TEMPORAL AND MODAL LOGIC*

by

E. Allen Emerson
Computer Sciences Department
University of Texas at Austin
Austin, Texas 78712
USA

March 14, 1995

Abstract

We give a comprehensive and unifying survey of the theoretical aspects of Temporal and Modal Logic. (Note: This paper is to appear in the *Handbook of Theoretical Computer Science*, J. van Leeuwen, managing editor, North-Holland Pub. Co.)

*Work supported in part by US NSF Grant CCR8511354 , ONR Contract N00014-86-K-0763 and Netherlands ZWO grant nf-3/nfb 62-500.

1 Introduction

The class of Modal Logics was originally developed by philosophers to study different "modes" of truth. For example, the assertion P may be false in the present world, and yet the assertion *possibly* P true, if there exists an alternate world where P is true. Temporal Logic is a special type of Modal Logic; it provides a formal system for qualitatively describing and reasoning about how the truth values of assertions change over time. In a system of Temporal Logic, various temporal operators or "modalities" are provided to describe and reason about how the truth values of assertions vary with time. Typical temporal operators include *sometimes* P which is true now if there is a future moment at which P becomes true and *always* Q which is true now if Q is true at all future moments.

In a landmark paper [Pn77] Pnueli argued that Temporal Logic could be a useful formalism for specifying and verifying correctness of computer programs, one that is especially appropriate for reasoning about nonterminating or continuously operating concurrent programs such as operating systems and network communication protocols. In an ordinary sequential program, e.g. a program to sort a list of numbers, program correctness can be formulated in terms of a Precondition/Postcondition pair in a formalism such as *Hoare's Logic* because the program's underlying semantics can be viewed as given by a *transformation* from an initial state to a final state. However, for a continuously operating, *reactive* program such as an operating system, its normal behavior is a nonterminating computation which maintains an ongoing interaction with the environment. Since there is no final state, formalisms such as Hoare's logic which are based on a transformational semantics, are of little use for such nonterminating programs. The operators of temporal logic such as *sometimes* and *always* appear quite appropriate for for describing the time-varying behavior of such programs.

These ideas were subsequently explored and extended by a number of researchers. Now Temporal Logic is an active area of research interest. It has been used or proposed for use in virtually all aspects of concurrent program design, including specification, verification, manual program composition (development), and mechanical program synthesis. In order to support these applications a great deal mathematical machinery connected with Temporal Logic has been developed. In this survey we focus on this machinery, which is most relevant to Theoretical Computer Science. Some attention is given, however, to motivating applications.

The remainder of this paper is organized as follows: In section 2 we describe a multi-axis classification of systems of Temporal Logic, in order to give the reader a feel for the large variety of systems possible. Our presentation centers around only a few—those most thoroughly investigated—types of Temporal Logics. In section 3 we describe the framework of Linear Temporal Logic. In both its propositional and First-order forms, Linear Temporal Logic has been widely employed in the specification and verification of programs. In section 4 we describe the competing framework of Branching Temporal Logic which has also seen wide use. In section 5 we describe how Temporal Logic structures can be used to model concurrent programs using nondeterminism and fairness. Technical machinery for Temporal reasoning is discussed in section 6, including decision procedures and axiom systems. Applications of Temporal Logic are discussed in section 7, while in the concluding section 8 other modal and temporal logics in computer science are briefly described.

2 Classification of Temporal Logics

We can classify most systems of TL (Temporal Logic) used for reasoning about concurrent programs along a number of axes: propositional versus first-order, global versus compositional, branching versus linear, points versus intervals, and past versus future tense. Most research to date has concentrated on global, point-based, discrete time, future tense logics; therefore our survey will focus on representative systems of this type. However, to give the reader an idea of the wide range of possibilities in formulating a system of Temporal Logic, we describe the various alternatives in more detail below.

2.1 Propositional versus First-order

In a propositional TL, the non-temporal (i.e., non-modal) portion of the logic is just classical propositional logic. Thus formulae are built up from *atomic propositions*, which intuitively express atomic facts about the underlying state of the concurrent system, *truth-functional connectives*, such as $\wedge$, $\vee$, $\neg$ (representing "and," "or," and "not," respectively), and the temporal operators. Propositional TL corresponds to the most abstract level of reasoning, analogous to classical propositional logic.

The atomic propositions of propositional TL are refined into expressions built up from variables, constants, functions, predicates, and quantifiers, to get *First-order TL*. There are several different types of First order TLs. We can distinguish between *uninterpreted* First order TL where we make no assumptions about the special properties of structures considered, and *interpreted* First order TL where a specific structure (or class of structures) is assumed. In a *fully* interpreted First order TL, we have a specific domain (e.g. *integer* or *stack*) for each variable, a specific, concrete function over the domain for each function symbol, and so forth, while in a *partially* interpreted First order TL we might assume a specific domain but, e.g., leave the function symbols uninterpreted. It is also common to distinguish between *local* variables which are assigned, by the semantics, different values in different states and *global* variables which are assigned a single value which holds globally over all states. Finally, we can choose to impose or not impose various syntactic restrictions on the interaction of quantifiers and temporal operators. An unrestricted syntax will allow, e.g., modal operators within the scope of quantifiers. For example, we have instances of *Barcan's Formula*: $\forall$y *always* (P(y)) $\equiv$ *always* ($\forall$y P(y)). Such unrestricted logics tend to be highly undecidable. In contrast we can disallow such quantification over temporal operators to get a restricted first-order TL consisting of essentially propositional TL plus a first-order language for specifying the "atomic" propositions.

2.2 Global versus Compositional

Most systems of TL proposed to date are *endogenous*. In an endogenous TL, all temporal operators are interpreted in a single universe corresponding to a single concurrent program. Such TLs are suitable for *global* reasoning about a complete, concurrent program. In an *exogenous* TL, the syntax of the temporal operators allows expression of correctness properties concerning several different programs (or program fragments) in the same formula. Such logics facilitate *compositional* (or *modular*) program reasoning: We can verify a complete program by specifying and verifying its constituent subprograms, and then combining them into a complete program together with its proof of correctness, using the proofs of the subprograms as lemmas (cf. [BKP84], [Pn84]).

2.3 Branching versus Linear Time

In defining a system of temporal logic, there are two possible views regarding the underlying nature of time. One is that the course of time is linear: At each moment there is only one possible future moment. The other is that time has a branching, tree-like nature: At each moment, time may split into alternate courses representing different possible futures. Depending upon which view is chosen, we classify a system of temporal logic as either a linear time logic in which the semantics of the time structure is linear, or a system of branching time logic based on the semantics corresponding to a branching time structure. The temporal modalities of a temporal logic system usually reflect the character of time assumed in the semantics. Thus, in a logic of linear time, temporal modalities are provided for describing events along a single time line. In contrast, in a logic of branching time, the modalities reflect the branching nature of time by allowing quantification over possible futures. Both approaches have been applied to program reasoning, and it is a matter of debate as to whether branching or linear time is preferable (cf. [La80], [EH86], [Pn85])

2.4 Points versus Intervals

Most temporal logic formalisms developed for program reasoning have been based on temporal operators that are evaluated as true or false of *points* in time. Some formalisms (cf. [SMV83], [Mo83], [HS86]),however, have temporal operators that are evaluated over *intervals* of time, the claim being that use of intervals greatly simplifies the formulation of certain correctness properties.

The following related issue has to do with the underlying structure of time.

2.5 Discrete versus Continuous

In most temporal logics used for program reasoning, time is *discrete* where the present moment corresponds to the program's current state and the next moment corresponds to the program's immediate successor state. Thus the temporal structure corresponding to a program execution, a sequence of states, is the nonnegative integers. However, tense logics interpreted over a *continuous* (or *dense*) time structure such as the reals (or rationals) have been investigated by philosophers. Their application to reasoning about concurrent programs was proposed in [BKP86] to facilitate the formulation of fully abstract semantics. Such continuous time logics may also have applications in so-called real-time programs where strict, quantitative performance requirements are placed on programs.

2.6 Past versus Future

As originally developed by philosophers, temporal modalities were provided for describing the occurrence of events in the past as well as the future. However, in most temporal logics for reasoning about concurrency, only future tense operators are provided. This appears reasonable since, as a rule, program executions have a definite starting time, and it can be shown that, as a consequence, inclusion of past tense operators adds no expressive power. Recently, however, it has been advanced that use of the past tense operators might be useful simply in order to make the formulation of specifications more natural and convenient (cf. [LPZ85]). Moreover, past tense operators appear to play an important role in compositional specification somewhat analogous to that of history variables.

3 The Technical Framework of Linear Temporal Logic

3.1 Timelines

In linear temporal logic the underlying structure of time is a totally ordered set (S,<). In the sequel we will further assume that the underlying structure of time is isomorphic to the natural numbers with their usual ordering ($\mathbb{N}$,<). Note that under our assumption time,

(i) is discrete,

(ii) has an initial moment with no predecessors, and

(iii) is infinite into the future.

We remark that these properties seem quite appropriate in view of our intended application: reasoning about the behavior of ongoing concurrent programs. Property (i) reflects the fact that modern day computers are discrete, digital devices; property (ii) is appropriate since computation begins at an initial state; and (iii) is appropriate since we develop our formalism for reasoning about ongoing, ideally nonterminating behavior.

Let AP be an underlying set of atomic proposition symbols, which we denote by P, Q, P_1, Q_1, P', Q', etc. We can then formalize our notion of a timeline as a *linear time structure* M = (S,x,L) where

S — is a set of *states*,
x : $\mathbb{N} \rightarrow$ S — is an infinite sequence of states, and
L : S $\rightarrow$ PowerSet(AP) — is a labelling of each state with
the set of atomic propositions in AP true at the state.

We usually employ the more convenient notation $x = (s_0, s_1, s_2, \ldots) = (x(0), x(1), x(2), \ldots)$ to denote the *timeline* x. Alternative terminology permits us to refer to x as a *fullpath*, or *computation sequence*, or *computation*, or simply as a *path*; the latter could cause confusion in rare instances since we intend the *maximal path* x, not just one of its prefixes, whence the term fullpath; but ordinarily no confusion will result. (Other synonyms include execution, execution sequence, trace, history, and run.)

Remark: We could convey the same information associating a truth value to each atomic proposition at each state by defining the labelling L as a mapping AP $\rightarrow$ PowerSet(S) which assigns to each atomic proposition the set of states at which it is true. Another equivalent alternative is to use a mapping $L : S \times AP \rightarrow \{true, false\}$ such that $L(s,P) = true$ iff it is intended that P be true at s. Still another alternative is to have $L : S \rightarrow (AP \rightarrow \{true, false\})$ so that L(s) is an interpretation of each proposition symbol at state s. In the future, we will use whichever presentation is most convenient for the purpose at hand, assuming the above equivalences to be obvious.

3.2 Propositional Linear Temporal Logic

In this subsection we will define the formal syntax and semantics of Propositional Linear Temporal Logic (PLTL). The basic temporal operators of this system are Fp ("sometime p"; also read as "eventually p"), Gp ("always p"; also read as "henceforth p"), Xp ("nexttime p"), and p U q ("p until q"). Figure 1 below illustrates their intuitive meanings. The formulae of this system are built up from atomic propositions, the truth-functional connectives ($\wedge$, $\vee$, $\neg$, etc.) and the above-mentioned temporal operators. This system, or some slight variation thereof, is frequently employed in applications of temporal logic to concurrent programming.

3.2.1 Syntax

The set of formulae of Propositional Linear Temporal Logic (PLTL) is the least set of formulae generated by the following rules:

1. Each atomic proposition P is a formula.
2. If p and q are formulae then $p \wedge q$ and $\neg p$ are formulae.
3. If p and q are formulae then p U q and Xp are formulae.

The other formulae can then be introduced as abbreviations in the usual way: For the propositional connectives, $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, and $p \Leftrightarrow q$ abbreviates $(p \Rightarrow q) \wedge (q \Rightarrow p)$. The boolean constant *true* abbreviates $p \vee \neg p$, while *false* abbreviates $\neg$*true*. Then the temporal connective Fp abbreviates (true U p) and Gp abbreviates $\neg F \neg p$. It is convenient to also have $\overset{\infty}{F}p$ abbreviate GFp (infinitely often), $\overset{\infty}{G}p$ abbreviate FGp ("almost everywhere"), and (p B q) ("p precedes q") abbreviate $\neg(\neg p \text{ U } q)$.

Remark: The above is an *abstract syntax* where we have suppressed detail regarding parenthesization, binding power of operators, and so forth. In practice, we use the following notational conventions, supplemented by auxiliary parentheses as needed: The connectives of highest binding power are the temporal operators F, G, X, U, B, $\overset{\infty}{F}$, and $\overset{\infty}{G}$. The operator $\neg$ is of next highest binding power, followed by $\wedge$, followed by $\vee$, followed by $\Rightarrow$, followed finally by $\Leftrightarrow$ as the operator of least binding power.

Example: $\neg p_1 \text{ U } q_1 \wedge r_1 \vee r_2$ means $(\neg(p_1 \text{ U } q_1)) \wedge r_1) \vee r_2$.

3.2.2 Semantics

We define the semantics of a formula p of PLTL with respect to a linear time structure $M = (S,x,L)$ as above. We write $M,x \models p$ to mean that "in structure M formula p is true of timeline x." When M is understood we write $x \models p$. The notational convention that x^i = the suffix path $s_i, s_{i+1}, s_{i+2} \ldots$ is used. We define $\models$ inductively on the structure of the formulae:

1. $x \models P$ iff $P \in L(s_0)$, for atomic proposition P
2. $x \models p \wedge q$ iff $x \models p$ and $x \models q$
   $x \models \neg p$ iff it is not the case that $x \models p$
3. $x \models (p \mathrm{\ U\ } q)$ iff $\exists j\ (x^j \models q$ and $\forall k < j(x^k \models p))$
   $x \models Xp$ iff $x^1 \models p$

The modality (p U q), read as "p until q" asserts that q does eventually hold and that p will hold everywhere prior to q.

The modality Xp, read as "next time p" holds now iff p holds at the next moment.

For conciseness, we took the temporal operator U and X as primitive, and defined the others as abbreviations. However, the other operators are themselves of sufficient independent importance that we also give their formal definitions explicitly.

The modality Fq, read as "sometimes q" or "eventually q" and meaning that at some future moment q is true, is formally defined so that

$x \models Fq$ iff $\exists j\ (x^j \models q)$

The modality Gq, read as "always q" or "henceforth q" and meaning that at all future moments q is true, can be formally defined as

$x \models Gq$ iff $\forall j\ (x^j \models q)$

The modality (p B q), read as "p precedes q" or "p before q" and which intuitively means that "if q ever happens in the future, it is strictly preceded by an occurrence of p," has the following formal definition

$x \models (p \mathrm{\ B\ } q)$ iff $\forall j\ (x^j \models q$ implies $\exists k < j(x^k \models p))$

The modality $\overset{\infty}{F}p$, which is read as "infinitely often p," intuitively means that it is always true that p eventually holds, or in other words that p is true infinitely often, can be defined formally as

$x \models \overset{\infty}{F}p$ iff $\forall k\ \exists j \geq k\ x^j \models p$

The modality $\overset{\infty}{G}p$, which is read as "almost everywhere p" or "almost always p," intuitively means that p holds at all but a finite number of times, can be defined as

$x \models \overset{\infty}{G}p$ iff $\exists k\ \forall j > k\ x^j \models p$

3.2.3 Basic Definitions

We say that PLTL formula p is *satisfiable* iff there exists a linear time structure M = (S,x,L) such that $x \models p$. We say that any such structure defines a *model* of p. We say that p is *valid*, and write $\models p$, iff for all linear time structures M = (S,x,L) we have $x \models p$. Note that p is valid iff $\neg p$ is not satisfiable.

3.2.4 Examples

We have the following examples:

$p \Rightarrow Fq$ intuitively means that "if p is true now then at some future moment q will be true." This formula is satisfiable, but not valid.

$G(p \Rightarrow Fq)$ intuitively means that "whenever p is true, q will be true at some subsequent moment." This formula is also satisfiable, but not valid.

$G(p \Rightarrow Fq) \Rightarrow (p \Rightarrow Fq)$ is a valid formula, but its converse only satisfiable.

$p \wedge G(p \Rightarrow Xp) \Rightarrow Gp$ means that if p is true now and whenever p is true it is also true at the next moment, then p is always true. This formula is valid, and is a temporal formulation of mathematical induction.

$(p \text{ U } q) \wedge ((\neg p) \text{ B } q)$ means that p will be true until q eventually holds, and that the first occurrence of q will be preceded by $\neg p$. This formula is unsatisfiable.

Significant Validities

The duality between the linear temporal operators are illustrated by the following assertions:

$$\models G \neg p \equiv \neg Fp$$
$$\models F \neg p \equiv \neg Gp$$
$$\models X \neg p \equiv \neg Xp$$
$$\models \overset{\infty}{F} \neg p \equiv \neg \overset{\infty}{G} p$$
$$\models \overset{\infty}{G} \neg p \equiv \neg \overset{\infty}{F} p$$
$$\models ((\neg p) \text{ U } q) \equiv \neg(p \text{ B } q)$$

The following are some important implications between the temporal operators, which cannot be strengthened to equivalences:

$$\models p \Rightarrow Fp$$
$$\models Gp \Rightarrow p$$
$$\models Xp \Rightarrow Fp$$
$$\models Gp \Rightarrow Xp$$
$$\models Gp \Rightarrow Fp$$
$$\models Gp \Rightarrow XGp$$
$$\models p \text{ U } q \Rightarrow Fq$$
$$\models \overset{\infty}{G} q \Rightarrow \overset{\infty}{F} q$$

The idempotence of F, G, $\overset{\infty}{F}$, and $\overset{\infty}{G}$ are asserted below:

$\models FFp \equiv Fp$
$\models \overset{\infty}{F}\overset{\infty}{F} \equiv \overset{\infty}{F}p$
$\models GGp \equiv Gp$
$\models \overset{\infty}{G}\overset{\infty}{G}p \equiv \overset{\infty}{G}p$

Note: of course, $XXp \equiv Xp$ is not valid. We also have that X commutes with F, G, and U

$\models XFp \equiv FXp$
$\models XGp \equiv GXp$
$\models ((Xp)\ U\ (Xq)) \equiv X(p\ U\ q)$

The infinitary modalities $\overset{\infty}{F}$ and $\overset{\infty}{G}$ "gobble up" other unary modalities applied to them:

$\models \overset{\infty}{F}p \equiv X\overset{\infty}{F}p \equiv F\overset{\infty}{F}p \equiv G\overset{\infty}{F}p \equiv \overset{\infty}{F}\overset{\infty}{F}p \equiv \overset{\infty}{G}\overset{\infty}{F}p$
$\models \overset{\infty}{G}p \equiv X\overset{\infty}{G}p \equiv F\overset{\infty}{G}p \equiv G\overset{\infty}{G}p \equiv \overset{\infty}{F}\overset{\infty}{G}p \equiv \overset{\infty}{G}\overset{\infty}{G}p$

(Note: in the above we make use of the abuse of notation that $\models a_1 \equiv ... \equiv a_n$ abbreviates the n-1 valid equivalences $\models a_1 \equiv a_2, ..., \models a_{n-1} \equiv a_n$.) The F, $\overset{\infty}{F}$ operators have an existential nature, the G, $\overset{\infty}{G}$ operators a universal nature, while the U operator is universal in its first argument and existential in its second argument. We thus have the following distributivity relations between these temporal operators and the boolean connectives $\wedge$ and $\vee$:

$\models F(p \vee q) \equiv (Fp \vee Fq)$
$\models \overset{\infty}{F}(p \vee q) \equiv (\overset{\infty}{F}p \vee \overset{\infty}{F}q)$
$\models G(p \wedge q) \equiv (Gp \wedge Gq)$
$\models \overset{\infty}{G}(p \wedge q) \equiv (\overset{\infty}{G}p \wedge \overset{\infty}{G}q)$
$\models ((p \wedge q)\ U\ r) \equiv ((p\ U\ r) \wedge (q\ U\ r))$
$\models (p\ U\ (q \vee r)) \equiv ((p\ U\ q) \vee (p\ U\ r))$

Since the X operator refers to a unique next moment, it distributes with all the boolean connectives:

$\models X(p \vee q) \equiv (Xp \vee Xq)$
$\models X(p \wedge q) \equiv (Xp \wedge Xq)$
$\models X(p \Rightarrow q) \equiv (Xp \Rightarrow Xq)$
$\models X(p \equiv q) \equiv (Xp \equiv Xq)$

(Note: $\models X\neg p \equiv \neg Xp$ was given above.)

When we mix operators of universal and existential characters we get the following implications, which again cannot be strengthened to equivalences:

$\models (Gp \vee Gq) \Rightarrow G(p \vee q)$
$\models (\overset{\infty}{G}p \vee \overset{\infty}{G}q) \Rightarrow \overset{\infty}{G}(p \vee q)$
$\models F(p \wedge q) \Rightarrow Fp \wedge Fq$
$\models \overset{\infty}{F}(p \wedge q) \Rightarrow (\overset{\infty}{F}p \wedge \overset{\infty}{F}q)$
$\models ((p\ U\ r) \vee (q\ U\ r)) \Rightarrow ((p \vee q)\ U\ r)$
$\models (p\ U\ (q \wedge r)) \Rightarrow ((p\ U\ q) \wedge (p\ U\ r))$

We next note that the temporal operators below are monotonic in each argument:

$$\models G(p \Rightarrow q) \Rightarrow (Gp \Rightarrow Gq)$$
$$\models G(p \Rightarrow q) \Rightarrow (Fp \Rightarrow Fq)$$
$$\models G(p \Rightarrow q) \Rightarrow (Xp \Rightarrow Xq)$$
$$\models G(p \Rightarrow q) \Rightarrow (\overset{\infty}{F}p \Rightarrow \overset{\infty}{F}q)$$
$$\models G(p \Rightarrow q) \Rightarrow (\overset{\infty}{G}p \Rightarrow \overset{\infty}{G}q)$$
$$\models G(p \Rightarrow q) \Rightarrow ((p\ U\ r) \Rightarrow (q\ U\ r))$$
$$\models G(p \Rightarrow q) \Rightarrow ((r\ U\ p) \Rightarrow (r\ U\ q))$$

Finally, we have following important *fixpoint characterizations* of the temporal operators (cf. Section 8.4):

$$\models Fp \equiv p \vee XFp$$
$$\models Gp \equiv p \wedge XGp$$
$$\models (p\ U\ q) \equiv q \vee (p \wedge X(p\ U\ q))$$
$$\models (p\ B\ q) \equiv \neg q \wedge (p \vee X(p\ B\ q))$$

3.2.5 Minor Variants of PLTL

One minor variation is to change the basic temporal operators. There are a number of variants of the until operator p U q, which is defined as the *strong until*: there does exist a future state where q holds and p holds until then. We could write $p\ U_s\ q$ or $p\ U_\exists\ q$ to emphasize its strong, existential character. The operator *weak until*, written $p\ U_w\ q$ (or $p\ U_\forall q$), is an alternative. It intuitively means that p holds for as long as q does not, even forever if need be. It is also called the *unless* operator. Its technical definition can be formulated as:

$$x \models p\ U_\forall\ q \text{ iff } \forall j\ (\ (\forall k \leq j\ x^k \models \neg q) \text{ implies } x^j \models p\ )$$

exhibiting its "universal" character. Note that, given the boolean connectives, each until operator is expressible in terms of the other:

(a) $p\ U_\exists\ q \equiv p\ U_\forall\ q \wedge Fq$

(b) $p\ U_\forall\ q \equiv p\ U_\exists\ q \vee Gp \equiv p\ U_\exists\ q \vee G(p \wedge \neg q)$

We also have variations based on the answer to the question: does the future include the present? The future does include the present in our formulation, and is thus called the *reflexive* future. We might instead formulate versions of the temporal operators referring to the *strict* future, i.e., those times strictly greater than the present. A convenient notation for emphasizing the distinction involves use of $>$ or $\geq$ as a superscript:

$F^{>}p$ — $\exists$ a strict future moment when p holds
$F^{\geq}p$ — $\exists$ a moment, either now or in the future, when p holds
$F^{>}p \equiv XF^{\geq}p$
$F^{\geq}p \equiv p \vee F^{>}p$

Similarly we have the strict always ($G^{>}p$) in addition to our "ordinary" always ($G^{\geq}p$).

