations are considered. We presented efficient algorithms for automatic verification of finite state concurrent processes using the specifications given in this logic. We showed it's application to well known practical problems. We feel that this approach may be useful in the area of developing robust protocols. It will be worthwhile to see if there are more expressive branching time logics than the one used by us, for which there are efficient algorithms for automatically verifying finite state concurrent programs. In this chapter we also considered a branching time logic called CTL*. There are no good known decision procedures for this logic. This is an important open problem.

In chapter 5, we extended temporal logic in a novel manner by introducing spatial modalities in addition to the temporal modalities. This logic allows us to reason about temporal and spatial behavior in a unified formal system. We have given applications for this logic from wide areas of multiprocessor networks such as VLSI. We showed that the validity problem for this logic is undecidable. It is an interesting open problem if certain restricted versions of this logic are decidable. Another important problem to investigate is the use of this logic in verifying some algorithms.

In chapter 6, we considered the possibility of characterization and axiomatization of buffered message passing systems in temporal logic. We showed that all bounded buffers are characterizable and axiomatizable in temporal logic. We also proved that unbounded FIFO buffers are in general not axiomatizable while unbounded LIFO, unbounded unordered buffers are axiomatizable. These results answer some questions regarding the possibility of obtaining complete proof systems in temporal logics for the correctness of concurrent programs that use message buffers for interprocess communication.

Finally, in chapter 7 we explored the problem of distributed implementation of CSP that ensures certain fairness properties. The two fairness properties we considered are weak fairness and strong fairness. For a natural class of algorithms that ensure weak fairness, we proved a lower-bound on the time complexity of any algorithm in this class. We presented near optimal algorithms in special cases. In a slightly different model we presented algorithms for weak fairness which have better complexity. We also presented algorithms for strong fairness. For the model we considered it may be possible to improve our lower-bound. Other than time complexity, number of messages may be another complexity measure. It will be interesting to study this problem using this complexity measure.

References

[Be73] C. Berge, *Graphs and Hypergraphs*, North-Holland Publishing Company, 1973.

[Be80] Arthur J. Bernstein, *Output guards and nondeterminism in "Communicating Sequential Processes"*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 2, April 1980.

[BMP81] M. Ben-ari, Z. Manna, A. Pnueli, *The temporal logic of branching time*, 8th ACM Symposium on Principles of Programming Languages, 1981, Williamsburg, VA.

[BS83] G.N. Buckley, A. Silberschatz, *An effective implementation for the generalized input-output construct of CSP*, 1983.

[BSW69] K.A.Bartlet, R.A.Scantlebury,P.T.Wilkinson, *A note on reliable full-duplex transmission over half-duplex links*, Communications of ACM 12, 5(1969) 260-261.

[CE81] E.M. Clarke, A. Emerson, *Design and synthesis of programming skeltons using branching time temporal logic*, IBM Conference of Logics of Programs, 1981, May.

[CES83] E.M.Clarke, A.Emerson, A.P.Sistla, *Automatic verification of finite state concurrent systems using temporal logic specifications: a practical approach*, Proceedings of POPL 83.

[Co69] S.N. Cole, *Real time computations by n-dimensional iterative arrays of finite state machines*, IEEE Transactions on Computers, 1969, 18, page 349-365.

[Ch] Ernest Chang, *Echo algorithms: depth parallel operations on general graphs* University of Toronto.

[EC80] E.A.Emerson, E.M.Clarke, *Characterizing properties of parallel programs as fixpoints*, Proceedings of the 7th International Colloquium on Automata,Languages and Programming, Lecture notes in Computer Science #85,1981.

[EH83] E.A.Emerson,J.Y.Halpern, *Sometimes and not never revisited: on branching versus linear time*, POPL 83.

[ES83] E.A.Emerson,A.P.Sistla, *Deciding branching time temporal logics*, Workshop on logics of programs, Carnegie-Mellon University, June 6-8, 1983.

[FL79] M. Fischer, R. Ladner, *Propositional dynamic logic of regular programs*, JCSS, 1979, 18(2).

[GPSS80] D. Gabbay, A. Pnueli, S. Shealah, J. Stavi, *Temporal analysis of fairness*, Seventh ACM Symposium on Principles of Programming Languages, 1979, Las Vegas, NV, December.

[Ho78] C.A.R.Hoare, *Communicating sequential processes*, Communications of the ACM 21,8(August 1978) 666-667.

[HO80] B.T.Hailpern,S.Owicki, *Verifying network protocols using temporal logic*, Tech. Report 192, Computer systems laboratory, Stanford university, June 1980.

[HR81] J. Y. Halpern, J.H. Reif, *The Propositional dynamic logic of deterministic, well-structured programs*, 22nd Symposium on Foundations of Computer Scinece, 1981, Nashville, TN.