The *strict* (strong) until P $U^{>}$ q $\equiv$ X(p U q) is of particular interest. Note that *false* $U^{>}$q $\equiv$ X(false U q) $\equiv$ Xq. The single modality strict, strong until is enough to define all the other linear time operators (as shown by Kamp [Ka68].)

Remark: One other common variation is simply notational. Some authors use $\Box$p for Gp, $\Diamond$p for Fp, and $\circ$p for Xp.

Another minor variation is to change the underlying structure to be any initial segment I of |N, possibly a finite one. This seems sensible because we may want to reason about terminating programs as well as nonterminating ones. We then correspondingly alter the meanings of the basic temporal operators, as indicated (informally) below:

Gp — for all subsequent times in I, p holds.
Fp — for some subsequent times in I, p holds.
p U q — for some subsequent time in I q holds, and p holds at all subsequent times until then.

We also now can distinguish two notions of nexttime:

$X_\forall$p—weak nexttime—if there exists a successor moment then p holds there
$X_\exists$ p—strong nexttime—there exists a successor moment and p holds there

Note that each nexttime operator is the dual of the other: $X_\exists$p $\equiv$ ($\neg X_\forall$ $\neg$p and $X_\forall$p $\equiv$ $\neg X_\exists$ $\neg$p).

Remark: Without loss of generality, we can restrict our attention to structures where the timeline = |N and still get the effect of finite timelines. This can be done in either of two ways:

(a) Repeat the final state so the finite sequence $s_0\ s_1 \ldots s_k$ of states is represented by the infinite sequence $s_0\ s_1 \ldots s_k\ s_k\ s_k \ldots$. (This is somewhat like adding a self-loop at the end of a finite, directed linear graph.)

(b) Have a proposition P_{GOOD} true for exactly the good (i.e., finite) portion of the timeline.

Adding past tense temporal operators

As used in computer science, all temporal operators are future tense; we might use the following suggestive notation and terminology for emphasis:

F^+p — sometime in the future p holds,
G^+p — always in the future p holds,
X^+p — nexttime p holds (Note: “next” implies implicitly the future)
p U^+ q — sometime in the future q holds and p holds subsequently until then

However, as originally studied by philosophers there were past tense operators as well; we can use the corresponding notation and terminology:

F^-p — sometime in the past p holds
G^-p — always in the past p holds
$X_\exists^-$p — lasttime p holds (Note: “last” implicitly refers to the past)
p U^-q — sometime in the past q holds and p holds previously until then

When needed for emphasis we use PLTLF for the logic with just future tense operators, PLTLP for the logic with just past tense operators, and PLTLB for the logic with both.

For temporal logic using the past tense operators, given a linear time structure M = (S,x,L) we interpret formulae over a pair (x,i), where x is the timeline and the natural number index i specifies *where* along the timeline the formula is true. Thus, we write M, (x,i) $\models$ p to mean that "in structure M along timeline x at time i formula p holds true;" when M is understood we write just (x,i) $\models$ p. Intuitively, pair (x,i) corresponds to the suffix x^i, which is the forward interval x[i:∞) starting at time i, used in the definition of the future tense operators. When the past is allowed the pair (x,i) is needed since formulae can reference positions along the entire timeline, both forward and backward of position i. If we restrict our attention to just the future tense as in the definition of PLTL, we can omit the second component of (x,i) – in effect assuming that i=0, and that formulae are interpreted at the beginning of the timeline – and write x $\models$ p for (x,0) $\models$ p.

The technical definitions of the basic past tense operators are as follows:

(x,i) $\models$ p U^-q iff $\exists$j(j $\leq$ i and (x,j) $\models$ q and $\forall$k (j $<$ k $\leq$ i implies (x,j) $\models$ p))
(x,i) $\models$ $X_\exists^-$p iff i $\geq$ 0 and (x,i-1) $\models$ p

Note that the lasttime operator is strong, having an existential character, asserting that there is a past moment; thus is false at time 0.

The other past connectives are then introduced as abbreviations as usual: e.g., the weak lasttime $X_\forall^-$p for $\neg X_\exists^-$ $\neg$p, F^-p for (*true* U^-p), and G^-p for $\neg F^-$ $\neg$p.

For comparison we also present the definitions of some of the basic future tense operators using the pair (x,i) notation:

(x,i) $\models$ (p U q) iff $\exists$j (j $\geq$ i and (x,j) $\models$ q and $\forall$k(i $\leq$ k $<$ j implies (x,k) $\models$ p))
(x,i) $\models$ Xp iff (x,i+1) $\models$ p
(x,i) $\models$ Gq iff $\forall$j(j $\geq$ i implies (x,j) $\models$ q)
(x,i) $\models$ Fq iff $\exists$j(j $\geq$ i and (x,j) $\models$ q)

Remark: Philosophers used a somewhat different notation. F^-p was usually written as Pp, G^-p as Hp, and p U q as p S q meaning "p since q." We prefer the present notation due to its more uniform character.

The decision whether to allow i to float or to anchor it at 0 yields different notions of equivalence, satisfiability, and validity. We say that a formula p is *initially satisfiable* provided there exists a linear time structure M = (S,x,L) such that M,(x,0) $\models$ p. We say that a formula p is *initially valid* provided for all timeline structures M = (S,x,L) we have M,(x,0) $\models$ p. We say that a formula p is *globally satisfiable* provided that there exists a linear time structure M = (S,x,L) and time i such that M,(x,i) $\models$ p. We say that a formula p is *globally valid* provided that for all linear time structures M = (S,x,L) and times i we have M,(x,i) $\models$ p.

In an almost trivial sense inclusion of the past tense operators increases the expressive power of our logic:

We say that formula p is *globally equivalent* to formula q, and write p $\equiv_g$ q, provided that $\forall$ linear structure x $\forall$ time i $\in$ |N [(x,i) $\models$ p iff (x,i) $\models$ q].

Theorem 3.1. As measured with respect to global equivalence, PLTLB is strictly more expressive than PLTLF.

Proof. The formula $F^{-}Q$ is not expressible in PLTLF, as can be seen by considering two structures x, x′ as depicted below.

$$\begin{array}{l} \quad Q \\ \quad \bullet \quad \rightarrow \bullet \sim\sim\sim\sim\rightarrow \\ \neg Q \\ \quad \bullet \quad \rightarrow \bullet \sim\sim\sim\sim\rightarrow \end{array}$$

The structures are identical except for their respective state at time 0. At time 1 $F^{-}Q$ distinguishes the two structures (i.e. $(x,1) \models F^{-}Q$ and $(x',1) \not\models F^{-}Q$) yet future tense PLTLF cannot distinguish $(x,1)$ from $(x',1)$, since the infinite suffixes beginning at time 1 are identical. □

Yet in the sense that programs begin execution in an initial state, inclusion of the past tense operators adds no expressive power.

We say that formula p is *initially equivalent* to formula q, and write $p \equiv_i q$, provided that $\forall$ linear structure x $[(x,0) \models p$ iff $(x,0) \models q]$.

Theorem 3.2. As measured with respect to initial equivalence, PLTLB is equivalent in expressive power to PLTLF.

This can be proved using results regarding the theory of linear orderings (cf. [GPSS80]):

We also note the following relationship between $\equiv_i$ and $\equiv_g$:

Proposition 3.3. $p \equiv_g q$ iff $Gp \equiv_i Gq$.

By convention we shall take satisfiable to mean initially satisfiable and valid to mean initially valid, unless otherwise stated. Intuitively, this makes sense since programs start execution in an initial state. Moreover, whenever we refer to expressive power we are measuring it with respect to initial equivalence, unless otherwise stated. One benefit of comparing expressive power on the basis of initial equivalence, is that it suggests we view formulae of PLTL and its variants as defining sets of sequences, i.e. formal languages. (See section 6.)

3.3 First-Order Linear Temporal Logic (FOLTL)

First-order linear temporal logic (FOLTL) is obtained by taking propositional linear temporal logic (PLTL) and adding to it a First order language **L**. That is, in addition to atomic propositions, truth-functional connectives, and temporal operators we now also have predicates, functions, individual constants, and individual variables, each interpreted over an appropriate domain with the standard Tarskian definition of truth.

Symbols of L

We have a first order language **L** over a set of *function* symbols and a set of *predicate* symbols. The zero-ary function symbols comprise the subset of *constant* symbols. Similarly, the zero-ary predicate symbols are known as the *proposition* symbols. Finally, we have a set of individual *variable* symbols.

We use the following notations:

ϕ, ψ, ..., etc. for n-ary, $n \geq 1$, predicate symbols,
P, Q, ..., etc. for proposition symbols,
f, g, ..., etc. for n-ary, $n \geq 1$, function symbols,
c, d, ..., etc. for constant symbols, and
y, z, ..., etc. for variable symbols.

We also have the distinguished binary predicate symbol $\approx$, known as the *equality symbol*, which we use in the standard infix fashion. Finally, we have the usual quantifier symbols $\forall$ and $\exists$, denoting universal and existential quantification, respectively, which are applied to individual variable symbols, using the usual rules regarding scope of quantifiers, and free and bound variables.

Syntax of L

The *terms* of **L** are defined inductively by the following rules:

T1 Each constant c is a term.

T2 Each variable y is a term.

T3 If $t_1, \ldots, t_n$ are terms and f is an n-ary function symbol then $f(t_1, \ldots, t_n)$ is a term.

The *atomic formulae* of **L** are defined by the following rules:

AF1 Each 0-ary predicate symbol (i.e. atomic proposition) is an atomic formula.

AF2 If $t_1, \ldots, t_n$ are terms and ψ is an n-ary predicate then $\psi(t_1, \ldots, t_n)$ is an atomic formula.

AF3 If t_1, t_2 are terms then $t_1 \approx t_2$ is also an atomic formula.

Finally, the (compound) formulae of **L** are defined inductively as follows:

F1 Each atomic formula is a formula.

F2 If p,q are formulae then $(p \wedge q)$, $\neg p$ are formulae.

F3 If p is a formula and y is a free variable in p then $\exists y p$ is a formula.

Semantics of L

The semantics of **L** is provided by an interpretation I over some domain D. The interpretation I assigns an appropriate meaning over D to the (non-logical) symbols of **L**: Essentially, the n-ary predicate symbols are interpreted as concrete, n-ary relations over D, while the n-ary function symbols are interpreted as concrete, n-ary functions on D. (Note: an n-ary relation over D may be viewed as an n-ary function $D^n \to |B$, where $|B = \{true, false\}$ is the distinguished Boolean domain.) More precisely I assigns a meaning to the symbols of **L** as follows:

- for an n-ary predicate symbol ψ, $n \geq 1$, the meaning $I(\psi)$ is a function $D^n \to |B$
- for a proposition symbol P, the meaning I(P) is an element of |B
- for an n-ary function symbol f, $n \geq 1$, the meaning I(f) is a function $D^n \to D$
- for an individual constant symbol c, the meaning I(c) is an element of D
- for an individual variable symbol y, the meaning I(y) is an element of D

The interpretation I is extended to arbitrary terms, inductively:

$I(f(t_1, \ldots, t_n)) = I(f)\ (I(t_1), \ldots, I(t_n))$

We now define the meaning of truth under interpretation I of formula p, written $I \models p$. First, for atomic formulae we have:

$I \models P$, where P is an atomic proposition, iff $I(P) = true$.
$I \models \psi(t_1, \ldots, t_n)$, where ψ is an n-ary predicate and $t_1, \ldots, t_n$ are terms,
iff $I(\psi)\ (I(t_1), \ldots, I(t_n)) = true$.
$I \models t_1 \approx t_2$ iff $I(t_1) = I(t_2)$.

Next, for compound formulae we have:

$I \models p \wedge q$ iff $I \models p$ and $I \models q$.
$I \models \neg p$ iff it is not the case that $I \models p$.
$I \models \exists y\ p$, where y is a free variable in p, iff there exists some $d \in D$ such that $I[y \leftarrow d] \models p$ where $I[y \leftarrow d]$ is the interpretation identical to I except that y is assigned value d.

Global versus Logical Symbols

For defining First Order Linear Temporal Logic (FOLTL), we assume that the set of symbols is divided into two classes, the class of *global* symbols and the class of *local* symbols. Intuitively, each global symbol has the same interpretation over all states; the interpretation of a local symbol may vary, depending on the state at which it is evaluated. We will subsequently assume that all function symbols (and thus all constant symbols) are global, and that all n-ary predicate symbols, for $n \geq 1$, are also global. Proposition symbols (i.e. 0-ary predicate symbols) and variable symbols may be local or may be global.

A *(first order) linear time structure* M = (S,x,L) is defined just as in the propositional case, except that L now associates with each state s an interpretation L(s) of all symbols at s, such that for each global symbol w, $L(s)(w) = L(s')(w)$, for all $s,s' \in S$. Note that the structure M has an underlying domain D, as for **L**. Also, it is sometimes convenient to refer to the global interpretation I associated with M by $I(w) = L(s)(w)$, where w is any global symbol and s is any state of M. (Note: Implicitly given with a structure is its *signature* or *similarity type* consisting of the alphabets of all the different kinds of symbols. The signature of the structure is assumed to match that of the language (FOLTL).)

Description of FOLTL

We are now ready to define the language of First-Order Linear Temporal Logic (FOLTL) obtained by adding **L** to PLTL. First, the terms of FOLTL are those generated by rules T1-3 for **L** plus the rule:

T4 If t is a term then Xt is a term (intuitively, denoting the immediate future value of term t).

The atomic formulae of FOLTL are generated by the same rules as for **L**, but now are used in conjunction with the expanded set of rules T1-4 for terms.

Finally, the (compound) formulae of FOLTL are defined inductively using the following rules:

FOLTL1 Each atomic formula is a formula.

FOLTL2 If p,q are formulae, then so are $p \wedge q$, $\neg p$.

FOLTL3 If p,q are formulae, then so are p U q, Xp.

FOLTL4 If p is a formula and y is a free variable in p, then $\exists y$ p is a formula.

The semantics of FOLTL is provided by a first order linear time structure M over a domain D as above. Global interpretation I of M assigns a meaning to each global symbol, while the local interpretations L(–) associated with M assign a meaning to each local symbol.

Since the terms of FOLTL are generated by rules T1-3 for **L** plus the rule T4 above, we extend the meaning function—now denoted by a pair (M,x)—for terms:

(M,x) (c) = I(c), since all constants are global.
(M,x) (y) = I(y), where y is a global variable.
(M,x) (y) = $L(s_0)$ (y), where y is a local variable and $x = (s_0,s_1,s_2,\ldots)$.
(M,x) $(f(t_1, \ldots, t_n))$ = (M,x) (f) $((M,x)\ (t_1), \ldots, (M,x)\ (t_n))$.
(M,x) (Xt) = (M,x^1) (t).

Now the extension of $\models$ is routine. For atomic formulae we have:

$M,x \models P$ iff $I \models P$ where P is a global proposition.
$M,x \models P$ iff $L(s_0)$ (P) = *true* where P is a local proposition and $x = (s_0,s_1,s_2,\ldots)$.
$M,x \models \psi(t_1,\ldots,t_n)$ iff (M,x) (ψ) $((M,x)\ (t_1),\ldots, (M,x)\ (t_n))$ = *true*.
$M,x \models t_1 \approx t_2$ iff (M,x) (t_1) = (M,x) (t_2).

We finish off the semantics of FOLTL with the inductive definition of $\models$ for compound formulae:

$M,x \models p \wedge q$ iff $M,x \models p$ and $M,x \models q$.

$M,x \models \neg p$ iff it is not the case that $M,x \models p$.

$M,x \models (p\ U\ q)$ iff $\exists j$ ($M,x^j \models q$ and $\forall k<j$ $(M,x^k \models p)$)

$M,x \models Xp$ iff $M,x^1 \models p$

$M,x \models \exists y\ p$, where y is a global variable free in p, iff there exists some $d \in D$ for which $M[y \leftarrow d],x \models p$, where $M[y \leftarrow d]$ is the structure having global interpretation $I[y \leftarrow d]$ identical to I except y is assigned the value d.

A formula p of FOLTL is *valid* iff for every for every first order linear time structure M = (S,x,L) we have $M,x \models p$. The formula p is *satisfiable* iff there exists M = (S,x,L) such that $M,x \models p$.

Remark. For notational simplicity we have assumed the **L** is a one-sorted first order language. Thus each symbol (function symbol, predicate symbol, etc.) is of the same sort and is interpreted over the single domain D. For certain applications, it is more convenient to assume that **L** is a multi-sorted language, where the symbols of **L** are partitioned into different sets, each of which corresponds to a different domain with different argument positions. The extension to multi-sorted languages is routine, although a bit cumbersome notationally.

4 The Technical Framework of Branching Temporal Logic

4.1 Tree-like Structures

In branching time temporal logics, the underlying structure of time is assumed to have a branching tree-like nature where each moment may have many successor moments. The structure of time thus corresponds to an infinite tree. In the sequel, we will further assume that along each path in the tree, the corresponding time line is isomorphic to |N. We do allow a node in the tree to have infinitely many (even uncountably many) successors, while we require each node to have at least one successor. It will turn out that, as far as our branching temporal logics are concerned, such trees are indistinguishable from trees with finite, even bounded, branching. Trees of the latter type have a natural correspondence with the computations of concurrent or nondeterministic programs, as discussed in the next section.

We say that a *temporal structure* M = (S,R,L) where

S is the set of *states*,

R is a total *binary relation* $\subseteq S \times S$ (i.e., one where $\forall s \in S\ \exists t \in S\ (s,t) \in R$), and

L:S $\rightarrow$ PowerSet(AP) is a labelling which associate with each state s an interpretation L(s) of all atomic proposition symbols at state s.

We may view M as a labelled, directed graph with node set S, arc set R, and node labels given by L. We say (the graph of) M

(a) is *acyclic* provided it contains no directed cycles;

(b) is *tree-like* provided that it is acyclic and each node has at most 1 R-predecessor (i.e., there is no "merging" of paths); and

(c) is a *tree* provided that it is tree-like and there exists a unique node—called the *root*—from which all other nodes of M are reachable and that has no R-predecessors.

We have not required that (the graph of) M be a tree. However, we may assume, without loss of generality, that it is. We define the Structure $\hat{M} = (\hat{S},\hat{R},\hat{L})$, called the structure obtained by unwinding M starting at state $s_0 \in S$, where $\hat{S}$, $\hat{R}$ are, respectively, the least subsets of $S \times |N$, $\hat{S} \times \hat{S}$ such that:

- $(s_0,0) \in \hat{S}$
- if $(s,n) \in \hat{S}$ then
  $\{(t,n+1) : t \text{ is an R-successor of s in M}\} \subseteq \hat{S}$, and
  $\{((s,n), (t,n+1)) : t \text{ is an R-successor of s in M}\} \subseteq \hat{R}$;

and $\hat{L}((s,n)) = L(s)$. Then (the graph of) $\hat{M}$ is a tree with root $(s_0,0)$, and it is easily checked that, for all the branching time logics we will consider, a formula p holds at s_0 in M iff p holds at $(s_0,0)$ in $\hat{M}$. See Figure 2.

4.2 Propositional Branching Temporal Logics

In this section we provide the formal syntax and semantics for two representative systems of propositional branching time temporal logics The simpler logic, CTL (Computational Tree Logic) allows basic temporal operators of the form: a path quantifier—either A ("for all futures") or E ("for some future"—followed by a single one of the usual linear temporal operators G ("always"), F ("sometime"), X ("nexttime"), or U ("until"). It corresponds to what one might naturally first think of as a branching time logic. CTL is closely related to branching time logics proposed in [La80], [EC80], [QS81], [BPM81], and was itself proposed in [CE81]. However, as we shall see, its syntactic restrictions significantly limit its expressive power. We therefore also consider the much richer language CTL*, which is sometimes referred to informally as full branching time logic. The logic CTL* extends CTL by allowing basic temporal operators where the path quantifier (A or E) is followed by an arbitrary linear time formula, allowing boolean combinations and nestings, over F, G, X, and U. It was proposed as a unifying framework in [EH86], subsuming both CTL and PLTL, as well as a number of other systems. Related systems of high expressiveness are considered in [Pa79], [Ab80], [ST81], and [VW83].

Syntax

We now give a formal definition of the syntax of CTL*. We inductively define a class of state formulae (true or false of states) using rules S1-3 below and a class of path formulae (true or false of paths) using rules P1-3 below:

S1 Each atomic proposition P is a state formula

S2 If p,q are state formulae then so are $p \wedge q$, $\neg p$

S3 If p is a path formula then Ep, Ap are state formulae

P1 Each state formula is also a path formula

P2 If p,q are path formulae then so are $p \wedge q$, $\neg p$

P3 If p,q are path formulae then so are Xp, p U q

The set of state formulae generated by the above rules forms the language CTL*. The other connectives can then be introduced as abbreviations in the usual way.

Remark: We could take the view that Ap abbreviates $\neg$E $\neg$p, and give a more terse syntax in terms of just the primitive operators E, $\wedge$, $\neg$, X, and U. However, the present approach makes it easier to give the syntax of the sublanguage CTL below.

The restricted logic CTL is obtained by restricting the syntax to disallow boolean combinations and nestings of linear time operators. Formally, we replace rules P1-3 by

P0 if p,q are state formulae then Xp, p U q are path formulae.

The set of state formulae generated by rules S1-3 and P0 forms the language CTL. The other boolean connectives are introduced as above while the other temporal operators are defined as abbreviations as follows: EFp abbreviates E(*true* U p), AGp abbreviates $\neg$EF $\neg$p, AFp abbreviates A(*true* U p), and EGp abbreviates $\neg$AF$\neg$p. (Note: this definition can be seen to be consistent with that of CTL*.)

Also note that the set of path formulae generated by rules by P1-P3 yield the linear time PLTL.

Semantics

A formula of CTL* is interpreted with respect to a structure M = (S,R,L) as defined above. A *fullpath* of is an infinite sequence $s_0, s_1, s_2, \ldots$ of states such that $\forall i\ (s_i, s_{i+1}) \in R$. We use the convention that $x = (s_0, s_1, s_2, \ldots)$ denotes a fullpath, and that x^i denotes the suffix path $(s_i, s_{i+1}, s_{i+2}, \ldots)$. We write $M, s_0 \models p$ (respectively, $M, x \models p$) to mean that state formula p (respectively, path formula p) is true in structure M at state s_0 (respectively, of fullpath x). We define $\models$ inductively as follows:

S1 $M, s_0 \models p$ iff $P \in L(s_0)$

S2 $M, s_0 \models p \wedge q$ iff $M, s_0 \models p$ and $M, s_0 \models q$ $M, s_0 \models \neg p$ iff $\mathrm{not}(M, s_0 \models p)$

S3 $M, s_0 \models Ep$ iff $\exists$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in M, $M, x \models p$
$M, s_0 \models Ap$ iff $\forall$ fullpath $x = (s_0, s_1, s_2, \ldots)$ in M, $M, x \models p$

P1 $M, x \models p$ iff $M, s_0 \models p$

P2 $M, x \models p \wedge q$ iff $M, x \models p$ and $M, x \models q$
$M, x \models \neg p$ iff $\mathrm{not}(M, x \models \neg p)$

P3 $M, x \models p\ U\ q$ iff $\exists i\ [M, x^i \models q$ and $\forall j\ (j < i$ implies $M, x^j \models p)]$
$M, x \models Xp$ iff $M, x^1 \models p$

A formula of CTL is also interpreted using the CTL* semantics, using rule P3 for path formulae generated by rule P0.

We say that a state formula p (resp., path formula p) is *valid* provided that for every structure M and every state s (resp., fullpath x) in M we have $M,s \models p$ (resp., $M,x \models p$). A state formula p (resp., path formula p) is *satisfiable* provided that for some structure M and some state s (resp., fullpath x) in M we have $M,s \models p$ (resp., $M,x \models p$).

Generalized Semantics

We can define CTL* and other logics over various generalized notions of structure. For example, we could consider more general structures $M = (S,X,L)$ where S is a set of states and L a labelling of states as usual, while $X \subseteq S^\omega$ is a family of infinite computation sequences (fullpaths) over S. The definition of CTL* semantics carries over directly, with path quantification restricted to paths in X, provided that "a fullpath x in M" is understood to refer to a fullpath x in X.

In the most general case X can be completely arbitrary. However, it is often helpful to impose certain requirements on X (cf. [La80], [Pr79], [Ab80], [Em83]). We say that X is *suffix closed* provided that if computation $s_0s_1s_2... \in X$, then the suffix $s_1s_2... \in X$. Similarly, X is *fusion closed* provided that whenever $x_1sy_1, x_2sy_2 \in X$ then $x_1sy_2 \in X$. The idea is that the system should always be able to follow the prefix of one computation and then continue along the suffix sy_2 of another computation; thus the computation actually followed is the "fusion" of two others. Both suffix and fusion closure are needed to ensure that the future behavior of a program depends only on the current state and not how the state is reached.

We may also wish to require that X be *limit closed* meaning that whenever x_1y_1, $x_1x_2y_2$, $x_1x_2x_3y_3$,... are all elements of X, then the infinite path $x_1x_2x_3$..., which is the limit of the prefixes x_1,x_1,x_2,$x_1x_2x_3$..., is also in X. In short, if it possible follow a path arbitrarily long, then it can be followed forever. Finally, a set of paths is *R-generable* if there exists a total binary relation R on S such that a sequence $x = s_0s_2s_2... \in X$ iff $\forall i\ (s_i,s_{i+1}) \in R$. It can be shown that X is R-generable iff it is limit closed, fusion closed and suffix closed. Of course, the basic type of structures we ordinarily consider are R-generable, which correspond to the execution of a program under pure nondeterministic scheduling.

Some such restrictions on the set of paths X are usually needed in order to have the abstract, computation path semantics reflect the behavior of actual concurrent programs. An additional advantage of these restrictions is that they ensure the validity of many commonly accepted principles of temporal reasoning. For example, fusion closure is needed to ensure that $EFEFp \equiv EFp$. Suffix closure is needed for $EFp \wedge \neg p \Rightarrow EXEFp$, and limit closure for $p \wedge AGEXp \Rightarrow EGp$. An R-generable structure satisfies all these natural properties.

Another generalization is to define a *multiprocess temporal structure*, which is a refinement of the notion of a branching temporal structure that distinguishes between different processes. Formally, a multiprocess temporal structure $M = (S,\mathbf{R},L)$ where

S is a set of states,

R is a finite family $\{R_1,\ldots,R_k\}$ of binary relations R_i on S (intuitively, R_i represents the transitions of process i) such that $R = \cup\, \mathbf{R}$ is total (i.e. $\forall s \in S\ \exists t \in S\ (s,t) \in R$), L associates with each state an interpretation of symbols at the state.

Just as for a (uniprocess) temporal structure, a multiprocess temporal structure may be viewed as a directed graph with labelled nodes and arcs. Each state is represented by a node that is labelled by the atomic propositions true there, and each transition relation R_i is represented by a set of arcs that are labelled with index i. Since there may be multiple arcs labelled with distinct indices between the same pair of nodes, technically the graph-theoretic representation is a directed multigraph.

The previous formulation of CTL* over uniprocess structures refers only to the atomic formulae labelling the nodes. However, it is straightforward to extend it to include, in effect, arc assertions indicating which process performed the transition corresponding to an arc. This extension is needed to formulate the technical definitions of fairness in the next section, so we briefly describe it.

Now, a fullpath $x = (s_0,d_1,s_1,d_2,s_2,\ldots)$, depicted below

$$\begin{array}{ccccccccc} & d_1 & & d_2 & & d_3 & & d_4 & \\ \bullet & \rightarrow & \bullet & \rightarrow & \bullet & \rightarrow & \bullet & \rightarrow \ldots & \\ s_0 & & s_1 & & s_2 & & s_3 & & \end{array}$$

is an infinite sequence of states s_i alternating with relation indices d_{i+1} such that $(s_i,s_{i+1}) \in R_{d_{i+1}}$, indicating that process d_{i+1} caused the transition from s_i to s_{i+1}. We also assume that there are distinguished propositions $enabled_1, \ldots, enabled_k$, $executed_1, \ldots executed_k$, where intuitively $enabled_j$ is true of a state exactly when process j is enabled, i.e., when a transition by process j is possible, and $executed_j$ is true of a transition when it is performed by process j. Technically, each $enabled_j$ is an atomic proposition—and hence a state formula—true of exactly those states in domain R_j:

$$M,s_0 \models enabled_j \text{ iff } s_0 \in \text{domain } R_j = \{ s \in S : \exists t \in S\ (s,t) \in R \}$$

while each $executed_j$ is an atomic arc assertion—and a path formula such that

$$M,x \models executed_j \text{ iff } d_1 = j.$$

It is worth pointing out that there are alternative formalisms that are essentially equivalent to this notion of a (multiprocess) structure. A *transition system* M is a formalism equivalent to a multi-process temporal structure consisting of a triple $M = (S,\mathbf{R},L)$ where $\mathbf{R}$ is a finite family of transitions $\tau_i : S \rightarrow \text{PowerSet}(S)$. To each transition τ_i there is a corresponding relation $R_i = \{(s,t) \in S \times S\text{: } t \in \sigma_i\}$ and conversely. Similarly, there is a correspondence between multiprocess temporal logic structures and **do-od** programs (cf. [Di76]). Assume we are given a **do-od** program $\rho =$ **do** $B_1 \rightarrow A_1$ [] ... [] $B_k \rightarrow A_k$ **od**, where each B_i may be viewed as subset of the state space S and each A_i as a function $S \rightarrow S$. Then we may define an equivalent structure $M=(S,\mathbf{R},L)$, where each $R_i = \{ (s,t) \in S\text{: } s \in B_i \text{ and } t = A_i(s)\}$, and L gives appropriate meanings to the symbols in the program. Conversely, given a structure M, there is a corresponding generalized **do-od** program ρ, where by generalized we mean that each action A_i is allowed to be a relation; viz., it is **do** $B_1 \rightarrow A_1$ []...[] $B_k \rightarrow A_k$ **od**, where each $B_i = \text{domain } R_i = \{ s \in S\text{: } \exists\ t \in S\ (s,t) \in R_i\}$ and $A_i = R_i$.

We can define a single type of general structure which subsumes all of those above. We assume an underlying set of symbols, divided into global and local subsets as before and called *state symbols* to emphasize that they are interpreted over states, as well as an additional set of *arc assertion* *symbols* that are interpreted over transitions (s,t) $\in$ R. Typically we think of L((s,t)) as the set of indices (or names) of processes which could have performed the transition (s,t). A (*generalized*) *fullpath* is now a sequence of states s_i alternating with arc assertions d_i as depicted above.

Now we say that a *general structure* M = (S,R,X,L) where

- S is a set of *states*,
- R is a total *binary relation* $\subseteq$ S $\times$ S,
- X is a set of *fullpaths* over R, and
- L is a mapping associating with each state s an interpretation L(s) of all state symbols at s, and with each transition (s,t) $\in$ R an interpretation of each arc assertion at (s,t)

There is no loss of generality due to including R in the definition: for any set of fullpaths X, let R = {(s,t) $\in$ S $\times$ S: there is a fullpath of the form ystz in X, where y is a finite sequence of states and z an infinite sequence of states in S}; then all consecutive pairs of states along paths in X are related by R.

The extensions needed to define CTL* over such a general structure M are straightforward. The semantics of path quantification as specified in rule S3 carries over directly to the general M, provided that a "full path in M" refers to one in X. If d is an arc assertion we have that:

M,x $\models$ d iff d $\in$ L((s_0,s_1))

4.3 First-Order Branching Temporal Logic

We can define systems of First-order Branching Temporal Logic. The syntax is obtained by combining the rules for generating a system of propositional Branching Temporal Logic plus a (multi-sorted) first-order language. The underlying structure M = (S,R,L) is extended so that it associates with each state s an interpretation L(s) of local and global symbols at state s, including in particular local variables as well as local atomic propositions. The semantics is given by the usual Tarskian definition of truth. Validity and satisfiability are defined in the usual way. The details of the technical formulation are closely analogous to those for first-order linear temporal logic and are omitted here.

5 Concurrent Computation: A Framework

5.1 Modelling Concurrency by Nondeterminism and Fairness

Our treatment of concurrency is the usual one where concurrent execution of a system of processes is modelled by the nondeterministic interleaving of atomic actions of the individual processes. The semantics of a concurrent program is thus given by a computation tree: a concurrent program starting in a given state may follow any one of a (possibly infinite) number of different computation paths in the tree (i.e., sequences of execution states) corresponding to the different sequences of nondeterministic choices the program might make. Alternatively, the semantics can be given simply by the set of all possible execution sequences, ignoring that they can be organized into a tree, for each possible starting state.

We remark that it is always possible to model concurrency by nondeterminism, since by picking a sufficiently fine level of granularity for the atomic actions to be interleaved, any behavior that could be produced by true concurrency (i.e., true simultaneity of action) can be simulated by interleaving. In practice, it is helpful to use as coarse of granularity as possible, as it reduces the number of interleavings that must be considered.

There is one additional consideration in the modelling of concurrency by nondeterminism. This is the fundamental notion of *fair scheduling assumptions*, commonly called *fairness*, for short.

In a truly concurrent system, implemented physically, it would be reasonable to assume that each sequential process P_i of a concurrent program $P_1 \| \ldots \| P_n$ is assigned to its own physical processor. Depending on the relative rates of speed at which the physical processors ran, we would expect that the corresponding nondeterministic choices modeling this concurrent system, would favor, more often the faster processes. For a very simple example, consider a system $P_1 \| P_2$ with just two processes. If each process ran on its own physical processor, and the processors ran at approximately equal speeds, we would expect the corresponding sequence of interleavings of steps of the individual processes to be of the form:

$$P_1 P_2 P_1 P_2 P_1 P_2 \ldots$$

or

$$P_2 P_1 P_2 P_1 P_2 P_1 \ldots$$

or, perhaps

$$P_1 P_1 P_2 P_1 P_2 P_2 P_1 P_1 P_2 \ldots$$

where, for each i, after i steps in all have been executed, roughly i/2 steps of each individual process has been executed. If processor 1 ran, say, three times faster than processor 2 we would expect corresponding interleavings such as

$$P_1 P_1 P_1 P_2 P_1 P_1 P_1 P_2 P_1 P_1 P_1 P_2 \ldots$$

where steps of process P_1 occur about 3 times more often than steps of process P_2.

Now, on the other hand, we would not expect to see a sequence of actions such as $P_1 P_1 P_1 P_1 \ldots$ where process P_1 is always chosen while process P_2 is never chosen. This would be *unfair* to process P_2. Under the assumption that each processor is always running at some positive, finite speed, regardless of how the relative ratios of the processor's speed might vary, we would thus expect to see *fair* sequences of interleavings where each process is executed infinitely often. This notion of fair scheduling thereby corresponds to the reasonable and very weak assumption that each process makes some progress. In the sequel, we shall assume that the nondeterministic choices of which process is to next execute a step are such that resulting infinite sequence is fair.

For the present we let the above notion of fairness—that each process be executed infinitely often—suffice; actually, however, there are a number of technically distinct refinements of this notion. (See, for example, the book by Francez [Fr86] as well as [Ab80], [FK84], [GPSS80], [La80], [LPS81], [Pn83], [QS83], [LPZ85] and [EL85].) Some of these will be described subsequently.

Thus to model the semantics of concurrency accurately we need fairness assumptions in addition to the computation sequences generated by nondeterministic interleaving of the execution of individual processes.

We remark on an advantage afforded by fairness assumptions. By the principal of separation of concerns, we should distinguish the issue of correctness of a program, from concerns with its efficiency or performance. Correctness is a qualitative sort of property. To say that we are concerned that a program be totally correct means we wish to establish that it does eventually terminate meeting a certain post condition. Establishing just when it terminates is a quantitative sort of property that is distinct from the qualitative notion of eventually terminating. Temporal logic is especially appropriate for such qualitative reasoning. Moreover, fairness assumptions facilitate such qualitative reasoning. Since fairness corresponds to the very weak qualitative notion that each process is running at some finite positive speed, programs proved correct under a fair scheduling assumption will be correct no matter what the rates are at which the processors actually run.

We very briefly summarize the preceding discussion by saying that, for our purposes, concurrency = nondeterminism + fairness. Somewhat less pithily but more precisely and completely, we can say that a concurrent program amounts to a global state transition system, with global state space essentially the cartesian product of the state spaces of the individual sequential processes and transitions corresponding to the atomic actions of the individual sequential processes, plus a fairness constraint and a starting condition. The behavior of a concurrent program is then described in terms of the trees (or simply sets) containing all the computation sequences of the global state transition system which meet the fair scheduling constraint and starting condition.

5.2 Abstract Model of Concurrent Computation

With the preceding motivation, we are now ready to describe our abstract model of concurrent computation.

An *abstract concurrent program* is a triple (M, ϕ_{START}, Φ) where M is a (multiprocess) temporal structure, ϕ_{START} is an atomic proposition corresponding to a distinguished set of starting states in M, Φ is a fair scheduling constraint which we, for convenience, take to be specified in linear temporal logic.

Among possible fairness constraints, are the following very common ones:

(1) *Impartiality*: An infinite sequence is impartial iff every process is executed infinitely often during the computation, which is expressed by $\Phi = \wedge_{i=1}^{k} \overset{\infty}{\mathrm{F}}\ \mathit{executed}_i$

(2) *Weak fairness* (also known as *justice*): An infinite computation sequence is *weakly fair* iff every process enabled almost everywhere is executed infinitely often, which is expressed by $\Phi = \wedge_{i=1}^{k} (\overset{\infty}{\mathrm{G}}\ \mathit{enabled}_i \Rightarrow \overset{\infty}{\mathrm{F}}\ \mathit{executed}_i)$

(3) *Strong fairness* (also known simply as *fairness*): An infinite computation sequence is strongly fair iff every process enabled infinitely often is executed infinitely often, which is expressed by $\Phi = \wedge_{i=1}^{k} (\overset{\infty}{\mathrm{F}}\ \mathit{enabled}_i \Rightarrow \overset{\infty}{\mathrm{F}}\ \mathit{executed}_i)$

5.3 Concrete Models of Concurrent Computation

Different concrete models of concurrent computation can be obtained from our abstract model by refining it in various ways. These include:

(i) providing structure for the global state space,

(ii) defining (classes of) instructions which each process can execute to manipulate the state space, and

(iii) providing concrete domains for the global state space.

We now describe some concrete models of concurrent computation.

Concrete Models of Parallel Computation Based on Shared Variables

Here, we consider parallel programs of the form $P_1 \| P_2 \| \ldots \| P_k$ consisting of a finite, fixed set of sequential processes $P_1, \ldots, P_k$ running together in parallel. There is also an underlying set of *variables* $v_1, \ldots, v_m$ assuming values in a domain D, that are *shared* among the processes in order to provide for inter-process communication and coordination. Thus, the global state set S consists of tuples of the form $(l_1, \ldots, l_k, v_1, \ldots, v_m) \in \times_{h=1}^{k} LOC(P_h) \times \times_{h'=1}^{m} D_{h'}$, where each process P_i has an associated set $LOC(P_i) = \{l_i^1, \ldots, l_i^{n_i}\}$ of locations. Each process P_i is described by a transition diagram with nodes labelled by locations. Alternatively, a process can be described by an equivalent text. Associated with each arc (l,l') there is an *instruction* I which may be executed by process P_i whenever process P_i is selected for execution and the current global state has the location of P_i at l. The instruction I is presented as a guarded command $B \rightarrow A$, where guard B is a predicate over the variables $\bar{v}$ and action A is an assignment $\bar{u} := \bar{e}$ of a tuple of expressions to the corresponding tuple of variables.

It is possible to make further refinements of the model. By imposing appropriate restrictions on the way instructions can access (i.e., read) and manipulate (i.e., write) the data we can get models ranging from those that can perform “test-and-set” instructions which permit a read followed by a write in a single atomic operation on a variable to those that only permit an atomic read or an atomic write of a variable.

We might also wish to impose restrictions on which processes are allowed which kind of access to which variables. One such rule is that each variable v is “owned” by some one unique process P, (think of v as being in the “local” memory of process p); then, each process can read any variable in the system, while only the process which owns a variable can write into it. This specialization is referred to as the *distributed shared-variables* model.

Still, another refinement is to specify a specific domain for the variables, say |N = the natural numbers. Yet another is to specify the type of instructions (e.g. “copy the value of variable y into variable z”). They can be combined to get a completely concrete program with instructions such as “load the value of variable z into variable y and decrement by the natural number 1.”

Concrete Models of Parallel Computation based on Message Passing

This model is similar to the previous one. However, each process has its own set of local variables $y_1, \ldots, y_n$ that cannot be accessed by other processes. All interprocess communication is effected by message passing primitives similar to those of CSP [Ho78]; processes communicate via *channels*, which are essentially message buffers of length 0. The communication primitives are

- B;e!α—send the value of expression e along channel α, provided that guard predicate B is enabled and there is a corresponding receive command ready.
- B;v?α—receive a value along channel α and store it in variable v, provided that the guard predicate B is enabled and there is a corresponding send command ready.

As in CSP, we assume that message transmission occurs as a single, synchronous event, with sender and receiver simultaneously executing the send, resp. receive primitive.

Remark. For programs in one of the above concrete frameworks, we use atomic propositions such as atl_i^j to indicate that, in the present state, process i is at location l_j.

5.4 Connecting the Concurrent Computation Framework with Temporal Logic

For an abstract concurrent program (M,ϕ_{START},Φ) and Temporal Logic formula p we write (M,ϕ_{START},Φ) $\models$ p and read it precisely (and a bit long-windedly) as "for program text M with starting condition ϕ_{START} and fair scheduling constraint Φ, formula p holds true;" the technical definition is as follows.

(i) in the linear time framework:
(M,ϕ_{START},Φ) $\models$ p iff $\forall$x in M such that M,x $\models \phi_{START}$ and M,x $\models \Phi$, we have M,x $\models$ p

(ii) in the branching time framework:
(M,ϕ_{START},Φ) $\models$ p iff $\forall$s in M such that M,s $\models \phi_{START}$ we have M,s $\models p_\Phi$,
where p_Φ is the branching time formula obtained from p by relativizing all path quantification to scheduling constraint Φ; i.e., by replacing (starting at the innermost subformulae and working outward) each subformula Aq by A($\Phi \Rightarrow$ q) and Eq by E($\Phi \wedge$ q).

6 Theoretical Aspects of Temporal Logic

In this section we discuss the work that has been done in the Computing Science community on the more purely theoretical aspects of Temporal Logic. This work has tended to focus on decidability, complexity, axiomitizability, and expressiveness issues. Decidability and complexity refer to natural decision problems associated with a system of Temporal Logic including (i) satisfiability—given a formula, does there exist a structure that is a model of the formula?, (ii) validity—given a formula, is it true that every structure is a model of the formula?, and (iii) model checking—given a formula together with a particular finite structure, is the structure a model of the formula? (Note: a formula is valid iff its negation is not satisfiable, so satisfiability and validity are, in effect, equivalent problems.) Axiomitizability refers to the question of the existence of deductive systems for proving all the valid formulae of a system of Temporal Logic, and the investigation of their soundness and completeness properties. Expressiveness concerns what correctness properties can and cannot be formulated in a given logic. The bulk of theoretical work has thus been to analyze, classify, and compare various systems of Temporal Logic with respect to these criteria, and to study the tradeoffs between them. We remark that these issues are not only of intrinsic interest, but are also significant due to their implications for mechanical reasoning applications.

6.1 Expressiveness

6.1.1 Linear Time Expressiveness

It turns out that PLTL has intimate connections with formal language theory. This connection was first articulated in the literature by Wolper who argued in [Wo83] that PLTL "is not sufficiently expressive":

Theorem 6.1. The property G_2Q, meaning that "at all even times (0,2,4,6,...etc.), Q is true," is not expressible in PLTL.

To remedy this shortcoming Wolper [Wo83] suggested the use of an extended logic based on grammar operators; for example, the grammar

$$V_0 \rightarrow Q;\ \mathit{true};\ V_0$$

defines the set of models of G_2Q. This relation with formal languages is discussed in more detail subsequently.

Quantified PLTL

Another way to extend PLTL is to allow quantification over atomic propositions (cf. [Wo82], [Si83]). The syntax of PLTL is augmented by the formation rule:

if p is a formula and Q is an atomic proposition occurring free in p,
then $\exists$Qp is a formula also.

The semantics of $\exists$Qp is given by

M,x $\models \exists$Qp iff there exists a linear structure M$'$ = (S,x,L$'$) such that M$'$,i $\models$ p where M = (S,x,L) and L$'$ differs from L in at most the truth value of Q.

The formula $\exists$Qp thus represents existential quantification over Q; since, under the interpretation M, Q may be viewed as defining an infinite sequence of truth values, one for every state s along x, this is a type of *2nd order* quantification. We use $\forall$Qp to abbreviate $\neg\exists$Q $\neg$p; of course, it means universal quantification over Q.

The extended temporal operator G_2Q can be defined in QPLTL:

$$G_2Q \equiv_i \exists Q'(Q' \wedge X\neg Q' \wedge G(Q' \Leftrightarrow XXQ') \wedge G(Q' \Rightarrow Q)$$

It can be shown that QPLTL coincides in expressive power with a number of formalisms from language theory including the just discussed grammar operators of [Wo83].

6.1.2 Monadic Theories of Linear Ordering

The First-Order Language of Linear Order (FOLLO) is that formal system corresponding to the "Right-Hand-Side" of the definitions of the basic temporal operators of PLTL. A formula of FOLLO is interpreted over a linear time structure (S,<); for our purposes, we as usual only consider (|N,<) or (I,<) where I is an initial segment of |N. The language of linear order is built from the following symbols:

P, Q, ...etc. denoting monadic (1 argument) predicate symbols (and intuitively corresponding to atomic propositions),

t, u, ...etc. denoting individual variables (and intuitively ranging over moments of time in |N), and

< — the distinguished *less than* symbol (representing the temporal ordering)

A linear time structure M=(S,x,L) is then defined just as for PLTL; note that L may be viewed as assigning to each monadic predicate symbol in AP the set of times at which it is true.