[KL78] H.T.Kung, C.E.Leiserson, Symposium on Sparse Matrix Computations and their Applications, Knoxville, Tennessee, Nov. 1978.

[Ko69] S.R. Kosaraju, *Computations on iterative automata*, University of Pennsylvania, 1969, Ph.D. Thesis.

[La77] R. Ladner, *The computational complexity of provability in systems of modal propositional logic*, SIAM J. Comp. 6, 1977, 467-480 (A9-Z).

[Le81] Charles Eric Leiserson, *Area-efficient VLSI computation*, Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Oct.1981.

[Ly80] Nancy A. Lynch, *Fast allocation of nearby resources in a distributed system*, Proceedings of ACM Symposium on Theory of Computing, 1980.

[Mi78] R. Milner, *Synthesis of communicating behavior*, 7th Symposium on Mathematical Foundations of Computer Science, Zakopane, Poland, 1978.

[MP81] Z. Manna, A. Pnuelli, *Verification of concurrent programs*, The Correctness Problem in Computer Science, International Lecture Series in Computer Sciences, 1981, Academic Press, London.

[MW81] Z. Manna, P.Wolper, *Synthesizing concurrent programs from temporal logic specifications*, IBM Conference on logics of programs,1981.

[OL80] S.Owicki, L.Lamport, *Proving liveness properties of concurrent programs*, Stanford University Technical Report 1980.

[Ow76] S.Owicki, *A Consistent and complete deductive system for verification of parallel programs*, 8th Annual Symposium on Theory of Computing, 1976

[Pn77] A. Pnueli, *The temporal logic of programs*, Proceedings of the Eighteenth Symposium on Foundations of Computer Science, 1977, Providence, RI, November.

[PV79] F.P. Preparata, J. Vuillemin, *The cube connected cycles: a versatile network for parallel computation*, FOCS, 1979, page 140-147.

[QS81] J.P.Quielle, J.Sifakis, *Specification and verification of concurrent systems in CESAR*, Proceedings of 5th International Symposium in Programming 1981.

[QS82] J.P.Quielle, J.Sifakis, *Fairness and related properties in transition systems* IMAG, 292(March 1982).

[Ro] Edward L.Robertson, *Structure of complexity in Weak Monadic Second Order Theories of the Natural numbers*,1977.

[RS81] J.H. Reif, P.G. Spirakis, *Distributed algorithms for synchronizing interprocess communication in real time*, Proceedings of 13th ACM symposium on Theory of Computing, Milwaukee, Wisconsin, May 1981.

[RSi83] J.H. Reif, A.P.Sistla, *A multi-processor network logic with spatial and temporal modalities*, Proceedinds of International Conference on Automata,Languages and Programming 1983, Barcelona, Spain.

[Sa70] W.J.Savitch, *Relationships between nondeterministic and deterministic tape complexities*, J. Computer and Systems Sciences 4:2, 177-192.

[Sc] Jerald Schwarz, *Distributed synchronization of communicating sequential processes*, DAI Research Report No. 56), Department of Artificial Intelligence, University of Edinburg.

[Sc80] J.T. Schwartz, *Ultracomputers*, ACM Transactions on Programming Languages and Systems, 1980, Vol.12, No.4, October, page 484-521.

[SCFG82] A.P. Sistla, E.M. Clarke, N. Francez, Y. Gurevich, *Are message buffers characterizable in linear temporal logic*, Proceedings of the Symposium on Principles of Distributed Computering, 1982, Ottawa, Canada, 1982.

[SC82] A.P. Sistla, E.M. Clarke, *The complexity of propositional linear temporal logics*, ACM Symposium on Theory of Computing, 1982, Page 159-167.

[Si79] Abraham Silberschatz, *Communication and synchronization in distributed systems*, IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, Nov. 1979.

[Si] D.P.Sidhu, *Rules for synthesizing correct communication protocols*, PNL Print, To appear in SIGCOMM.

[Sp81] Paul George Spirakis, *Probabilistic algorithms, algorithms with random inputs, and random combinatorial structures*, TR-33-81, Center for Research in Computing Technology, Harvard University, 1981.

[St71] H.S. Stone, *Parallel processing with the perfect shuffle*, IEEE Transactions on Computers, 1971, Vol. c-20, No. 2, February.

[Wo81] P. Wolper, *Temporal logic can be more expressive*, Proceedings of 22nd Symposium on Foundations of Computer Science, 1981, Nashville, TN, October.

[Wo82] P.Wolper, Ph.D. Thesis, Stanford University, 1982.

[Za80] P.Zafiropulo, C.West, H.Rudin, D.Cowan, D.Brand, *Towards analyzing and synthesizing protocols*, IEEE Transactions on Communications COM-28(April 1980), 651-671.