The formulae of FOLLO are those generated by the following rules:

LO0: If t,u are individual variables then $t < u$ is a formula

LO1: If P is a monadic predicate symbol and t is an individual variable, then P(t) is a formula

LO2: If p,q are formulae then so are $p \wedge q$, $p \vee q$, $\neg p$

LO3: If p is a formula and t is a free individual variable in p, then $\exists t(p)$ is a formula

The Second Order Language of Linear Order (SOLLO) is obtained by using the following additional rule:

LO4: If Q is a monadic predicate symbol not in AP that appears free in formula p then $\exists Qp$ is a formula

The semantics of SOLLO and FOLLO are defined in the obvious way. The results depicted below indicate how the expressive powers of the variants of PLTL and the Theories of Linear Ordering compare:

Theorem 6.2. As measured with respect to initial equivalence, the relative expressive power of these linear time formalisms is as depicted below:

PLTLF $\equiv_i$ PLTLB $\equiv_i$ FOLLO $<_i$ SOLLO $\equiv_i$ QPLTLB $\equiv_i$ QPLTLF.

For the sake of thoroughness, we include

Theorem 6.3. As measured with respect to global equivalence, the relative expressive power of these linear time formalisms is as depicted below, where any two logics not connected by a chain of $\equiv_g$'s and $<_g$'s are of incomparable expressive power:

$$\begin{array}{ccccc} & & \mathrm{PLTLB} \equiv_{\mathrm{g}} \mathrm{FOLLO} & & \\ & / & & \backslash & \\ \mathrm{PLTLF} <_{\mathrm{g}} & & & & <_{\mathrm{g}} \mathrm{SOLLO} \equiv_{\mathrm{g}} \mathrm{QPLTLB} \\ & \backslash & & / & \\ & & \mathrm{QPLTLF} & & \end{array}$$

Note that QPLTLB (respectively, QPLTLF) denotes the version of PLTL with quantification over auxiliary propositions having both past and future tense temporal operators (respectively, future tense temporal operators only).

6.1.3 Regular Languages and PLTL

There is an intimate relationship between languages definable in (variants and extensions of) PLTL, the monadic theories of linear ordering, and the regular languages. We will first consider languages of finite strings, and then languages of infinite strings. In the sequel let Σ be a finite alphabet. For simplicity, we further assume Σ = PowerSet(AP) for some set of atomic propositions AP. Moreover, we assume that the empty string λ is excluded so that the languages of finite strings are subsets of Σ^+, rather than Σ^*.

Languages of Finite Strings

Before presenting the results we briefly review regular expression notations and certain concept concerning finite state automata. The reader is referred to [Th89] for more details.

There are several types of regular expression notations:

- the *restricted regular expressions* which are those built up from the alphabet symbols σ, for each $\sigma \in \delta$, and $\bullet$, $\cup$, and $*$, denoting "concatenation," "union," and "kleene (or star) closure" respectively.
- the *general regular expressions* which are those built up from the alphabet symbols σ for each $\sigma \in \Sigma$ and $\bullet$, $\cap$, $\neg$, $\cup$, $*$ denoting "concatenation," "intersection," "complementation (with respect to Σ^*)," "union," and "kleene (or star) closure" respectively.
- the *star-free regular expressions* are those general regular expressions with no occurrence of $*$.

The restricted regular expressions are equivalent in expressive power to the general regular expressions; however, the star-free regular expressions are strictly less expressive.

A finite state automaton M=(Q,Σ,δ,q_0,F) is said to be *counter-free* iff it is *not* the case that there exist distinct states $q_0,\ldots,q_{k-1} \in Q$, $k \geq 1$, and a word $w \in \Sigma^+$ such that $q_{i+1 \bmod k} \in \delta(q_i,w)$. A language L is said to be *noncounting* iff it is accepted by some counter-free finite state automaton. Intuitively, a counter-free automaton cannot count modulo n for any $n \geq 2$. It is also known that the noncounting languages coincide with those expressible by star-free regular expressions. We now have the following results:

Theorem 6.4. The following are equivalent conditions on languages L of finite strings:

(a) $L \subseteq \Sigma^+$ is definable in PLTL

(b) $L \subseteq \Sigma^+$ is definable in FOLLO

(c) $L \subseteq \Sigma^+$ is definable by a star-free regular expression

(d) $L \subseteq \Sigma^+$ is definable by a counter-free finite state automaton

This result thus accounts for why Wolper's property G_2P is not expressible in PLTL, for it requires counting modulo 2. The following result also suggests why his regular grammar operators suffice:

Theorem 6.5. The following are equivalent conditions for languages L of finite strings:

(a) $L \subseteq \Sigma^+$ is definable in QPLTL

(b) $L \subseteq \Sigma^+$ is definable in SOLLO

(c) $L \subseteq \Sigma^+$ is definable by a regular expression

(d) $L \subseteq \Sigma^+$ is definable by a finite state automaton

The equivalence of conditions (b), (c), and (d) was established using lengthy and difficult arguments in the monograph of McNaughton & Pappert [MP62]. The equivalence of conditions (a) and (b) in Theorem 6.4 was established in Kamp [Ka68], while for Theorem 6.5 it was established in [LPZ85]. Direct translations between PLTL and star-free regular expressions were given in [Zu86].

Remark: Since we have past tense operators, it is natural to think of history variables. If $x = (s_0, s_1, s_2, \ldots)$ is a computation then the most general history variable h would be that which at time j has accumulated the complete history $s_0 \ldots s_j$ up to (and including) time j. The expressive power of a language with history variables depends on the type of predicates we may apply to the history variables. One natural type of history predicate is of the form $[\alpha]_H$ where α is a (star-free) regular expression, with semantics given by

$(x,i) \models [\alpha]_H$ iff $s_0 \ldots s_i$ considered as a string over $\Sigma = \mathrm{PowerSet}(AP)$
is in the language over Σ denoted by α

These history variables will be helpful in describing canonical forms for languages of infinite strings subsequently.

Languages of Infinite Strings

In extending the notion of regular language to encompass languages of infinite strings, the principle concern is how to finitely describe an infinite string. For finite state automata this is done using an extended notion of acceptance involving repeating a designated set of states infinitely often. See [Th89]. The framework of regular expressions can be similarly extended, in one of two ways:

(i) by adding an infinite repetition operator: ω. If α is an (ordinary) regular expression, then α^ω represents all strings of the form $a_1\ a_2\ a_3 \ldots$, where each $a_i \in \alpha$.

(ii) by adding a limit operator: lim. If α is an ordinary regular expression, then lim α consists of all those strings in Σ^ω which have infinitely many (distinct) finite prefixes in α.

We now have the two results below which follow from an assemblage of results in the literature (cf. [Ka68], [LPZ85], [Th89]):

Theorem 6.6 The following are equivalent conditions for languages L of infinite strings:

(a) $L \subseteq \Sigma^\omega$ is definable in QPLTL

(b) $L \subseteq \Sigma^\omega$ is definable in SOLLO

(c_0) $L \subseteq \Sigma^\omega$ is definable by an ω-regular expression, i.e. an expression of the form $\cup_{i=1}^m \alpha_i \beta_i^\omega$ where α_i, β_i are regular expressions

(c_1) $L \subseteq \Sigma^\omega$ is definable by an ω-limit regular expression, i.e. an expression of the form $\cup_{i=1}^m \alpha_i$ lim β_i where α_i, β_i are regular expressions

(c_2) $L \subseteq \Sigma^\omega$ is representable as $\cup_{i=1}^m$ (lim $\alpha_i \cap \neg$lim β_i) where α_i, β_i are regular expressions

(c_3) $L \subseteq \Sigma^\omega$ is expressible as $\vee_{i=1}^m (\overset{\infty}{F} [\alpha_i]_H \wedge \neg\overset{\infty}{F} [\beta_i]_H)$ where α_i, β_i are regular expressions

For the case of the star-free ω-languages we have

Theorem 6.7. The following are equivalent conditions for languages L of infinite strings:

(a) $L \subseteq \Sigma^\omega$ is definable in PLTL

(b) $L \subseteq \Sigma^\omega$ is definable in FOLLO

(c_1) $L \subseteq \Sigma^\omega$ is definable by an ω-regular expression, i.e. an expression of the form $\cup_{i=1}^m \alpha_i$ lim β_i where α_i, β_i are star-free regular expressions

(c_2) $L \subseteq \Sigma^\omega$ is representable as $\cup_{i=1}^m$ (lim $\alpha_i \cap \neg$lim β_i) where α_i, β_i are star-free regular expressions

(c_3) $L \subseteq \Sigma^\omega$ is expressible in the form $\vee_{i=1}^m (\overset{\infty}{F} [\alpha_i]_H \wedge \neg\overset{\infty}{F} [\beta_i]_H)$ where α_i, β_i are star-free regular expressions.

Result 6.7 (c_0) analogous to Result 6.6 (c_0) was intentionally omitted—because it does not hold as noted in [Th79]. It is not the case that $\cup_{i=1}^m \alpha_i \beta_{i=1}^\omega$ where α_i, β_i are star-free regular expressions, must itself denote a star-free regular set. For example, consider the language $L = (00 \cup 1)^\omega$. L is expressible as a union of $\alpha_i \beta_i^\omega$: take $m = 1$, $\alpha_i = \epsilon$, $\beta_1 = 00 \cup 1$. But L, which consists intuitively of exactly those strings for which there is an even number of 0's between every consecutive pair of 1's, is not definable in FOLLO, nor is it star-free regular.

Remark: One significant issue we do not address here in any detail — and which is not very thoroughly studied in the literature — is that of *succinctness*. Here we refer to how long or short a formula is needed to capture a given correctness property. Two formalisms may have the same raw expressive power, but one may be much more succinct than the other. For example, while FOLLO and PLTL have the same raw expressive power, it is known that FOLLO can be significantly (nonelementarily) more succinct than PLTL (cf. [Me74]).

6.1.4 Branching Time Expressiveness

Analogy with the linear temporal framework suggests several formalisms for describing infinite trees that might be compared with branching temporal logic. Among these are: finite state automata on infinite trees, the monadic second order theory of many successors (SnS), and the monadic second order theory of partial orders. However, not nearly so much is known about the comparison with related formalisms in the branching time case.

One difficulty is that, technically, the types of branching objects considered differ. Branching Temporal Logic is interpreted over structures which are, in effect, trees with nodes of infinite outdegree, whereas, e.g., tree automata take input trees of fixed finite outdegree. Another difficulty is that the logics, such as CTL*, as ordinarily considered, do not distinguish between, e.g., left and right successor nodes, whereas the tree automata can.

To facilitate a technical comparison, we therefore restrict our attention to (a) structures corresponding to infinite binary trees and (b) tree automata with a "symmetric" transition function that do not distinguish, e.g., left from right. Then we have the following result from [ESi84] comparing logics augmented with existential quantification over atomic propositions with tree automata.

Theorem 6.8.

(i) EQCTL* is exactly as expressive as symmetric pairs automata on infinite trees

(ii) EQCTL is exactly as expressive as symmetric Buchi automaton infinite trees.

Here, EQCTL* consists of the formula $\exists Q_1 \ldots \exists Q_m f$, where f is a CTL* formula and the Q_i are atomic propositions appearing in f. The semantics is that, given a structure M=(S,R,L), $M,s \models \exists Q_1 \ldots \exists Q_m f$ iff there exists a structure M'=(S,R,L') such that $M',s \models f$ and L' differs from L at most in the truth assignments to each Q_i, $1 \leq i \leq m$. Similarly, EQCTL consists of formulae $\exists Q_1 \ldots Q_m f$, where f is a CTL formula.

A related result is from [HT87]:

Theorem 6.9. CTL* is exactly as expressive as the monadic second order theory of two successors with set quantification restricted to infinite paths, over infinite binary trees.

Remark: By augmenting CTL* with arc assertions which allow it to distinguish outgoing arc i from arc i+1 the result extends to infinite n-ary trees, $n > 2$. By taking $n = 1$, the result specializes to the "expressive completeness" result of Kamp [Ka68] that PLTL is equivalent in expressive power to FOLLO (our Theorem 6.7 (a,b)).

While less is known about comparisons of BTLs (Branching Time Logics) against external "yardsticks," a great deal is known about comparisons of BTLs against each other. This contrasts with the reversed situation for LTLs (Linear Time Logics). Perhaps this reflects the much greater degree of "freedom" due to the multiplicity of alternative futures found in the BTL framework.

It is useful to define the notion of a *basic modality* of a BTL. This is a formula of the form Ap or the form Ep, where p is a pure linear time formula (containing no path quantifiers.) Then a formula of a logic may be seen as being built up by combining basic modalities using boolean connectives and nesting. For example, EFP is a CTL basic modality; so is AFQ. EFAFQ is formula of CTL (but not a basic modality) obtained by nesting AFQ within EFP (more precisely, by substituting AFQ for P within EFP). E(FP $\wedge$ FQ) is a basic modality of CTL*, but not a basic modality nor a formula of CTL.

A large number of sublanguages of CTL* can be defined by controlling the way the linear time operators combine using boolean connectives and nesting of operators in the basic modalities of the language. For instance, we use B(F,X,U) to indicate the language where only a single linear time operator X, F, or U can follow a path quantifier, and B(F,X,U,$\wedge$,$\neg$) to indicate the language where boolean combinations of these linear operators are allowed, but not nesting of the linear operators. Thus formula E(Fp $\wedge$ Gq) is in the language B(F,X,U,$\wedge$,$\neg$) but not in B(X,F,U).

The diagram in Figure 3 shows how some of these logics compare in expressive power. The notation $L_1 < L_2$ means that L_1 is strictly less expressive than L_2, which holds provided that

(a) $\forall$ formula p of L_1 $\exists$ a formula q of L_2 such that $\forall$ structure M $\forall$ state s in M, M,s $\models$ p iff M,s $\models$ q, and

(b) the converse of (a) does not hold,

while $L_1 \equiv L_2$ means L_1 and L_2 are equivalent in expressive power, and $L_1 \leq L_2$ means $L_1 < L_2$ or $L_1 \equiv L_2$.

Most of the logics shown are known from the literature. B(F) is the branching time logic of Lamport [La80], having basic modalities of the form A or E followed by F or G. The logic B(X,F), which has basic modalities of the form A or E followed by X, F, or G, was originally proposed in [BPM82] as the logic UB. The logic B(X,F,U) is of course CTL. The logic B(X,F,U,$\overset{\infty}{F}$,$\wedge$,$\neg$) is essentially the logic proposed in [EC80]; its infinitary modalities $\overset{\infty}{F}$ and $\overset{\infty}{G}$ permit specification of fairness properties.

We now give some rough, high-level intuition underlying these results. Semantic containment along each edge follows directly from syntactic containment in all cases, except edges 2 and 4, which follow given the semantic equivalence of edge 3 (discussed below).

The X operator (obviously) cannot be expressed in terms of the F operator, which accounts for edge 0, B(F) $\leq$ B(F,X). Similarly, the U operator cannot be expressed in terms of X, F, and boolean connectives. This was known "classically" (cf. [Ka68]), and accounts for edge 2: B(X,F,$\wedge$,$\neg$) < B(X,F,U).

To establish the equivalence of edge 3, we need to provide a translation of B(X,F,U,$\wedge$,$\neg$) into B(X,F,U). The basic idea behind this translation can be understood by noting that E(FP $\wedge$ FQ) $\equiv$ EF(P $\wedge$ EFQ) $\vee$ EF(Q $\wedge$ EFP). However, it is a bit more subtle than that; the ability to do the translation in all cases depends on the presence of the until (U) operator (cf. edge 1). The following validities, two of which concern the until, can be used to inductively translate each B(X,F,U,$\wedge$,$\neg$) formula into an equivalent B(X,F,U) formula:

$$\mathrm{E}(\ (p_1\ \mathrm{U}\ q_1) \wedge (p_2\ \mathrm{U}\ q_2)\ ) \equiv$$
$$\quad \mathrm{E}(\ (p_1 \wedge p_2)\ \mathrm{U}\ (q_1 \wedge \mathrm{E}(p_2\ \mathrm{U}\ q_2))) \vee \mathrm{E}(\ (p_2 \wedge p_1)\ \mathrm{U}\ (q_2 \wedge \mathrm{E}(p_1\ \mathrm{U}\ q_1)))$$
$$\mathrm{E}(\neg(p\ \mathrm{U}\ q)) \equiv \mathrm{E}(\ (\neg q \wedge \neg p)\ \mathrm{U}\ (q \wedge p)\ ) \vee \mathrm{EG}\neg q$$
$$\mathrm{E}(\neg \mathrm{X}p) \equiv \mathrm{EX}\neg p$$

$E\overset{\infty}{F}Q$, a $B(X,F,U,\overset{\infty}{F})$ formula is not expressible in $B(X,F,U)$ accounting for the strict containment on arc 4. This is probably the most significant result, for it basically says that correctness under fairness assumptions cannot be expressed in a BTL with a simple set of modalities. For example, the property that P eventually becomes true along all fair computations (fair inevitability of P) is of the form $A(\overset{\infty}{F}Q \Rightarrow FP)$ for even a (very) simple fairness constraint like $\overset{\infty}{F}Q$. Neither it, nor its dual $E(\overset{\infty}{F}Q \wedge GP)$, is expressible in $B(X,F,U)$, since by taking P to be *true* the dual becomes $E\overset{\infty}{F}Q$.

The inexpressibility of $E\overset{\infty}{F}Q$ was established in [EC80], using recursion-theoretic arguments to show that the predicate transformer associated with $E\overset{\infty}{F}Q$ is Σ_1^1-complete while the predicate transformers for $B(X,F,U)$ are arithmetical. The underlying intuition is that $E\overset{\infty}{F}Q$ uses second order quantification in an essential way to assert that that there exists a sequence of nodes in the computation tree where Q holds. Another version of this inexpressiveness result was established by Lamport [La80] in a somewhat different technical framework. Still another proof of this result was given by Emerson and Halpern [EH86]. The type of inductive, combinatorial proof used is paradigmatic of the proofs of many inexpressiveness results for TL, so we describe the main idea here.

Theorem 6.10. $E\overset{\infty}{F}Q$ is not expressible in $B(X,F,U)$

Proof Idea. We inductively define two sequences $M_1,M_2,M_3,\ldots$ and $N_1,N_2,N_3,\ldots$ of structures as shown in Figure 4. It is plain that for all i,

(*) $M_i,s_i \models E\overset{\infty}{F}Q$ and $N_i,s_i \models \neg E\overset{\infty}{F}Q$

Thus $E\overset{\infty}{F}Q$ distinguishes between the two sequences. However, we can show by an inductive argument that each formula of $B(X,F,U)$ is "confused" by the two sequences, in that

(**) $M_i,s_i \models p$ iff $N_i,s_i \models p$

If some formula p of $B(X,F,U)$ were equivalent to $E\overset{\infty}{F}Q$, we would then have for i = the length of p that

$M,s_i \models p$ and $N,s_i \models \neg p$ by virtue of (*)
and also that
$N,s_i \models p$, by virtue of (**), a contradiction. □

The strict containment along the rest of the edges follow from these inexpressiveness results: $E(FP \wedge GQ)$ is not expressible in $B(X,F)$, for edge 1. $E(\overset{\infty}{F}P_1 \wedge \overset{\infty}{F}P_2)$ is not expressible in $B(X,F,U,\overset{\infty}{F})$, for edge 5. $A(F(P \wedge XP))$ is not expressible in $B(X,F,U,\overset{\infty}{F},\wedge,\neg)$, for edge 6. The proofs are along the lines of the theorem above for $E\overset{\infty}{F}Q$.

It is also possible to compare branching with linear time logics. When a linear time formula is interpreted over a program, there is usually an implicit universal quantification over all possible computations. This suggests that when given a linear time language L, which is of course a set of path formulae, we convert it into a branching time language by prefixing each path formula by the universal path quantifier A. We thus get the corresponding branching language BL(L) = { Ap: p ∈ L}. Figure 5 shows how various branching and linear logics compare. Not surprisingly, the major limitation of linear time is its inability to express existential path quantification (cf. [La80], [EH86]).

Theorem 6.11. The formula EFP is not expressible in any BL(—) logic.

6.2 Decision Procedures for Propositional Temporal Logics

In this section we discuss algorithms for testing if a given formula p_0 in a system of propositional TL is satisfiable. The usual approach to developing such algorithms is to first establish the *small model property* for the logic: if a formula is satisfiable, then it is satisfiable in a "small" finite model, where "small" means of size bounded by some function, say, f, of the length of the input formula. This immediately yields a decision procedure for the logic. Guess a small structure M as a candidate model of given formula p_0; then check that M is indeed a model of p_0. This check can be done by exhaustive search, since M is finite, and can often be done efficiently.

An elegant technique for establishing the small model property is through use of the *quotient construction*, also called—in classical model logic—*filtration*, where an equivalence relation of small finite index is defined on states. Then equivalent states are identified to collapse a possibly infinite model to a small finite one.

An example of a quotient construction is its application to yield a decision procedure for Propositional Dynamic Logic of [FL79], discussed in [KT89]. There the equivalence relation is defined so that, in essence, two states are equivalent when they agree (i.e., have the same truth value) on all subformulae of the formula p_0 being tested for satisfiability. This yields a decision procedure of nondeterministic exponential time complexity, calculated as follows. The total complexity is the time to guess a small candidate model plus the time to check that it is indeed a model. The candidate model can be guessed in time polynomial in its size which is exponential in the length of p_0, since for a formula of length n there are about n subformulae and 2^n equivalence classes. And it turns out that checking that the candidate model is a genuine model can be done in polynomial time.

Of course the deterministic time complexity of the above algorithm is double exponential. The complexity can be improved through use of the tableau construction.

A *tableau* for formula p_0 is a finite directed graph with nodes labelled by subformulae associated with p_0 that, in effect, encodes all potential models of p_0. In particular, as in the case of Propositional Dynamic Logic, the tableau contains as a subgraph the quotient structure corresponding to any model of p_0. The tableau can be constructed, and then tested for consistency to see if it contains a genuine quotient model. Such testing can often be done efficiently. In the case of of Propositional Dynamic Logic, the tableau is of size exponential in the formula length, while the testing can be done in deterministic polynomial time in the tableau size, yielding a deterministic single exponential time decision procedure.

For some logics, no matter how we define a finite index equivalence relation on states, the quotient construction yields a quotient structure that is not a model. However, for many logics, the quotient structure still provides useful information. It can be viewed as a "pseudo-model" that can be unwound into a genuine, yet still small, model. The tableau construction, moreover, can still be used to perform a systematic search for a pseudo-model, to be unwound into a genuine model.

We remark that the tableau construction is a rather general one, that applies to many logics. Tableau-based decision procedures for various logics are given in [Pr79], [BPM81], [BHP82], [Wo82], [Wo83], [HS84]. See also the excellent survey by Wolper [Wo84]. In the sequel we describe a tableau-based decision procedure for CTL formulae, along the lines of [EC82] and [EH85]. The following definitions and terminology are needed.

We assume that the candidate formula p_0 is in *positive normal form*, obtained by pushing negations inward as far as possible using de Morgan's laws ($\neg(p \vee q) \equiv \neg p \wedge \neg q$, $\neg(p \wedge q) \equiv \neg p \vee \neg q$) and dualities ($\neg AGp \equiv EF\neg p$, $\neg A[p\ U\ q] \equiv E[\neg p\ B\ q]$, etc.). This at most doubles the length of the formula, and results in only atomic propositions being negated. We write $\sim$p for the formula in positive normal form equivalent to $\neg$p. The *closure* of p_0, $cl(p_0)$, is the least set of subformulae such that:

- Each subformulae of p_0, including p_0 itself, is a member of $cl(p_0)$;
- If EFq, EGq, E[p U q], or E[p B q] $\in cl(p_0)$ then, respectively, EXEFq, EXEGq, EXE[p U q], or EXE[p B q] $\in cl(p_0)$;
- If AFq, AGq, A[p U q], or A[p B q] $\in cl(p_0)$ then, respectively, AXAFq, AXAGq, AXA[p U q], or AXA[p B q] $\in cl(p_0)$;

The *extended closure* of p_0, $ecl(p_0) = cl(p_0) \cup \{\sim p\colon p \in cl(p_0)\}$. Note that $card(ecl(p_0)) = O(length(p_0))$.

At this point we give the technical definitions for the quotient construction, as they are needed in the proof of the small model theorem of CTL. We also show the the quotient construction by itself is inadequate for getting a small model theorem for CTL.

Let M=(S,R,L) be a model of p_0, let H be a set of formulae, and let $\equiv_H$ be an equivalence relation on S induced by agreement on the formulae in H, i.e. $s \equiv_H t$ whenever $\forall\, q \in H$, $M,s \models q$ iff $M,t \models q$. We use [s] to denote the equivalence class $\{t\colon t \equiv_H s\}$ of s. Then the quotient structure of M by $\equiv_h$, $M/\equiv_H$, = (S′,R′,L′) where $S' = \{[s]\colon s \in S\}$, $R' = \{([s],[t])\colon (s,t) \in R\}$, and $L'([s]) = L(s) \cap H$. Ordinarily, we take $H = ecl(p_0)$.

However, as the following theorem shows, no way of defining the equivalence relation for the quotient construction preserves modelhood:

Theorem 6.12. For every set H of (CTL) formulae, the quotient construction does not preserve modelhood for the formula AFP. In particular, there is a model M of AFP such that for every finite set H, $M/\equiv_H$ is not a model for AFP.

Proof Idea. Note the structure shown in Figure 6(a) is a model of AFP. But however the quotient relation collapses the structure two distinct states s_i and s_j will be identified, resulting in a cycle in the quotient structure, along which P is always false, as suggested in Figure 6(b). Hence AFP does not hold along the cycle. □

We now proceed with the technical development needed. To simplify the exposition, we assume that the candidate formula p_0 is of the form $p_1 \wedge AGEX\mathit{true}$, syntactically reflecting the semantic requirement that each state in a structure have a successor state.

We say that a formula is *elementary* provided that it is a proposition, the negation of a proposition, or has main connective AX or EX. Any other formula is *nonelementary*. Each nonelementary formula may be viewed as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. Clearly, $f \wedge g$ is an α formula and $f \vee g$ is a β formula. A modal formula may be classified as α or β based on its fixpoint characterization (cf. section 8.4); e.g., $EFp = p \vee EXEFp$ is a β formula and $AGp = p \wedge AXAGp$ is an α formula. The following table summarizes the classification:

$\alpha = p \wedge q$	$\alpha_1 = p$	$\alpha_2 = q$
$\alpha = A[p\ B\ q]$	$\alpha_1 = \sim q$	$\alpha_2 = p \vee AXA[p\ B\ q]$
$\alpha = E[p\ B\ q]$	$\alpha_1 = \sim q$	$\alpha_2 = p \vee EXE[p\ B\ q]$
$\alpha = AGq$	$\alpha_1 = q$	$\alpha_2 = AXAGq$
$\alpha = EGq$	$\alpha_1 = q$	$\alpha_2 = EXEGq$
$\beta = p \vee q$	$\beta_1 = p$	$\beta_2 = q$
$\beta = A[p\ U\ q]$	$\beta_1 = q$	$\beta_2 = p \wedge AXA[p\ U\ q]$
$\beta = E[p\ U\ q]$	$\beta_1 = q$	$\beta_2 = p \wedge EXE[p\ U\ q]$
$\beta = AFq$	$\beta_1 = q$	$\beta_2 = AXAFq$
$\beta = EFq$	$\beta_1 = q$	$\beta_2 = EXEFq$

A formula of the form A[p U q] or E[p U q] is an *eventuality* formula. An eventuality makes a promise that something will happen. This promise must be *fulfilled*. The eventuality A[p U q] (E[p U q]) is fulfilled for s in M provided that for every (respectively, for some) path starting at s, there exists a finite prefix of the path in M whose last state is labelled with q and all of whose other states are labelled with p. Since AFq and EFq are special cases of A[p U q] and E[p U q], respectively, they are also eventualities. In contrast, A[p B q], E[p B q], and their special cases AGq and EGq, are *invariance* formulae. An invariance property asserts that whatever happens to occur (if anything) will meet certain conditions (cf. subsection 7.1).

We say that a *prestructure* M is a triple (S,R,L) just like a structure except that the binary relation R is not required to be total. An *interior* node of a prestructure is one with at least one successor. A *frontier* node is one with no successors.

It is helpful to associate certain consistency requirements on the labelling of a (pre)structure:

Propositional Consistency Rules:

PC0 $\sim$p $\in$ L(s) implies p $\notin$ L(s)
PC1 $\alpha \in$ L(s) implies $\alpha_1 \in$ L(s) and $\alpha_2 \in$ L(s)
PC2 $\beta \in$ L(s) implies $\beta_1 \in$ L(s) or $\beta_2 \in$ L(s)

Local Consistency Rules:

LC0 AXp $\in$ L(s) implies $\forall$ successor t of s, p $\in$ L(t)
LC1 EXp $\in$ L(s) implies $\exists$ successor t of s, p $\in$ L(t)

A *fragment* is a prestructure whose graph is a dag (directed acyclic graph) such that all of its nodes satisfy PC0-2 and LC0 above, and all of its interior nodes satisfy LC1 above.

A *Hintikka structure (for p_0)* is a structure M=(S,R,L) (with $p_0 \in$ L(s) for some s $\in$ S) which meets the following conditions:

1. the propositional consistency rules PC0-2,
2. the local consistency rules LC0-1, and
3. each eventuality is fulfilled.

Proposition 6.13. If structure M = (S,R,L) defines a model of p_0 and each state s is labelled with exactly the formula in ecl(p_0) true at s, then M is a Hintikka structure for p_0. Conversely, a Hintikka structure for p_0 defines a model of p_0.

If M is a Hintikka structure, then for each node s of M and each eventuality r in ecl(p_0) such that M,s $\models$ r, there is a fragment, call it DAG[s,r], which certifies fulfillment of r at s in M. What is the nature of this fragment? It has s as its root, i.e., node from which all other nodes in DAG[s,r] are reachable. If r is of the form AFq, then DAG[s, AFq] is obtained by taking node s and all nodes along all paths emanating from s up to and including the first state where q is true. The resulting subgraph is indeed a dag all of whose frontier nodes are labelled with q. If r were of the form A[p U q], DAG[s, A[p U q]] would be the same except its interior nodes are all labelled with p. In the case of DAG[s, EFq] take a shortest path leading from node s to a node labelled with q, and then add sufficient successors to ensure that LC1 holds of each interior node on the path. In the case of DAG[s, E[p U q]], the only change is that p labels each interior node on the path.

In a Hintikka structure M for p_0, each fulfilling fragment DAG[s, r] for each eventuality r, is "cleanly embedded" in M. If we collapse M by applying a finite index quotient construction, the resulting quotient structure is not, in general, a model because cycles are introduced into such fragments. However, there is still a fragment, call it DAG′[s,r], "contained" in the quotient structure of M. It is simply no longer cleanly embedded. Technically, we say prestructure $M_1 = (S_1,R_1,L_1)$ is *contained* in prestructure $M_2 = (S_2,R_2,L_2)$ whenever $S_1 \subseteq S_2$, $R_1 \subseteq R_2$, and $L_1 = L_2|S_1$, the labelling L_2 restricted to S_1. We say M_1 is *cleanly embedded* in M_2 provided M_1 is contained in M_2, and also every interior node of M_1 has the same set of successors in M_1 as in M_2.

A *pseudo-Hintikka structure (for p_0)* is a structure M=(S,R,L) (with $p_0 \in$ L(s) for some s $\in$ S) which meets the following conditions:

1. the propositional consistency rules PC0-2,

2. the local consistency rules LC0-1, and

3. each eventuality is *pseudo-fulfilled* in the following sense:

   AFq $\in$ L(s) (resp., A[p U q] $\in$ L(s))
   implies there is a finite fragment—called DAG[s, AFq] (resp., DAG[s, A[p U q]])—
   rooted at s contained in M such that
   for *all* frontier nodes t of the fragment, q $\in$ L(t)
   (resp., and for all interior nodes u of the fragment, p $\in$ L(u));

   EFq $\in$ L(s) (resp., E[p U q] $\in$ L(s))
   implies there is a finite fragment—called DAG[s, EFq] (resp., DAG[s, E[p U q]])—
   rooted at s contained in M such that
   for *some* frontier node t of the fragment, q $\in$ L(t)
   (resp., and for all interior nodes u of the fragment, p $\in$ L(u)).

Theorem 6.14. (Small Model Theorem for CTL) Let p_0 be a CTL formula of length n. Then the following are equivalent:

(a) p_0 is satisfiable
(b) p_0 has a infinite tree model with finite branching bounded by O(n)
(c) p_0 has a finite model of size $\leq$ exp(n)
(d) p_0 has a finite pseudo-Hintikka structure of size $\leq$ exp(n)

Proof Sketch: We show that (a) $\Rightarrow$ (b) $\Rightarrow$ (d) $\Rightarrow$ (c) $\Rightarrow$ (a).

(a) $\Rightarrow$ (b): Suppose M,s $\models p_0$. Then as described in subsection 5.1. M can be unwound into an infinite tree model M_1, with root state s_1 a copy of s. It is possible that M_1 has infinite branching at some states, so (if needed) we chop out spurious successor states to get a bounded branching subtree M_2 of M_1 such that still $M_2,s_1 \models p_0$. We proceed down M_1 level-by-level deleting all but n successors of each state. The key idea is that for each formula EXq $\in$ L(s), where s is a retained node on the current level, we keep a successor t of s of least q-rank, where the q-rank(s) is defined as the length of the shortest path from s fulfilling q, if q is of the form EFr or E[p U r], and is defined as 0 if q is of any other form. This will ensure that each eventuality of the form EFr or E[p U r] is fulfilled in the tree model M_2. Moreover, since there are at most O(n) formulae of the form EXq in ecl(p_0), the branching at each state of M_2 is bounded by O(n).

(b) $\Rightarrow$ (d): Let M be a bounded branching infinite tree model with root s_0, such that M,$s_0 \models p_0$. We claim that the quotient structure $M' = M/{\equiv_{ecl(p_0)}}$ is a pseudo-Hintikka structure. It suffices to show that for each state [s] of M′, and each eventuality r in the label of [s] there is a finite fragment contained in M′ certifying pseudo-fulfillment of r. We sketch the argument in the case r = AFq. The argument for other types of eventuality is similar.

So suppose AFq appears in the label of [s]. By definition of the quotient construction, in the original structure M AFq is true at state s, and thus there exists a finite fragment DAG[s, AFq] with root s cleanly embedded in M. Extract (a copy of) the fragment DAG[s, AFq]. Chop out states with duplicate labels. Given two states s, s′ with the same label, let the deeper state replace the shallower, where the depth of a state is the length of the longest path from the state back to the root s_0. This ensures that after the more shallow node has been chopped out, the resulting graph is still a dag, and moreover, a fragment. Since we can chop out any pair of duplicates the final fragment, call it DAG′[[s],AFq] has at most a single occurrence of each label. Therefore (a copy of) DAG′[[s], AFq] is contained in the quotient structure M′. It follows that M′ is a pseudo-Hintikka model as desired.

(d) $\Rightarrow$ (c): Let M = (S,R,L) be a pseudo-Hintikka model for p_0. For simplicity we identify a state s with its label L(s). Then for each state s and each eventuality q $\in$ s, there is a fragment DAG[s,q] contained in M certifying fulfillment of q. We show how to splice together copies of the DAGs, in effect unwinding M, to obtain a Hintikka model for p_0.

For each state s and each eventuality q, we construct a dag rooted at s, DAGG[s,q]. If q $\in$ s then DAGG[s,q] = DAG[s,q]; otherwise DAGG[s,q] is taken to be the subgraph consisting of s plus a sufficient set of successors to ensure that local consistency rules LC0-1 are met.

We now take (a single copy of) each DAGG[s,q] and arrange them in a matrix as shown in Figure 7, the rows range over eventualities $q_1,...q_m$ and the columns range over the states $s_1,...,s_N$ in the tableau. Now each frontier node s in row i is replaced by the copy of s that is the root of DAGG[s,q_{i+1}] in row i+1. Note that each fullpath through the resulting structure goes through each row infinitely often. As a consequence, the resulting graph defines a model of p_0, as can be verified by induction on the structure of formulae. The essential point is that each eventuality q_i is fulfilled along each fullpath where needed, at least by the time the fullpath has gone through row i.

The cyclic model consists of m·N DAGG's, each consisting of N nodes. It is thus of size m·N^2 nodes, where the number of eventualities $m \leq n$ and the number of tableau nodes $N \leq 2^n$, and n is the length of p_0. We can chop out duplicate nodes with the same label within a row, using an argument based on the depth of a node like that used above in the proof of (b) $\Rightarrow$ (d), to get a model of size m·N = exp(n).

(c) $\Rightarrow$ (a) is immediate. $\Box$

We now describe the tableau-based decision procedure for CTL. Let p_0 be the candidate CTL formula which is to be tested for satisfiability. We proceed as follows.

1. Build an initial *tableau* T=(S,R,L) for p_0, which encodes potential pseudo-Hintikka structures for p_0. Let S be the collection of all maximal, propositionally consistent subsets s of ecl(p_0), where by maximal we mean that for every formula p $\in$ ecl(p_0), either p or $\sim$p $\in$ s, while propositionally consistent refers to rules PC0-2 above. Let R $\subseteq$ S · S be defined so that (s,t) $\in$ R unless AXp $\in$ s and not(p) $\in$ t, for some formula AXp $\in$ ecl(p_0). Let L(s) = s. Note that the tableau as initially constructed meets all propositional consistency rules PC0-2 and local consistency rule LC0.

2. Test the tableau for consistency and pseudo-fulfillment of eventualities, by repeatedly applying the following deletion rules until no more nodes in the tableau can be deleted:

   - Delete any state s such that eventuality r $\in$ L(s) and there does *not* exist a fragment DAG[s,r] rooted at s contained in the tableau which certifies pseudo-fulfillment of r.
   - Delete any state which has no successors.
   - Delete any state which violates LC1.

Note that this portion of the algorithm must terminate, since there are only a finite number of nodes in the tableau.

3. Let T′ be the final tableau. If there exists a state s′ in T′ with $p_0 \in L(s')$ then return "YES, p_0 is satisfiable."; If not, then return "NO, p_0 is unsatisfiable".

To test the tableau for the existence of the appropriate fragments to certify fulfillment of eventualities we can use a ranking procedure. For an A[p U q] eventuality initially assign rank 1 to all nodes labelled with q and rank ∞ to all other nodes. Then for each node s and each formula r such that EXr is in the label of s, define $SUCC_r(s) = \{s''$: s′ is a successor of s in the tableau with $r \in$ label of $s''\}$ and compute $rank(SUCC_r(s)) = \min \{rank\ s': s'' \in SUCC_r(s)\}$. Now for each node s of rank $= \infty$ such that $p \in L(s)$ let $rank(s) = 1 + \max \{rank(SUCC_r(s): EXr \in L(s)\}$. Repeatedly apply the above ranking rules until stabilization. A node has finite rank iff A[p U q] is fulfilled at it in the tableau. To test for fulfillment of an AFq is a special case of the above, ignoring the formula p. To test for fulfillment of E[p U q] use a procedure like the above, but compute rank(s) $= 1 + \min \{rank(SUCC_r(s): EXr \in L(s)\}$. To test for fulfillment of EFq is again a special case, where the formula p is ignored.

Theorem 6.15. The problem of testing satisfiability for CTL is complete for deterministic exponential time.

Proof idea. The above algorithm can be shown to run in deterministic exponential time in the length of the input formula, since the size of the tableau is, in general, exponential in the formula size, and the tableau can be constructed and tested for containment of a pseudo-Hintikka structure in time polynomial in it size. This establishes the upper bound. The lower bound follows by a reduction from alternating polynomial space bounded Turing machines, similar to that used to establish exponential time hardness for Propositional Dynamic Logic (see [KT89]). □

The above formulation of the CTL decision procedure is sometimes known as the *maximal model* approach, since the nodes in the initial tableau are maximal, propositionally consistent sets of formulae and we put in as many arcs as possible. One drawback is that its average case complexity is as bad as its worst case complexity, since it always constructs the exponential size collection of maximal, propositionally consistent sets of formulae.

An alternative approach is to build the initial tableau incrementally, which in practice often results in a significant decrease in its size and time required to construct it. The tableau construction will now begin with a bipartite graph $T' = (C,D,R_{CD},R_{DC},L)$ where nodes in C are referred to as states while nodes in D are known as prestates; $R_{CD} \subseteq C \times D$ and $R_{DC} \subseteq D \times C$. The labels of the states will be *sparsely downward closed* sets of formulae in $ecl(p_0)$, i.e., sets which satisfy PC0, PC1, and PC2′: $\beta \in L(s)$ implies either $\beta_1 \in L(s)$ or $\beta_2 \in L(s)$.

Initially, let C = the empty set, D = a single prestate d labelled with p_0.

Repeat

Let e be a frontier node of T'

If e is a prestate d then

let $c_1,...,c_k$ be states whose labels comprise all the sparsely downward closed supersets of L(d)

add $c_1,...,c_k$ as R_{DC}-successors of d in T'

Note: if any c_i has the same label as another state c' already in T', then identify c_i and c' (i.e., delete c_i and draw an R_{DC}-arc from d to c'.)

If e is a state c labelled with nexttime formulae $AXp_1,...,AXp_l,EXq_1,...,EXq_k$ then

create prestates $d_1,...,d_k$ labelled with sets resp. $\{p_1,...,p_l,q_1\},..., \{p_1,...,p_l,q_k\}$ and add them as R_{CD}-successors to c in T'

Note: if any d_i has the same label as another prestate d' already in T', then identify d_i and d' as above

Until all nodes in T' have at least one successor

Now the tableau T = (C,R,L|C) where C is the set of states in T' above and $R = R_{CD} \cdot R_{DC}$, L|C is the labelling L restricted to C. Then the remainder of the decision procedure described previously can be applied to this new tableau constructed incrementally.

Remark: It is possible to construct the original type of tableau incrementally. Let the initial prestate be labelled with $p_0 \vee \sim p_0$ and use maximal, propositionally consistent sets for the labels of states.

The decision procedure for CTL also yields a deterministic exponential time decision procedure for PLTL:

Theorem 6.16. Let p_0 be a PLTL formula in positive normal form. Let p_1 be the CTL formula obtained from p_0 by replacing each temporal operator F, G, X, U, B by AF, AG, AX, AU, AB, resp. Then p_0 is satisfiable iff p_1 is satisfiable.

We can in fact do better for PLTL and various fragments of it. The following results on the complexity of deciding linear time are due to Sistla and Clarke [SC85]:

Theorem 6.17. The problem of testing satisfiability for PLTL is PSPACE-complete.

Proof Idea. To establish membership in PSPACE, we design a nondeterministic algorithm that, given an input formula p_0, guesses a satisfying path through the tableau for p_0 which defines a linear model of size exp(n), where n = length(p_0). This path can be guessed and verified to be a model in only O(n) space, since the algorithm need only remember the label of the current and next state along the path, and the point where the path loops back, in order to check that eventualities are fulfilled. PSPACE-hardness can be established by a generic reduction from polynomial space Turing machines. □

For the sublanguage of PLTL restricted to allow only the F operator (and its dual G), denoted PLTL(F) further improvement is still possible. We first establish the somewhat surprising

Theorem 6.18. (Linear Size Model Theorem for PLTL(F)) If PLTL(F) formula p_0 of length n is satisfiable, then it has a finite linear model of size O(n).

Proof idea. The important insight is that truth of a PLTL(F) formula only depends on the set of successor states, and not their order or arrangement. Now suppose p_0 is satisfiable. Let x =

s_0,s_1,s_2,... be a model of p_0. Then there exist i and j such that $i < j$ and $s_i = s_j$ and the set of states appearing infinitely often along x equals $\{s_i, ..., s_{j-1}\}$. Let x' be the linear structure obtained be deleting all states of index greater than j-1 and making s_i the successor of s_{j-1}. It is readily checked that $x' \models p_0$. Moreover, since the order of successor states does not matter we can, in general, delete many states, while preserving the truth of p_0 in the resulting linear structure. We need only retain in the "loop," from state s_i to s_{j-1} and back, a single state labelled q, for each formula Fq that appears in the label some state in the loop. The other states in the loop can be deleted, reducing its size to at most n states. We also need to ensure that each Fq that appears somewhere in the "stem," from s_0 to s_{i-1}, is fulfilled by a q labelling some subsequent state. The other states in the stem can be deleted reducing the size of the stem to at most n states. The final structure, x'', is still a model of p_0, and is of size at most 2n states. □

Theorem 6.19. The problem of testing satisfiability for PLTL(F) is NP-complete.

Proof Idea. Membership in NP follows using the Linear Size Model Theorem. An algorithm can be designed that, given a formula of length n, guesses a candidate model of size O(n) and then checks that it is indeed a model in time $O(n^2)$. NP-hardness follows since the logic subsumes propositional logic. □

Finally, it can be shown that the complexity of testing satisfiability of the very expressive branching time logic CTL* has an upper bound of deterministic double exponential time, by means of a quite elaborate reduction to the nonemptiness problem for finite state automata on infinite trees (see section 6.5). A lower bound of deterministic double exponential time has also been established by a reduction from alternating exponential space Turing machines in [VS85]. (Note: By double exponential we mean exp(exp(n)), where exp(n) is a function c^n, for some $c > 1$.) Thus we have,

Theorem 6.20. The problem of testing satisfiability for CTL* is complete for deterministic double exponential time.

6.3 Deductive Systems

A deductive system for a temporal logic consists of a set of axiom schemes and inference rules. A formula p is said to be *provable*, written $\vdash$p, if there exists a finite sequence of formulae, ending with p such that each formula is an instance of an axiom or follows from previous formulae by application of one of the inference rules. A deductive system is said to be *sound* if every provable formula is valid. It is said to be *complete* if every valid formula is provable.

Consider the following axioms and rules of inference:

Axiom Schemes:

Ax1.	All validities of propositional logic
Ax2	$\mathrm{EFp} \equiv \mathrm{E}[\mathit{true}\ \mathrm{U}\ \mathrm{p}]$
Ax2b.	$\mathrm{AGp} \equiv \neg\mathrm{EF}\neg\mathrm{p}$
Ax3.	$\mathrm{AFp} \equiv \mathrm{A}[\mathit{true}\ \mathrm{U}\ \mathrm{p}]$
Ax3b.	$\mathrm{EGp} \equiv \neg\mathrm{AF}\neg\mathrm{p}$
Ax4.	$\mathrm{EX(p \vee q) \equiv EXp \vee EXq}$
Ax5.	$\mathrm{AXp \equiv \neg EX\ \neg p}$
Ax6.	$\mathrm{E(p\ U\ q) \equiv q \vee (p \wedge EXE(p\ U\ q))}$
Ax7.	$\mathrm{A(p\ U\ q) \equiv q \vee (p \wedge AXA(p\ U\ q))}$
Ax8.	$\mathrm{EX}\mathit{true} \wedge \mathrm{AX}\mathit{true}$
Ax9.	$\mathrm{AG(r \Rightarrow (\neg q \wedge EXr)) \Rightarrow (r \Rightarrow \neg A(p\ U\ q))}$
Ax9b.	$\mathrm{AG(r \Rightarrow (\neg q \wedge EXr)) \Rightarrow (r \Rightarrow \neg AFq))}$
Ax10.	$\mathrm{AG(r \Rightarrow (\neg q \wedge (p \Rightarrow AXr))) \Rightarrow (r \Rightarrow \neg E(p\ U\ q))}$
Ax10b.	$\mathrm{AG(r \Rightarrow (\neg q \wedge AXr)) \Rightarrow (r \Rightarrow \neg EFq))}$
Ax11.	$\mathrm{AG(p \Rightarrow q) \Rightarrow (EXp \Rightarrow EXq)}$

Rules of Inference:

R1. if $\vdash$p then $\vdash$AGp (Generalization)

R2. if $\vdash$p and $\vdash$p $\Rightarrow$ q then $\vdash$q (Modus Ponens)

This deductive system for CTL is easily seen to be sound. We can also establish the following (cf. [EH85], [BPM81]):

Theorem 6.21. The above deductive system for CTL is complete.

Proof Sketch. Suppose p_0 is valid. Then $\sim p_0$ is unsatisfiable. We apply the above tableau-based decision procedure to $\sim p_0$. All nodes whose label includes $\sim p_0$ will be eliminated. In the sequel, we use the following notation and terminology. We use $\wedge$s to denote the conjunction of all formulae labelling node s. We also write p $\in$ s for p $\in$ L(s), and we say that formula p is *consistent* provided that not $\vdash \sim$p.

Claim 1: If node s is deleted then $\vdash \sim(\wedge s)$.

Assuming the claim, we will show that $\vdash p_0$. We will use the formulae below, whose validity can be established by propositional reasoning:

$$\vdash q \equiv \vee\{\wedge s\text{: s is a node in the tableau and } q \in s\} \text{ for each formula } q \in ecl(p_0)$$
$$\equiv \vee\{\wedge s\text{: s is a node in the tableau and } q \in s \text{ and } \wedge s \text{ is consistent}\}$$

$$\vdash \mathit{true} \equiv \vee\{\wedge s\text{: s is a node in the tableau}\} \equiv \vee\{\wedge s\text{: s is a node in the tableau and } \wedge s \text{ is consistent}\}$$

Thus $\vdash\sim p_0 \equiv \vee\{\wedge s$: s is a node in the tableau and not$(p_0) \in s\}$. Because $\sim p_0$ is unsatisfiable the decision procedure will delete each node s containing p_0 in its label. By Claim 1 above, for each such node s that is eliminated, $\vdash$ not($\wedge$s). Thus we get $\models\sim\sim p_0$ and also $\vdash p_0$.

Before proving Claim 1, we establish

Claim 2: If (s,t) $\notin$ R as originally constructed then $\wedge$s $\wedge$ EX$\wedge$t is inconsistent.

Proof: Suppose (s,t) $\notin$ R. Then for some formulae AXp, AXp $\in$ s and $\sim$p $\in$ t. Thus, we can prove the following

a.	$\vdash \wedge s \Rightarrow AXp$	(since AXp $\in$ s)
b.	$\vdash \wedge t \Rightarrow \sim p$	(since $\sim$p $\in$ t)
c.	$\vdash AG(\wedge t \Rightarrow \sim p)$	(generalization rule)
d.	$\vdash EX\wedge t \Rightarrow EX\neg p$	(Ax11: monotonicity of EX operator)
e.	$\vdash (\wedge s \wedge EX\wedge t) \Rightarrow AXp \wedge EX\sim p$	(lines a,d and propositional reasoning)
f.	$\vdash (\wedge s \wedge EX\wedge t) \Rightarrow$ *false*	(Ax5 and def. AX operator)
g.	$\vdash \sim(\wedge s \wedge EX\wedge t)$	(propositional reasoning)

Thus we have established that $\wedge$s $\wedge$ EX$\wedge$t is inconsistent, thereby completing the proof of claim 2.

We now are ready to give the proof of Claim 1. We argue by induction on when a node is deleted that, if node s is deleted then $\vdash \sim\wedge s$.

Case 1: If $\wedge$s is consistent, then s is not deleted on account of having no successors.

To see this, we note that we can prove

$$
\begin{aligned}
\vdash \wedge s \quad & \equiv \wedge s \wedge EX\mathit{true} \\
& \equiv \wedge s \wedge EX(\vee\{\wedge t\text{: } \wedge t \text{ is consistent}\}) \\
& \equiv \wedge s\ (\vee\{EX\wedge t\text{: } \wedge t \text{ is consistent}\}) \\
& \equiv \vee\{\wedge s \wedge EX\wedge t\text{: } \wedge t \text{ is consistent}\}
\end{aligned}
$$

Thus if $\wedge$s is consistent, $\wedge$s $\wedge$ EX$\wedge$t is consistent for some t. By Claim 1 above (s,t) $\in$ R in the original tableau. By induction, hypothesis, node t is not eliminated. Thus, (s,t) $\in$ R in the current tableau and node s is not eliminated due to having no successors.

Case 2: Node s is eliminated on account of EXq $\in$ s, but s has no successor t with q $\in$ t.

This is established using an argument like that in case 1.

Case 3: Node s is deleted on account of EFq $\in$ s, which is not fulfilled (ranked) at s.

Let V = {t : EFq $\in$ t but is not fulfilled} $\cup$ {t : EFq $\notin$ t}. Note that node s $\in$ V. Moreover, the complement of V = {t: EFq $\in$ t and is fulfilled}.

Let r = $\vee$\{$\wedge$ t: t $\in$ V\}. We claim that $\vdash$ r $\Rightarrow$ ($\neg$q $\wedge$ AXr). It is clear that $\vdash$ r $\Rightarrow$ $\neg$q, because $\neg$q $\in$ t for each t $\in$ V and $\vdash$ $\wedge$t $\Rightarrow$ $\neg$q. We must now show that $\vdash$ r $\Rightarrow$ AXr. It suffices to show that for each t $\in$ V, $\vdash$ $\wedge$t $\Rightarrow$ AXr. Suppose not. Then $\exists$ t $\in$ V, $\wedge$t $\wedge$ EX$\sim$r is consistent. Since $\neg$r = $\vee$\{$\wedge$t$'$: t$'$ $\notin$ V\}, $\exists$ t $\in$ V $\exists$ t$'$ $\notin$ V, $\wedge$t $\wedge$ EX$\wedge$t$'$ is consistent. By claim 2 above, (t,t$'$) $\in$ R as originally constructed, and since $\wedge$t and $\wedge$t$'$ are each consistent neither is eliminated, by induction hypothesis. So (t,t$'$) $\in$ R in the current tableau. Since t$'$ $\notin$ V, EFq $\in$ $'$ and is ranked. But by virtue of the arc (t,t$'$) in the tableau, t should also be ranked for EFq, a contradiction to t being a member of V. Thus $\vdash$ r $\Rightarrow$ AXr.

By generalization $\vdash$ AG(r $\Rightarrow$ AXr) and by the induction axiom for EF and modus ponens, $\vdash$ r $\Rightarrow$ $\neg$EFq. Now $\vdash \wedge$s $\Rightarrow$ r, by definition of r (as the disjunction of formulae for each state in V, which includes node s). However, we had assumed EFq $\in$ s which of course means that $\vdash \wedge$s $\Rightarrow$ EFq. Thus $\vdash \wedge$s $\Rightarrow$ *false*, so that $\wedge$ s is inconsistent.

The proofs for the other cases for eventualities E(p U q), AFq, and A(p U q) are similar to that for case 3. □

6.4 Model Checking

The model checking problem (roughly) is: Given a finite structure M and a propositional TL formula p, does M define a model of p? For most any propositional TL the model checking problem is decidable since we can do, if needed, an exhaustive search through the paths of the finite input structure. The problem has important applications to mechanical verification of finite state concurrent systems (see section 7.3). The significant issues from the theoretical standpoint are to analyze and classify logics with respect to the complexity of model checking. For some logics, which have adequate expressive power to capture certain important correctness properties, we can develop very efficient algorithms for model checking. Other logics cannot be model checked so efficiently.

We say roughly because there is some potential ambiguity in the above definition. What system of TL is the formula p from? In particular, it is branching or linear time? Also, what does it mean for a structure M to be a model of a formula p? From the definition of satisfiability for a formula p_0 of branching time logic, a state formula, it seems that we should say a structure M is a model of a formula p_0 provided it contains a state s such that M,s $\models p_0$. From the technical definition of satisfiability for a formula p_0 of linear time logic, it appears we should say a structure M is a model of a formula p_0 provided it contains a fullpath x such that M,x $\models p_0$. However, the number of fullpaths can be exponential in the size of a finite structure M. It thus seems that the complexity of model checking for linear time could be very high, since in effect an examination of all paths through the structure could be required.

To overcome these difficulties, we therefore formalize the model checking problem as follows:

The *Branching Time Logic Model Checking Problem (BMCP)* formulated for propositional branching time logic BTL is: Given a finite structure M=(S,R,L) and a BTL formula p, determine for each state s in S whether M,s $\models$ p and, if so, label s with p. The *Linear Time Logic Model Checking Problem (LMCP)* for propositional linear time logic LTL can be similarly formulated as follows: Given a finite structure M=(S, R, L) and an LTL formula p, determine for each state in S, whether there is a fullpath satisfying p starting at s, and, if so, label s with Ep.

This definition of LMCP may, at first glance, appear to be incorrectly formulated because it defines truth of linear time formulae in terms of states. However, one should note that there is a fullpath in finite structure M satisfying linear time formula p_0 iff there is such a fullpath starting at some state s of M. It thus suffices to solve LMCP and then scan the states to see if one is labelled with Ep. We can also handle the applications-oriented convention that linear time formula p is true of a structure (representing a concurrent program) iff it is true of all (initial) paths in the structure, because p is true of all paths in the structure iff Ap holds at all states of the structure.

Since $Ap \equiv \neg E \neg p$, by solving LMCP and then scanning all (initial) states to check whether Ap holds, we get a solution to the applications formulation.

We now analyze the complexity of model checking linear time. The next three results are from [SC85]:

Lemma 6.22. The model checking problem for PLTL is polynomial time reducible (transformable) to the satisfiability problem for PLTL.

Proof Sketch. The key idea is that we can readily encode the organization of a given finite structure into a PLTL formula. Suppose M = (S,R,L) is a finite structure and p_0 a PLTL formula, over underlying set of atomic propositions AP. Let AP′ be an extension of AP obtained by including a new, "fresh" atomic proposition Q_s for each state $s \in S$. The local organization of M at each state s is captured by the formula

$$q_s = Q_s \Rightarrow (\wedge_{P \in L(s)} P \wedge \wedge_{P \notin L(s)} \wedge \vee_{(s,t) \in R} XQ_t$$

while the formula below asserts that the above local organization prevails globally:

$$q' = G((\Sigma_{s \in S} Q_s{=}1) \wedge \wedge_{s \in S} q_1)$$

and means, in more detail, that exactly one Q_s is true at each time and that the corresponding q_s holds.

Claim: There exists a fullpath x_1 in M such that $M,x_1 \models p_0$ iff $q' \wedge p_0$ is satisfiable.

The $\rightarrow$ direction is clear: annotate M with propositions from AP′. The path x_1 so annotated is model of $q' \wedge p_0$.

The $\leftarrow$ direction can be seen as follows. Suppose $M',x \models q' \wedge p_0$. The $x = u_0,u_1,u_2,\ldots$ matches the organization of M in that, for each i (a) with state u_i we associate a state s of M—the unique one such that $M',u_i \models P_s$—that satisfies the same atomic propositions in AP as does s; call it $s(u_i)$ and (b) the successor u_{i+1} along x of u_i is associated with a state $t = s(u_{i+1})$ of M which is a successor of s in M. Thus, the path $x_1 = s(u_0),s(u_1),s(u_2),\ldots$ in M is such that $M,x_1 \models p_0$. □

Theorem 6.23. The model checking problem for PLTL is PSPACE-complete.

Proof Idea. Membership in PSPACE follows from the preceding lemma and the theorem establishing that satisfiability is in PSPACE. PSPACE-hardness follows by a generic reduction from PSPACE Turing machines. □

Remark: The above PSPACE-completeness result holds for PLTL(F,X), the sublanguage of PLTL obtained by restricting the temporal operators to just X, F, and its dual G. It also holds for PLTL(U), the sublanguage of PLTL obtained by restricting the temporal operators to just U and its dual B.

Theorem 6.24. The problem of model checking for PLTL(F) is NP-complete.

Proof Idea. To establish membership in NP, we design a nondeterministic algorithm that guesses a finite path in the input structure M leading to a strongly connected component, such that any unwinding of the component prefixed by some finite path comprises a candidate model of the input formula p_0. To check that it is indeed a model evaluate each subformula of each state of the candidate model, which can be done in polynomial time. NP-hardness follows by a reduction from 3-SAT. □

We now turn to model checking for branching time logic. First we have from [CE81]:

Theorem 6.25. The model checking problem for CTL is in deterministic polynomial time.

This result is somewhat surprising since CTL seems somehow more complicated than the linear time logic PLTL. Because of such seemingly unexpected complexity results, the question of the complexity of model checking has been an issue in the branching versus linear time debate. Branching time, as represented by CTL, appears to be more efficient than linear time, but at the cost of potentially valuable expressive power, associated with, for example, fairness.

However, the real issue for model checking is not branching versus linear time, but simply what are the basic modalities of the branching time logic to be used. Recall that the basic modalities of a branching time logic are those of the form Ap or Ep, where p is a "pure" linear time formula containing no path quantifiers itself. Then we have the following result of [EL85]:

Theorem 6.26. Given any model checking algorithm for a linear logic LTL there is a model checking algorithm for the corresponding branching logic BTL, whose basic modalities are defined by the LTL, of the same order of complexity.

Proof idea. Simply evaluate nested branching time formulae Ep or Ap by recursive descent. For example, to model check EFAGP, recursively model check AGP, then label every state labelled with AGP with fresh proposition Q and model check EFQ. □

For example, CTL* can be reduced to PLTL since the basic modalities of CTL* are of the form A or E followed by a PLTL formula. As a consequence we get (cf. [CES83]):

Corollary 6.27. The model checking problem for CTL* is PSPACE-complete.

Thus the increased expressive power of the basic modalities of CTL* incurs a significant complexity penalty. However, it can be shown that basic modalities for reasoning under fairness assumptions do not cause complexity difficulties for model checking. These matters are discussed further in Section 7.

6.5 Automata on Infinite Objects

There has been a resurgence of interest in finite state automata on infinite objects, due to their close connection to TL. They provide an important alternative approach to developing decision procedures for testing satisfiability for propositional temporal logics. For linear time temporal logics the tableau for formula p_0 can be viewed as defining a finite automaton on infinite strings that essentially accepts a string iff it defines a model of the formula p_0. The satisfiability problem for linear logics is thus reduced to the emptiness problem of finite automata on infinite strings. In a related but somewhat more involved fashion, the satisfiability problem for branching time logics can be reduced to the nonemptiness problem for finite automata on infinite trees.

For some logics, the only known decision procedures of elementary time complexity (i.e., of time complexity bounded by the composition of a fixed number of exponential functions), are obtained by reductions to finite automata on infinite trees. The use of automata transfers some difficult combinatorics onto the automata-theoretic machinery. Investigations into such automata-theoretic decision procedures is an active area of research interest.

We first outline the automata-theoretic approach for linear time. As suggested by Theorem 6.16, the tableau construction for CTL can be specialized, essentially by dropping the path quantifiers to define a tableau construction for PLTL. The extended closure of a PLTL formula p_0, ecl(p_0), is defined as for CTL, remembering that in a linear structure, $Ep \equiv Ap \equiv p$. The notions of maximal, and propositionally consistent subsets of of ecl(p_0) are also defined analogously. The (initial) tableau for p_0 is then a structure T = (S,R,L) where S is the set of maximal, propositionally consistent subsets of ecl(p_0), i.e., states, $R \subseteq S \cdot S$ consists of the transitions (s,t) defined by the rule $(s,t) \in R$ exactly when $\forall$ formula $Xp \in ecl(p_0)$, $Xp \in s$ iff $p \in t$, and L(s) = s, for each $s \in S$.

We may view the tableau for PLTL formula p_0 as defining the transition diagram of a nondeterministic finite state automaton $\mathcal{A}$ which accepts the set of infinite strings over alphabet Σ = PowerSet(AP) that are models of p_0, by letting the arc (u,v) be labelled with AtomicPropositions(v), i.e., the set of atomic propositions in v. Technically, $\mathcal{A}$ is a tuple of the form $(S \cup \{s_0\}, \Sigma, \delta, s_0, \text{—}\ )$ where $s_0 \notin S$ is a unique start state, δ is defined so that $\delta(s_0,a) = \{$ states $s \in S$: $p_0 \in s$ and AtomicPropositions(s) = a$\}$ for each $a \in \Sigma$, $\delta(s,a) = \{$ states $t \in S$: $(s,t) \in R$ and AtomicPropositions(s) = a$\}$. The acceptance condition is defined below. A *run* r of $\mathcal{A}$ on input $x = a_1a_2a_3... \in \Sigma^\omega$ is an infinite sequence of states $s_0s_1s_2...$ such that $\forall i \geq 0$ $\delta(s_i,a_{i+1}) \supseteq \{s_{i+1}\}$. Note that $\forall i \geq 1$ AtomicPropositions(s_i) = a_i. Any run of $\mathcal{A}$ would correspond to a model of p_0, in that $\forall i \geq 1$, $x^i \models \wedge\{$ formulae p: $p \in s_i\}$, except that eventualities might not be fulfilled. To check fulfillment, we can easily define acceptance in terms of complemented pairs (cf. [Th89]). If ecl(p_0) has m eventualities $(p_1 \mathrm{U} q_1),...,(p_m \mathrm{U} q_m)$, we let $\mathcal{A}$ have m pairs (RED_i, $GREEN_i$) of lights. Each time a state containing $(p_i \mathrm{U} q_i)$ is entered, flash RED_i; each time a state containing q_i is entered flash $GREEN_i$. A run r is accepted iff for each $i \in [1:m]$, there are infinitely many RED_i flashes implies there are infinitely many $GREEN_i$ flashes iff every eventuality is fulfilled iff the input string x is a model of p_0.

We can convert $\mathcal{A}$ into an equivalent nondeterministic Buchi automaton $\mathcal{A}_1$, where acceptance is defined simply in terms of a single GREEN light flashing infinitely often. We need some terminology. We say that the eventuality (p U q) is *pending* at state s of run r provided that $(p \mathrm{U} q) \in s$ and $q \notin s$. Observe that run r of $\mathcal{A}$ on input x corresponds to a model of p_0 iff not($\exists$ eventuality $(p \mathrm{U} q) \in ecl(p_0)$, (p U q) is pending almost everywhere along r) iff $\forall$ eventuality $(p \mathrm{U} q) \in ecl(p_0)$, (p U q) is not pending infinitely often along r. The Buchi automaton $\mathcal{A}_1$ is then obtained from $\mathcal{A}$ augmenting the state with an m+1 valued counter. The counter is incremented from i to i+1 mod (m + 1) when the i^{th} eventuality, $(p_i \mathrm{U} q_i)$ is next seen to be not pending along the run r. When the counter is reset to 0, flash GREEN and set the counter to 1. (If m = 0, flash GREEN is every state.) Now observe that there are infinitely many GREEN flashes iff $\forall i \in [1:m]$ $(p_i \mathrm{U} q_i)$ is not pending infinitely often iff every pending eventuality is eventuality fulfilled iff the input string x defines a model of p_0. Moreover, $\mathcal{A}_1$ still has $\exp(|p_0|) \cdot O(|p_0|) = \exp(|p_0|)$ states.

Similarly, the tableau construction for a branching time logic with relatively simple modalities such as CTL can be viewed as defining a Buchi tree automaton that, in essence, accepts all models of a candidate formula p_0. (More precisely, every tree accepted by the automaton is a model of p_0, and if p_0 is satisfiable there is some tree accepted by the automaton.) General automata-theoretic techniques for reasoning about a number of relatively simple logics, including CTL, using Buchi tree automata have been described by Vardi and Wolper [VW84].

For branching time logics with richer modalities such as CTL*, the tableau construction is not directly applicable. Instead, the problem reduces to constructing a tree automaton for the branching time modalities (such as Ap) in terms of the string automaton for the corresponding linear time formula (such as p). This tree automaton will in general involve a more complicated acceptance condition such as pairs or complemented pairs, rather than the simple Buchi condition. Somewhat surprisingly, the only known way to build the tree automaton involves difficult combinatorial arguments and/or appeals to powerful automata-theoretic results such as McNaughton's construction ([McN66]) for determinizing automata on infinite strings.

The principal difficulty manifests itself with just the simple modality Ap. The naive approach of building the string automaton for p and then running it down all paths to get a tree automaton for Ap will not work. The string automaton for p must be determinized first. To see this, consider two paths xy and xz in the tree which start off with the same common prefix x but eventually separate to follow two different infinite suffixes y or z. It is possible that p holds along both paths, but in order for the nondeterministic automaton to accept, it might have to "guess" while reading a particular symbol of x whether it will eventually read the suffix y or the suffix z. The state it guesses for y is in general different from the state it guesses for z. Consequently, no single run of a tree automaton based on a nondeterministic string automaton can lead to acceptance along all paths.

For a CTL* formula of length n, use of classical automata-theoretic results yields an automaton of size triple exponential in n. (Note: by triple exponential we mean exp(exp(exp(n))), etc.) The large size reflects the exponential cost to build the string automaton as described above for a linear time formula p plus the double exponential cost of McNaughton's construction to determinize it. Nonemptiness of the automaton can be tested in exponential time to give a decision procedure of deterministic time complexity quadruple exponential. in n. In [ESi84] it was shown that, due to the special structure of the string automata derived from linear temporal logic formulae, such string automata could be determinzed with only single exponential blowup. This reduced the complexity of the CTL* decision procedure to triple exponential. Further improvement is possible as described below.

The size of a tree automaton is measured in terms of two parameters: the number of states and the number of pairs in the acceptance condition. A careful analysis of the tree automaton constructions in temporal decision procedures shows that the number of pairs is logarithmic in the number of states, and for CTL* we get an automaton with double exponential states and single exponetial pairs. An algorithm of [EJ88] shows how to test nonemptiness in time polynomial in the number of states, while exponential in the number of pairs. For CTL* this yields a decision procedure of deterministic double exponential time complexity, matching the lower bound of [VS85].

One drawback to the use of automata is that, due to the delicate combinatorial constructions involved, there is usually no clear relationship between the structure of the automaton and the syntax of the candidate formula. An additional drawback is that in such cases, the automata-theoretic approach provides no aid in finding sound and complete axiomitizations. For example, the existence of an explicit, sound and complete axiomitization for CTL* has been an open question for some time. (Note: We refer here to an axiomitization for its validities over the usual semantics generated by a binary relation; interestingly, for certain nonstandard semantics, complete axiomitizations are known (cf. [Ab80], [LS84]).)

However, there are certain definite advantages to the automata-theoretic approach. First, it does provide the only known elementary time decision procedures for some logics. Secondly, automata can provide a general, uniform framework encompassing temporal reasoning (cf. [VW5] [VW86] [V87]). Automata themselves have been proposed as a potentially useful specification language. Automata, moreover, bear an obvious relation to temporal structures, abstract concurrent programs, etc. This makes it possible to account for various types of temporal reasoning applications such as program synthesis and mechanical verification of finite state programs in a conceptually uniform fashion. Verification systems based on automata have also been developed (cf. [Ku86]).

We note that not only has the field of TL benefited from automata theory, but the converse holds as well. For example, the tableau concept for the branching time logic CTL, particularly the state/prestate formulation, suggests a very helpful notion of the transition diagram for a tree automaton (cf. [Em85]). This has made it possible to apply tableau-theoretic techniques to automata, resulting in more efficient algorithms for testing nonemptiness of automata, which in turn can be used to get more efficient decision procedures for satisfiability of TL's (cf. [EJ88]). Still another improved nonemptiness algorithm, motivated by program synthesis applications is given in [PR89]. New types of automata on infinite objects have also been proposed to facilitate reasoning in TL's (cf. [St81], [VS85], [MP87a]). A particularly important advance in automata theory motivated by TL is Safra's construction ([Sa88]) for determinizing an automaton on infinite strings with only a single exponential blowup, without regard to any special structure possessed by the automaton. Not only is Safra's construction an exponential improvement over McNaughton's construction but it is conceptually much more simple and elegant. In this way we see that not only can TL sometimes benefit from adopting the automata-theoretic viewpoint, but also conversely and even synergistically, the study of automata on infinite objects has been advanced by work motivated by and using the techniques of TL.

7 The Application of Temporal Logic to Program Reasoning

Temporal Logic has been suggested as a formalism especially appropriate to reasoning about ongoing concurrent programs, such as operating systems, which have a *reactive* nature, as explained below (cf. [Pn86]).

We can identify two different classes of programs (also referred to as systems). One class consists of those ordinarily described as "sequential" programs. Examples include a program to sort a list, programs to implement a graph algorithm as discussed in, say, the chapter on graph algorithms, and programs to perform a scientific calculation. What these programs have in common is that they normally terminate. Moreover, their behavior has the following pattern: they initially accept some input, perform some computation, and then terminate yielding final output. For all such systems, correctness can be expressed in terms of a Precondition/Postcondition pair in a formalism such as Hoare's logic or Dijkstra's weakest preconditions, because the systems' underlying semantics can be viewed as a *transformation* from initial states to final states, or from Postconditions to Preconditions.

The other class of programs consists of those which are continuously operating, or, ideally, nonterminating. Examples include operating systems, network communication protocols, and air traffic control systems. For a continuously operating program its normal behavior is an arbitrarily long, possibly nonterminating computation, which maintains an ongoing interaction with the environment. Such programs can be described as *reactive* systems. The key point concerning such systems is that they maintain an ongoing interaction with the environment, where intermediate outputs of the program can influence subsequent intermediate inputs to the program. Reactive systems thus subsume many programs labelled as concurrent, parallel, or distributed, as well as process control programs. Since there is in general no final state, formalisms such as Hoare's logic which are based on an initial state-final state semantics, are of little use for such reactive programs. The operators of temporal logic such as *sometimes* and *always* appear quite appropriate for for describing the time-varying behavior of such programs.

What is the relationship between concurrency and reactivity? They are in some sense independent. There are transformational programs that are implemented to exploit parallel architectures (usually, to speed processing up, allowing the output to be obtained more quickly). A reactive system could also be implemented on a sequential architecture.

On the other hand, it can be recommended that in general concurrent programs should be viewed as reactive systems. In a concurrent program consisting of two or more processes running in parallel, each process is generally maintaining an ongoing interaction with its environment, which usually includes one or more of the other processes. If we take the compositional viewpoint, where the meaning of the whole is defined in terms of the meaning of its parts, then the entire system should be viewed in the same fashion as its components, and the view of any system is a reactive one. Even if we are not working in a compositional framework, the reactive view of the system as a whole seems a most natural one in light of the ongoing behavior of its components. Thus, in the sequel when we refer to a concurrent program, we mean a reactive, concurrent system.

There are two main schools of thought regarding the application of TL to reasoning about concurrent programs. The first might be characterized as "proof-theoretic." The basic idea is to manually compose a program and a proof of its correctness using a formal deductive system, consisting of axioms and inference rules, for an appropriate temporal specification language. The second might be characterized as "model-theoretic." The idea here is to use decision procedures that manipulate the underlying temporal models corresponding to programs and specifications to automate the tasks of program construction and verification. We subsequently outline the approach of each of these two schools. First, however, we discuss the types of correctness properties of practical interest for concurrent programs and their specification in TL.

7.1 Correctness Properties of Concurrent Programs

There are a large number of correctness properties that we might wish to specify for a concurrent program. These correctness properties usually fall into two broad classes (cf. [Pn77], [OL82]). One class is that of "safety" properties also known as "invariance" properties. Intuitively, a safety property asserts that "nothing bad happens." The other class consists of the "liveness" properties also referred to as "eventuality" properties or "progress" properties. Roughly speaking, a liveness property asserts that "something good will happen." These intuitive descriptions of safety and liveness are made more precise below, following [Pn86].

A *safety* property states that each finite prefix of a (possibly infinite) computation meets some requirement. Safety properties are thus those that are (initially) equivalent to a formula of the form Gp, for some past formula p. The past formula describes the condition required of finite prefixes, while the G operator ensures that p holds of all finite prefixes. Note that this formal definition of safety requires that always "nothing bad has happened yet," consistent with the intuitive characterization of [OL82] mentioned above.

Any formula built-up from past formulae, the propositional connectives $\wedge$ and $\vee$, and the future temporal operators G and U_w can be shown to express a safety property. For example, $(p\ U_w\ q) \equiv_i G(G^- p \vee F^-(q \wedge X^- G^- p)$.

A number of concrete examples of safety properties can be given. The partial correctness of a program with respect to a precondition ϕ and postcondition ψ, which stipulates that if program execution begins in a state satisfying ϕ, then if it terminates the final state satisfies ψ, is expressed by

$$atl_0 \wedge \phi \Rightarrow G(atl_h \Rightarrow \psi)$$

where the program's start label is l_0 and its halt label is l_h. (Note: this formula is initially equivalent to $G(F^-(\neg(atl_0 \wedge \phi) \wedge X_w^-\ \mathit{false})) \vee G(atl_h \Rightarrow \psi))$ thereby demonstrating that it is safety property according to the technical definition.)

Other safety properties include *global invariance* of assertion p is expressed simply by Gp. To capture *local invariance* which means that p holds whenever control is at location l, we write $G(atl \Rightarrow p)$.

The requirement of *mutual exclusion* for a two process solution to the critical section problem can be written

$$G(\neg(atCS_1 \wedge atCS_2))$$

where $atCS_i$ indicates that control of process i is at its critical section.

Another very important property for concurrent programs is *freedom from deadlock*. A concurrent program is *deadlocked* if no process is enabled to proceed. The formula $G(\ \mathit{enabled}_1 \vee ... \vee \mathit{enabled}_m\ )$ captures freedom from deadlock for a concurrent program with m processes.

Liveness properties are in some sense dual to safety properties, requiring that some finite prefix property hold a certain number of times.

The *basic liveness* properties are technically defined to be those (initially) expressible in the form Fp, $\overset{\infty}{F}p$, or $\overset{\infty}{G}p$, where p is a past formula required to hold for some, for infinitely many, or for all but but a finite number, resp., of the finite prefixes of a computation. It is interesting to note that $(p\ U_s\ q) \equiv_i F(q \wedge X_w^-\ G^- p)$ for any past formulae p and q, thus showing the strong until to be a basic liveness property, even though it is not immediately obvious that it can be expressed in the required form. Also note that $Fp \equiv_i GF(F^- p) \equiv_i FG(F^- p)$ and is technically redundant, even though we find it more convenient to keep Fp separated out. A more serious redundancy is that, by our definition, each safety property is a basic liveness property, since $Gp \equiv_i GF(G^- p)$ for any past formula p.

If we wish to avoid this redundancy, we can first define an *invincible* past formulae to be one such that every finite sequence x has a finite extension x' with $(x',\text{length}(x')) \models p$ (i.e., with p holding at the last state of x').

We then define the *pure liveness* properties to be those initially equivalent to one of the formulae Fp, GFp, FGp, for some invincible past formula p. Note that any satisfiable state formula p is an invincible past formula, so that the pure liveness formulae still include a broad range of properties. However, $(p\ U_s\ q)$ is not a pure liveness property, because while $(p\ U_s\ q) \equiv_i F(q \wedge X^-F^-p)$, the formula $q \wedge X^-F^-p$ is not invincible. It is expressible as the conjunction of a safety property and a pure liveness property: $(p\ U_s\ q) \equiv_i (p\ U_w\ q) \wedge Fq$.

Note that if p is a pure liveness property, then it has the following characteristic: every finite sequence x can be extended to a finite or infinite sequence x' such that $(x',0) \models p$. This corresponds to the intuitive characterization of liveness, that "something good will happen," of [OL82].

Further work on syntactic and semantic characterizations of safety and liveness properties are given in [AS85] and [Si85.]

One important generic liveness property has the form

$$G(p \Rightarrow Fq)$$

for past formulae p and q, and is called *temporal implication* (cf. [Pn77], [La80]). Many specific correctness properties are instances of temporal implication, as described below.

An *intermittent assertion* is expressed by

$$G(\ (atl \wedge \phi) \Rightarrow F(atl' \wedge \phi'))$$

meaning that whenever ϕ is true at location l, then ϕ' will eventually be true at location l' (cf. [Bu74], [MW78]). An important special type of intermittent assertion is *total correctness* of a program with respect to a precondition ϕ and postcondition ψ. It is expressed by

$$atl_0 \wedge \phi \Rightarrow F(atl_h \wedge \psi)$$

which indicates that if the program starts in a state satisfying ϕ, then it halts in a state satisfying ψ.

The property of *guaranteed accessibility* for a process in a solution to the mutual exclusion problem to enter its critical section, once it has indicated that it wishes to do so is expressed by

$$G(atTry_i \Rightarrow FatCS_i)$$

where $atTry_i$ and $atCS_i$ indicated that process i is in its Trying section or Critical section, respectively. This property is sometimes referred to as *absence of individual starvation for process i.* General guaranteed accessibility is of the form

$$G(atl \Rightarrow Fatl')$$

Still another property expressible in this way is *responsiveness.* Consider a system consisting of a resource controller that monitors access to a shared resource by competing user processes. We would like to ensure that each request for access eventually leads to a response in the form of a granting of access. This is captured by an assertion of the form $G(req_i \Rightarrow Fgrant_i)$ where req_i and $grant_i$ are predicates indicating that a request by process is made or a grant of access to process i is given, respectively.

The fairness properties discussed in Section 5 are also liveness properties.

A final general type of correctness property is informally known as the *precedence* properties. These properties have to do do with temporal ordering, precedence, or priority of events. We shall not give a formal definition but instead illustrate the class by several examples.

To express *absence of unsolicited response* as in the resource controller example above, where we want a $grant_i$ to be issued only if preceded by a req_i we can write

$$\neg grant_i \Rightarrow (\neg grant_i \ U_w \ req_i).$$

Alternatively, we can write $(req_i \ B \ grant\ _i)$, where we recall that the precedes operator $(p \ B \ q)$ asserts that the first occurrence of q, if any, is strictly preceded by an occurrence of p.

The important property of First-In-First-Out(FIFO) responsiveness can be written in a straightforward but slightly imprecise fashion as

$$(req_i \ B \ req_j) \Rightarrow (grant_i \ B \ grant_j).$$

A more accurate expression is

$$(req_i \land \neg req_j \land \neg grant_j) \Rightarrow ((\neg grant_j) \ U_w \ grant_i)$$

where we rely on the assumption that once a request has been made, it is not withdrawn before it has been granted. Hence, $req_i \land \neg req_j$ implies that process i's request preceded that of process j.

It is interesting to note the importance of correctly formalizing in the formal specification language our intuitive understanding of the problem. An important application where this issue arises is the specification of correct behavior for a message buffer. Such buffers are often used in distributed systems based on message passing, where one process transmits messages to another process via an intermediate, asynchronous buffer that temporarily stores messages in transit.

We assume that the buffer has an input channel x and output channel y. It also has unbounded storage capacity and is assumed to operate according to FIFO discipline. We want to specify that the log of input/output transactions for the buffer is correct, viz., that the sequence of messages output on channel y equals to the sequence of messages input on channel x.

An important limitation of PLTL and related formalisms was established by Sistla et. al. [SCFM84] which shows that an unbounded FIFO buffer cannot be specified in PLTL. Essentially, the problem is that any particular formula p of PLTL is of a fixed size and corresponds to a bounded size finite state automaton, while the buffer can hold an arbitrarily large sequence of messages, thereby permitting the finite automaton to become "confused." Moreover, the problem is not alleviated by extending the formalism to be pure (i.e., uninterpreted) FOLTL (cf. [Ko87]).

However, as noted in [SCFM84] there exist partially interpreted FOLTL's which make it possible to capture correct behavior for a message buffer. One such logic provides history variables that accumulate the string of all previous states along with a prefix predicate ($\leq$) on these histories. The safety portion of the specification is given by $G(y \leq x)$ which asserts that the sequence of messages output is always a prefix of the sequence of messages input. The liveness requirement is expressed by $\forall z\ G(x{=}z \Rightarrow F(y{=}z))$ which ensures that whatever sequence appears along the input channel is eventually replicated along the output channel.

The essential feature of the above specification based on histories is the ability to, in effect, associate a unique sequence number with each message, thereby ensuring that all messages are distinct. Using in(m) to indicate that message m is placed on input channel x and out(m) for the placement of message on output channel y, we have the following alternative specification in the style of [Ko87]): The formula

$$\forall m\ G(in(m)\ B\ out(m))$$

specifies that any message output must have been previously input.

The formula

$$\forall m \forall m'\ G(in(m) \wedge XFin(m') \Rightarrow F(\ out(m) \wedge XFout(m')))$$

asserts that FIFO disciple is maintained, i.e. messages are output in the same order they were input.

The liveness requirement is expressed by

$$\forall m\ G(in(m) \Rightarrow Fout(m))$$

while the assumption of message uniqueness is captured by

$$\forall m\ \forall m'\ G(\ (in(m) \wedge XFin(m')) \Rightarrow (m \neq m')\ )$$

Note that the requirement of message uniqueness is essential for the correctness of the specification. Without it, a computation with, e.g., the same message output twice for each input message would be permitted.

Recently, Wolper [Wo86] has provided additional insight into the power of logical formalisms for specifying message buffers. First, he pointed out that PLTL is *a priori* inadequate for specifying message buffers when the underlying data domain is infinite, since each PLTL formula is finite. However, he goes on to show that PLTL is nonetheless adequate for specifying message buffer protocols that the *data independence* criterion, which requires that the behavior of the protocol does not depend on the value or content of a message. While it is in general undecidable whether a protocol is data independent, a simple syntactic check of the protocol, if positive, ensures data independence. This amounts to checking that the only possible operation performed on message contents are reading from channels to variables, writing from variables to channels, and copying between variables.

It is shown in [Wo86] that it is enough for data independent buffer protocols to assert correctness over a 3 symbol message alphabet $\Sigma = \{m_1, m_2, m_3\}$, so that the input is of the form $m_3{}^*m_1m_3{}^*m_2m_3{}^\omega$ iff the output is of the form $m_3{}^*m_1m_3{}^*m_2m_3{}^\omega$. This matching of output to input can be expressed in PLTL, using propositions in_m_i and out_m_i (assumed to be exclusive and exhaustive), $1 \le i \le 3$, to indicate the appearance of message m_i on the input channel and on the output channel, respectively, as

$$
\begin{aligned}
&(\ (in\_m_3 \ U\ (in\_m_1 \wedge X(in\_m_3 \ U\ (in\_m_2 \wedge XGin\_m_3)))) \Rightarrow \\
&\quad (out\_m_3 \ U\ (out\_m_1 \wedge X(out\_m_3 \ U\ (out\_m_2 \wedge XGout\_m_3))))\ ) \\
&\wedge \wedge_{i=1\ldots3}\ (out\_m_i \ B\ in\_m_i).
\end{aligned}
$$

Intuitively this works because it ensures that each pair of distinct input messages are transmitted through to the output correctly; since the buffer is assumed to be oblivious to the message contents, the only way it can ensure such correct transmission for the three symbol alphabet is to transmit correctly over any alphabet, including those with distinct messages.

The reader may have noticed that the above example specifications were given in linear TL. If we wished to express them in branching TL we would merely need to prefix each assertion by the universal path quantifier. The reason linear TL sufficed was that above we were mainly interested in properties holding of ıall computations of a concurrent program. If we want to express lower bounds on nondeterminism and/or concurrency we need the ability to use existential path quantification, provided only by branching time logic. Such lower bounds are helpful in applications such as program synthesis. Moreover, branching time makes it possible to distinguish between *inevitability* of predicate P, which is captured by AFP, and *potentiality* of predicate P, which is captured by EFP. It also ensures that our specification logic is closed under semantic negation so that we can express, for example, not only absence of deadlock along all futures but also the possibility of deadlock along some future (cf. [La80], [EH86], [Pn85]).

7.2 Verification of Concurrent Programs: Proof-Theoretic Approach

A great deal of work has been done investigating the proof-theoretic approach to verification of concurrent programs using TL (cf. e.g. [Pn81], [MP81], [MP82], [MP83], [La 80], [Ha81], [OL82], [La83], [SMS82]). Typically, one tries to prove, by hand, that a given program meets a certain TL specification using various axioms and inference rules for the system of TL. A drawback of this approach is that proof construction is often a difficult and tedious task, with many details that require considerable effort and ingenuity to organize in an intellectually manageable fashion. The advantage is that human intuition can provide useful guidance that would be unavailable in a (purely) mechanical verification system. It should also be noted that the emphasis of this work has been to develop axioms, rules, and techniques that are useful in practice, as demonstrated on example programs, as opposed to meta-theoretic justifications of proof systems.

A proof system in the LTL framework has been given by Manna and Pnueli [MP83] consisting of three parts (i) A *general* part for reasoning about temporal formulae valid over all interpretations. This includes PLTL and FOLTL; (ii) A *domain* part for reasoning about variables and data structures over specific domains, such as the natural numbers, trees, lists, etc.; and (iii) A *program* part specialized to program reasoning. This system is referred to as a *global* system, since is it intended for reasoning about a program *as a whole*. In this survey, we focus on some useful proof rules from the program part, applicable to broad classes of properties. The reader is referred to [MP82] and [Pn86] for more detail.

The rules are presented in the form

A_1
.
.
.
A_n
——
B

where A_1,...,A_n are premises and B is the conclusion. The meaning is that if all the premises are shown to hold for a program then the conclusion is also true of the program.

The following *invariance* rule (INVAR) is adequate for proving most safety properties. Let ϕ be an assertion:

$$\frac{\begin{array}{l}\phi \\ \mathrm{G}(\phi \Rightarrow \mathrm{X}\phi)\end{array}}{\mathrm{G}\phi}$$

Note that this rule really has the form of an induction rule. The first premise, the basis, ensures that ϕ holds initially. The second premise, the induction step, states that whenever ϕ holds, it also holds at the following moment. The conclusion is thus that ϕ always holds.

To perform the induction step, we must show that ϕ is preserved across all atomic actions of the program. In practice this can often be determined by inspection, considering only the potentially falsifying transitions and ignoring those which obviously cannot make ϕ *false*.

As an example, we now verify safety for Peterson's solution ([Pe81]) to the mutual exclusion problem shown in Figure 8. Each process has a noncritical section (l_0, m_0, resp.) in which it idles unless it needs access to its critical section (l_3, m_3, resp.), signalled by entry into its trying region (l_1 and l_2, m_1 and m_2, resp.) Presence in the critical sections should be mutually exclusive. The safety property we wish to establish is thus that the system never reaches a state where both processes are in their respective critical sections at the same time: $\mathrm{G}(\neg(\mathrm{atl}_3 \wedge \mathrm{atm}_3))$.

It is helpful to establish several preliminary invariances. We use the notation $\mathrm{atl}_{1...3}$ to abbreviate $\mathrm{atl}_1 \vee \mathrm{atl}_2 \vee \mathrm{atl}_3$:

$$\begin{array}{lll}
\mathrm{G}\phi_1, & \phi_1: & y_1 \equiv \mathrm{atl}_{1...3} \\
\mathrm{G}\psi_1, & \psi_1: & y_2 \equiv \mathrm{atm}_{1...3} \\
\mathrm{G}\phi_2, & \phi_2: & \mathrm{atl}_3 \wedge \mathrm{atm}_2 \Rightarrow t \\
\mathrm{G}\psi_2, & \psi_2: & \mathrm{atm}_3 \wedge \mathrm{atl}_2 \Rightarrow t \\
\mathrm{G}\phi, & \phi: & \neg(\mathrm{atl}_3 \wedge \mathrm{atm}_3)
\end{array}$$

ϕ_1 plainly holds initially. Only transitions of process p_1 can affect it. Transitions $l_0 \rightarrow l_1$ leaves it true. Each of the other transitions of P_1 preserve its truth also, since y_1 is true whenever P_1 is at l_1, l_2, or l_3, and false when P_1 is at l_0. Thus $\mathrm{G}\phi_1$ is established.

A similar argument proves $\mathrm{G}\psi_1$.

ϕ_2 is vacuously true initially. The only potentially falsifying transitions for ϕ_2 are:

$l_1 \to l_2$ ensures atl_3 is *false* so ϕ_2 is preserved.

$l_2 \to l_3$ while atm_2—is enabled only when $\neg y_2 \vee t$ holds. Since y_2 is *true*, by virtue of ψ_1 and atm_2, it must be that t is *true* both before and after the transition. Hence ϕ_2 is preserved.

$m_1 \to m_2$ makes t *true* so that ϕ_2 is again preserved.

Thus $G\phi_2$ is established.

A similar argument establishes ψ_2.

Now to prove $G\phi$ we first note that ϕ holds initially. The only potentially falsifying transitions are in fact never enabled:

$l_2 \to l_3$ by process P_1 while process P_2 atm_3—By ψ_2, t is *false* and by ψ_1, y_2 holds. Since the enabling condition for the transition is $\neg y_2 \vee t$, the transition is never enabled.

$m_2 \to m_3$ by process P_2 while process P_1 atl_3—is similarly shown to be impossible.

Thus $G\phi$ (i.e, $G(\neg(atl_3 \wedge atm_3))$) is established.

We have the following liveness rule (LIVE), which is adequate for establishing eventualities based on a single step of a *helpful* process. Here we have formulae ϕ and ψ, and write $X_k p$ for $enabled_k \Rightarrow (executed_k \Rightarrow Xp)$, which means that the next execution of a step of process P_k will establish p. The rule is

$$\frac{\begin{array}{l} G(\phi \Rightarrow X(\phi \vee \psi)) \\ G(\phi \Rightarrow X_k\psi) \\ G(\phi \Rightarrow \psi \vee enabled_k) \end{array}}{G(\phi \Rightarrow F\psi)}$$

Often several invocations of LIVE must be linked together to prove an eventuality. We thus have the following rule, CHAIN:

$$\frac{G(\phi_i \Rightarrow F(\vee_{j<i}\phi_j \vee \psi))}{G(\vee_{i\leq k}\phi_i \Rightarrow F\psi)}$$

In many cases the rule CHAIN is adequate, in particular for finite state concurrent programs. In some instances, however, no a priori bound on the number of intermediate assertions ϕ_i can be given. We therefore use an assertion $\phi(a)$ with parameter a ranging over a given well-founded set $(W, <)$, which is a set W partially ordered by $<$ having no infinite decreasing sequence $a_1 > a_2 > a_3 > \ldots$. Note that this rule, WELL, generalizes the CHAIN rule, since we can take W to be the interval [1:k] with the usual ordering and $\phi(i) = \phi_i$.

$$\frac{G(\phi(a) \Rightarrow F(\exists b < a\ \phi(b) \vee \psi))}{G(\exists a \phi(a) \Rightarrow F\psi)}$$

We illustrate the application of the CHAIN rule on Peterson's [Pe81] algorithm for mutual exclusion. We wish to prove guaranteed accessibility:

$G(\mathrm{atl}_1 \Rightarrow \mathrm{Fatl}_3)$

(which is sometimes also called absence of starvation for process P_1), indicating that whenever process 1 wants to enter its critical section, it will eventually be admitted.

We define the following assertions

ψ : atl_3
ϕ_1 : $\mathrm{atl}_2 \wedge \mathrm{atm}_2 \wedge \mathrm{t}$
ϕ_2 : $\mathrm{atl}_2 \wedge \mathrm{atm}_1$
ϕ_3 : $\mathrm{atl}_2 \wedge \mathrm{atm}_0$
ϕ_4 : $\mathrm{atl}_2 \wedge \mathrm{atm}_3$
ϕ_5 : $\mathrm{atl}_2 \wedge \mathrm{atm}_2 \wedge \neg\mathrm{t}$
ϕ_6 : atl_1

and establishing the corresponding temporal implication by an application of the LIVE rule in order to meet the the hypothesis of the CHAIN rule:

$G(\phi_6 \Rightarrow F(\phi_5 \vee \phi_4 \vee \phi_3 \vee \phi_2))$, using helpful process P_1
$G(\phi_5 \Rightarrow F\phi_4)$, using helpful process P_2
$G(\phi_4 \Rightarrow F\phi_3)$, using helpful process P_2
$G(\phi_3 \Rightarrow F\phi_2 \vee \psi)$, using helpful process P_1
$G(\phi_2 \Rightarrow F\phi_1 \vee \psi)$, using helpful process P_2
$G(\phi_1 \Rightarrow F\psi)$, using helpful process P_1

The CHAIN rule now yields $G(\phi_6 \Rightarrow F\psi)$, i.e., $G(\mathrm{atl}_1 \Rightarrow \mathrm{Fatl}_3)$ as desired. The argument can be summarized in a *proof lattice* as depicted in Figure 9 (cf. [OL82], [MP82]).

7.3 Mechanical Synthesis of Concurrent Programs from Temporal Logic Specifications

One ambitious but promising possibility is that of automatically synthesizing concurrent programs from high-level specifications expressed in Temporal Logic. Here one deals with the *synchronization skeleton* of the program, which is an abstraction of the actual program where detail irrelevant to synchronization is suppressed. For example, in the synchronization skeleton for a solution to the critical section problem each process's critical section may be viewed as a single node since the internal structure of the critical section is unimportant. Most solutions to synchronization problems in the literature are in fact given as synchronization skeletons. Because synchronization skeletons are in general finite state, a propositional version of Temporal Logic suffices to specify their properties.

The synthesis method exploits the small model property of the propositional TL. It uses a decision procedure so that, given a TL formula, p, it will decide whether p is satisfiable or unsatisfiable. If p is satisfiable, a finite model of p is constructed. In this application, unsatisfiability of p means that the specification is inconsistent (and must be reformulated). If the formula p is satisfiable, then the specification it expresses is consistent. A model for p with a finite number of states is constructed by the decision procedure. The synchronization skeleton of a program meeting the specification can be read from this model. The small model property ensures that any program whose synchronization properties can be expressed in the TL can be realized by a system of concurrently running processes, each of which is a finite state machine.

One suitable logic is the branching time logic CTL. It has been used to specify and to synthesize, e.g., a starvation-free solution to the mutual exclusion problem (cf. [EC82]). Consider two processes P_1 and P_2, where each process is always in one of three regions of code: NCS_i—the *N*on*C*ritical *S*ection, TRY_i—the *TRY*ing Section, or CS_i—the *C*ritical *S*ection, which it cycles through, in order, repeatedly. When it is in region NCS_i, process P_i performs "noncritical" computations which can proceed in parallel with computations by other process P_j. At certain times, however, P_i may need to perform certain "critical" computations in the region CS_i. Thus, P_i remains in NCS_i as long as it has not yet decided to attempt critical section entry. When and if it decides to make this attempt, it moves into the region TRY_i. From there it enters CS_i as soon as possible, provided that the mutual exclusion constraint $\neg(atCS_1 \wedge atCS_2)$ is not violated. It remains in CS_i as long as necessary to perform its "critical" computations and then re-enters NCS_i.

It is assumed that only transitions between different regions of sequential code are recorded. Moves entirely within the same region are not considered in specifying synchronization. Moreover, the programs are running in a shared-memory environment with test-and-set primitives. The behavior of the system can be specified using the formulae listed below:

1. start state

   $atNCS_1 \wedge atNCS_2$

2. mutual exclusion

   $\mathrm{AG}\ (\neg(atCS_1 \wedge atCS_2))$

3. absence of starvation for P_i (i = 1,2)

   $\mathrm{AG}\ (atTRY_i \Rightarrow \mathrm{AF}\, atCS_i)$

plus some additional formulae to formally specify the information regarding the model of concurrent computation which was informally communicated in the above narrative. The global state transition diagram of a program meeting the conjunction of the above specifications, obtained by applying the synthesis method outlined, is shown in Figure 10. Solutions to other well known synchronization problems such as readers-writers and dining philophers can also be synthesized.

A closely related synthesis method for CSP programs based on the use of a decision procedure for PLTL was given in [MW84]. In the recent [PR89] a method for synthesizing an individual component of a reactive system from a specification in (essentially) CTL* is described. Earlier informal efforts toward synthesis of concurrent programs from TL-like formalisms include [La78] and [RK80].

There are a number of advantages to this type of automatic program synthesis method. It obviates the need to compose a program as well as the need to construct a correctness proof. Moreover, since it is algorithmic rather than heuristic in nature, it is both sound and complete. It is sound in that any program produced as a solution does in fact meet the specification. It is complete in that if the specification is satisfiable, a solution will be generated.

A drawback of this method is, of course, the (at least) exponential complexity of the decision procedure. Is this an insurmountable barrier to the development of this method into a practical software tool? Recall that while deciding satisfiability of propositional formulae requires exponential time in the worst case using the best known algorithms, the average case performance appears to be substantially better, and working automatic theorem provers and program verifiers are a reality. Similarly, the performance in practice of the decision procedure used by the synthesis method may be substantially better than the potentially exponential time worst case. (See [ESS89].) Furthermore, synchronization skeletons are generally small. It therefore seems conceivable that this approach may, in the long run, turn out to be useful in practical applications.

7.4 Automatic Verification of Finite State Concurrent Systems

The global state transition graph of a finite state concurrent system may be viewed as a finite temporal logic structure, and a model checking algorithm (cf. Section 6.3) can be applied to determine whether the structure is a model of a specification expressed as a formula in an appropriately chosen system of propositional TL. In other words, the model checking algorithm is used to determine whether a given finite state program meets a particular correctness specification. Provided that the model checking algorithm is efficient, this approach is potentially of wide applicability since a large class of concurrent programming problems have finite state solutions, and the interesting properties of many such systems can be specified in a propositional TL. For example, many network communication protocols can be modeled as a finite state system.

The basic idea behind this mechanical model checking approach to verification of finite state systems is to make brute force graph reachability analysis efficient and expressive through the use of TL as an assertion language. Of course, research in protocol verification has attempted to exploit the fact that protocols are frequently finite state, making exhaustive graph reachability analysis possible. The advantage offered by model checking seems to be that it provides greater flexibility in formulating specifications through the use of TL as a single, uniform assertion language that can express a wide variety of correctness properties. This makes it possible to reason about, e.g., both safety and liveness properties with equal facility.

Historically, [Pn77] showed that the problem of deciding truth of a linear temporal formula over a finite structure was decidable. However, his decision procedure was nonelementary, and the problem is PSPACE-complete in general (Theorem 6.17). The term “model checking” was coined by [CE81], who gave an efficient (polynomial time) model checking algorithm for the branching time logic CTL, and first proposed that it could be used as the basis of a practical automatic verification technique. At roughly the same time, [QS82] gave a model checking algorithm for a similar branching time logic, but did not analyze its complexity.

To illustrate how model checking algorithms work, we now describe a simple model checking algorithm for CTL. Note that it is similar to the global flow analysis algorithms used in compiler optimization. Assume that M = (S,R,L) is a finite structure and p_0 is a CTL formula. The goal is to determine at which states s of M, we have $M,s \models p_0$. The algorithm is designed to operate in stages: the 1st stage processes all subformulae of p_0 of length 1, the 2nd stage processes all subformulae of p_0 of length 2, and so on. At the end of the ith stage, each state will be labeled with the set of all subformulae of length $\leq$ i that are true at the state. To perform the labelling at stage i, information gathered in earlier stages is used. For example, subformulae $q \wedge r$ should be placed in the label of a state s precisely when q and r are both already in the label of s. For the modal subformula A[q U r], information from the successor states of s, as well as state s itself, is used. Since A[q U r] $\equiv$ q $\vee$ AXA[q U r], A[q U r] is initially added to the label of each state already labelled with r. Then satisfaction of A[q U r] is propagated outward, by repeatedly adding A[q U r] to the label of each state labelled by q and having A[q U r] in the label of all successors.

Let

$$(\mathrm{A}[\mathrm{q\ U\ r}])^1 = \mathrm{r}$$
$$(\mathrm{A}[\mathrm{q\ U\ r}])^{j+1} = \mathrm{r} \vee (\mathrm{q} \wedge \mathrm{AX}(\mathrm{A}[\mathrm{q\ U\ r}])^j)$$

It can be shown that M,s $\models$ (A[q U r])^j iff M,x $\models$ A[q U r] and along every path starting at s, r holds within distance j. Thus, states where (A[q U r])^1 holds are found first, then states where (A[q U r])^2 holds, etc. If A[q U r] holds, then (A[q U r])$^{card(S)}$ must hold since all loop-free paths in M are of length $\leq$ card(S). Thus, if after card(S) steps of propagating outward, A[q U r] has still not been found to hold at state s, then A[q U r] is false at s. Satisfaction of the other CTL modality E[p U q] propagates outward in the same fashion.

This version of the algorithm can be naively implemented to run in time linear in the length of p_0 and quadratic in the size of structure M. A more clever version of the algorithm can be implemented to run in time linear in the length of the input formula p and the size of M (cf. [CES86]).

```
for i = 1 to length(p0)
    for each subformula p of p0 of length i
        Case on the form of p
            p = P, an atomic proposition /* nothing to do */

            p = q ∧ r : for each s ∈ S
                                if q ∈ L(s) and r ∈ L(s) then
                                    add q ∧ r to L(s)
                             end
            p = ¬q :   for each s ∈ S
                                if q ∉ L(s) then
                                    add ¬q to L(s)
                            end
            p = EXq:  for each s ∈ S
                                if (for some successor t of s, q ∈ L(t)) then
                                    add EXq to L(s)
                            end
            p = A[q U r] : for each s ∈ S
                                    if r ∈ L(s) then
                                        add A[q U r] to L(s)
                                end
                            for j = 1 to Card(S)
                                for each s ∈ S
                                    if q ∈ L(s) and (for each successor t of s,
                                        A[q U r]∈L(t)) then add A[q U r] to L(s)
                                end
                            end
            p = E[q U r]: for each s ∈ S
                                    if r ∈ L(s) then
                                        add E[q U r] to L(s)
                                end
                for j = 1 to Card(s)
                                for each s ∈ S
                                    if q ∈ L(s) and (for some successor t of s,
                                        E[q U r] ∈ L(t)) then add E[q U r] to L(s)
                                end
                end
        end of case
    end
end
```

One limitation of the logic CTL is, of course, that it cannot express correctness under fair scheduling assumptions. However, the extended logic FairCTL described in [EL85] can express correctness under fairness (cf. [QS83]). An FCTL specification (p_0, Φ_0) consists of a functional assertion p_0, which is a state formula, and an underlying fairness assumption Φ_0, which is a pure path formula. The functional assertion p_0 is expressed in essentially CTL syntax with basic modalities of the form either A_Φ ("for all *fair* paths") or E_Φ ("for some *fair* path") followed by one of the linear time operators F, G, X, or U. The path quantifiers range over paths meeting the fairness constraint

Φ_0, which is a boolean combination of the infinitary linear time operators $\overset{\infty}{F}$ ("infinitely often") and $\overset{\infty}{G}$ ("almost always"), applied to propositional arguments. We can then view a subformula such as $A_\Phi FP$ of functional assertion p_0 as an abbreviation for the CTL* formula $A[\Phi_0 \Rightarrow FP]$. Similarly, $E_\Phi GP$ abbreviates $E[\Phi_0 \wedge GP]$. In this way FairCTL inherits its semantics from CTL*. Provided that Φ_0 is in the canonical form

$$\vee_{i=1}^{n} \wedge_{j=1}^{n_i} (\overset{\infty}{F} p_{ij} \vee \overset{\infty}{F} q_{ij})$$

then the model checking problem for FairCTL can be solved in time that is linear in the input structure size and small polynomial in the specification size.

Nevertheless, there are still correctness properties that one might like to describe that are not expressible within FairCTL, although they are describable in CTL* or even PLTL. The PSPACE-completeness of these latter logics, on first hearing, would seem to be a serious drawback. Lichtenstein and Pnueli [LP85] noted, however, that model checking is a problem with two input parameters, the structure and the specification, and then proceeded to develop a model checking algorithm for PLTL of complexity exponential in the the length of the specification but only linear in the size of the structure. They argued that since specifications are generally quite short while the structures representing programs are usually quite large, the exponential complexity in the specification size can be discounted. In practice, the dominating factor in the complexity should thus be the linear growth in the structure size.

It is worth pointing out that model checking, despite (because of?) its simplicity, is one approach to automatic verification that really seems to be useful in practice. It has been used to verify a large variety of finite state concurrent programs. These programs range from examples in the academic literature on concurrency to large-scale network communication protocols. For instance, a solution to the mutual exclusion problem given in [OL82] and proved correct there manually using linear TL is actually finite state. It was mechanically verified using the CTL model checking algorithm as described in [CES83]. Model checking is also applicable to the design of VLSI hardware and asynchronous circuits: Clarke has developed an efficient implementation of the CTL model checker along with various pieces of support software, which together forms the EMC (Extended Model Checker) system at CMU. In [MC86] the use of the EMC system resulted in the detection of a previously unknown error in a circuit for a self-time queue element published in the text [MC78]. Other applications to the design of sequential circuits are discussed in [BCD85], [BCDM86a], and [DC86], as well as the overview article [CG87]. Finally, model checking is applicable to large-scale network communication protocols. Indeed, one project in France [Si87] has bought dedicated hardware to use for model checking network protocols. Finite state systems with on the order of 10^5 states (and arcs) can currently be handled.

Despite the above practical successes, a potentially serious drawback to the entire model checking approach is that the size of the global state transition graph grows exponentially with the number of processes. Recent work in [CG86], [SG87], [CG87] suggests that it may be possible to avoid this exponential blowup in some cases for concurrent systems with many "copies" of the same process, although this is not possible in general (cf. [AK86]). Other work on reducing the size of the state graph based on hierarchical specification and hiding of states at lower levels of abstraction is presented in [MC85].

8 Other Modal and Temporal Logics in Computer Science

8.1 Classical Modal Logic

The class of Modal Logics was originally developed by philosophers to study different "modes" of truth. Such modes include possibility, necessity, obligation, knowledge, belief, and perception. Among the most important modes of truth are what "must be" true (necessity) and what "may be" true (possibility). For example, the assertion P may be false in the present world, and yet the assertion *possibly* P true, if there exists another world where P is true. The assertion *necessarily* P is true provided that P is true in all worlds.

Thus we have the well known *possible worlds* semantics of Kripke, where the truth value of a modal assertion at a world depends on the truth value(s) of its subassertions(s) at other possible worlds. This is formalized in terms of a *Kripke structure* M = (S,R,L) consisting of an underlying set S of possible *worlds*, also called *states*, an accessibility relation $R \subseteq S \times S$ between worlds, and a labelling L which provides an interpretation of primitive (i.e, nonmodal) assertions at each world. The technical definitions are such that *possibly* P is true at world s iff P is true at some world accessible from s, and *necessarily* P is true at world s iff P is true at all worlds accessible from s.

As we have seen, Temporal Logic is a particular kind of Modal Logic that has been specialized for reasoning about program behavior. Temporal Logic provides a much richer set of modalities, varying in how their truth value depends on which argument(s) hold(s) at which worlds, with the accessibility relation corresponding to the evolution of a concurrent system over time.

8.2 Propositional Dynamic Logic

An alternative development of a modal logic framework for program reasoning is represented by Dynamic Logic, originally proposed by Pratt [Pr76] in the first order version, specialized to the propositional version by Fischer and Ladner [FL79], and, in general studied intensively by Harel [Ha79] and others. (Detailed treatements of Dynamic logic can be found in [KT89] and [Ha84].) The basic modalities of Propositional Dynamic Logic (PDL) are of the form $<\alpha>$p where α is a regular expression over "atomic programs" and p is a formula. The intuitive meaning is that there exists an execution of program α leading from the present state to a state where p holds. PDL may be viewed as a propositional Branching Time Logic, with basic modalities of the form Eβ, where β is a regular expression over atomic propositions (node labels) and atomic programs (arc labels), and which means that there exists a path (a sequence of alternating states and arcs) starting at the present state that matches the regular expression β. A variety of extensions of PDL have been proposed in order to increase its expressive power. One is of particular interest to Temporal Logicians, viz., that with a repetition construct referred to a PDL + repeat or Δ-PDL. In Temporal Logic terms, its basic modalities are of the form Eβ, where β is an ω-regular expression such as $\alpha\gamma^\omega$. The ω iteration operator corresponds to the repeat operator Δ. Δ-PDL strictly subsumes CTL* in expressive power, and is thus able to express modalities such as AFp that cannot be expressed in ordinary PDL. Historically, Δ-PDL is important for reasons beyond its high expressive power. It is with Δ-PDL that automata-theoretic techniques for testing satisfiability were pioneered by Streett [St81].

8.3 Probabilistic Logics

Various probabilistic Temporal Logics have been proposed. For instance, [Lehmann & Shelah] describe logic, TC, with essentially the same syntax as CTL*, but where Ap means for almost all paths (i.e., for a set of paths of measure 1) p holds, and Ep means for significantly many paths (i.e, for a set of paths of positive measure) p holds. They give a deterministic double exponential time decision procedure and a sound and complete axiomitization for it. Interestingly, the logic TC has the same axiomitization as the logic MPL (Modal Process Logic) of Abrahamson [Ab80]. MPL has essentially the same syntax as CTL* but interprets it over more abstract structures, where the set of paths is not required to be generated by a binary relation. A probabalistic version of CTL is considered in [HS84].

8.4 Fixpoint Logics

Temporal operators can be characterized in terms of extremal fixpoints of monotonic functionals. Let M = (S,R,L) be a structure. We use PRED(S) to denote the lattice of total predicates over state set S, where each predicate is identified with the set of states which make it true and the ordering on predicates is set inclusion. Then a formula p defines a member of PRED(S), $\{ s \in S: M,s \models p\}$, and if it contains an atomic proposition Q, e.g., p(Q), it defines a function PRED(S) $\rightarrow$ PRED(S) where the value of p(Q) varies as Q varies. Let τ : PRED(S) $\rightarrow$ PRED(S) be given; then

- τ is said to be *monotonic* provided $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$
- τ is said to be $\cup$-*continuous* provided that $P_1 \subseteq P_2 \subseteq P_3 ...$ implies $\tau(\cup_i P_i) = \cup_i \tau(P_i)$.
- τ is said to be $\cap$-*continuous* provided that $P_1 \supseteq P_2 \supseteq P_3 ...$ implies $\tau(\cap_i P_i) = \cap_i \tau(P_i)$.

A predicate P is said to be a *fixpoint* of functional τ if $P = \tau(P)$. The theorem of Tarski-Knaster ([Ta55]) ensures that a monotonic functional τ: PRED(S) $\rightarrow$ PRED(S) always has a least fixpoint, $\mu Z.\tau(Z) = \cap\{ Y: \tau(Y) = Y\}$, and a greatest fixpoint $\nu Z.\tau(Z) = \cup\{ Y: \tau(Y) = Y\}$. Whenever τ is $\cup$-continuous then $\mu Z.\tau(Z) = \cup_i \tau^i(\mathit{false}$, and whenever τ is $\cap$-continuous then $\nu Z.\tau(Z) = \cap_i \tau^i(\mathit{true})$. (Note: $\tau^2(\mathit{false}) = \tau(\tau(\mathit{false}))$, etc.)

For example, shown below are the fixpoint characterizations for certain CTL modalities.

$$\begin{array}{ll} EFP = \mu Z.P \vee EXZ & AGP = \nu Z.P \wedge AXZ \\ AFP = \mu Z.P \vee AXZ & EGP = \nu Z.P \wedge EXZ \end{array}$$

Intuitively, the properties characterized as least fixpoints correspond to eventualities, while those characterized as greatest fixpoints are invariance properties

Assume for simplicity that each state in the underlying structure has a finite number of successors. Then each of the above functionals is both $\cup$-continuous and $\cap$-continuous in the argument Z, and we can readily establish the correctness for the above characterizations. For example, to show that $EFP = \mu Z.\tau(Z)$, with $\tau(Z) = P \vee EXZ$, it suffices to show that for each i (ranging over

$|N)$, $\tau^i(\mathit{false}) = \{$ states s in M : there exists a path of length i in M from state s to some state t such that M,t $\models$ P$\}$

These fixpoint characterizations are used in the model checking algorithm of section 7 and in the tableau-based decision procedure of section 6.

We can define an entire logic built-up from atomic proposition constants P, Q,..., atomic proposition variables Y,Z,... boolean connectives $\wedge$, $\vee$, $\neg$, the nextime operators EX, AX, and the least fixpoint μ and greatest fixpoint ν operators. We require that each formula be syntactically monotone, meaning that fixpoint formulae such as $\mu Z.\tau(Z)$ (or $\nu X.\tau(Z)$) are legal only when Z appears under an even number of negations within τ. The semantics is given in the obvious way suggested above.

Essentially this system was dubbed the "the Propositional Mu-Calculus" by Kozen [Ko83]. This Mu-Calculus has very considerable expressive power. It can encode (and in fact subsumes) CTL, FairCTL, CTL*, and, interpreted over multi-process structures, also PDL and PDL+repeat. It practical terms it also allows expression of extended modalities such as P is true at all even moments along all futures, which is captured by $\nu Z.P \wedge AXAXZ$. Related systems were considered in [EC80] and [PR81]. Other proposals for formalisms based on fixpoints can be found in, e.g., [deBS69], [Pa70], [deRo76], [Di76], and [Pa80].

8.5 Knowledge

There has recently been interest in the development of modal and temporal logics for reasoning about the states of knowledge in reactive systems. Knowledge can be especially important in the realm of distributed systems where processes are geographically dispersed and, at any given moment, possess only incomplete knowledge regarding the status of other processes in the system. Indeed, in many informal instances of reasoning about the behavior of distributed systems, it is a natural metaphor to refer to what a process *knows*. Logics of knowledge represent an effort to provide a formal basis for such reasoning.

A number of systems have been proposed (cf. [HM84], [Le84], [LR86], [DM86]). Typical modalities include K_ip which means that "process i knows p" and Cp which means that "p is common knowledge" in the sense that "all processes know p, all processes know that all processes know p, all processes know that all process know that all process know p, ..." These modalites of knowledge can be combined in various ways with temporal operators to permit reasoning about distributed systems. Certain semantic constraints, expressed as axioms (e.g. $K_i Xp \Rightarrow XK_i p$), are usually required. Subtle interactions between the syntax of the logic and the assumptions made regarding the model of distributed computation can lead to widely varying complexities for the decision problems of the resulting logics (cf. [HV86]). In general, this seems a promising area for future research. We refer the reader to the excellent survey by Halpern [Ha87] for an in-depth treatment.

Acknowledgement. We would like to thank Amir Pnueli for very helpful comments.

9 Bibliography

[AB80] Abrahamson, K., Decidability and Expressiveness of Logics of Processes, PhD Thesis, University of Washington, 1980.

[AK86] Apt, K. and Kozen, D., Limits for Automatic Verification of Finite State Systes, IPL vol. 22, no. 6., pp. 307-309, 1986.

[AS85] Alpern, B. and Schneider, F., Defining Safety and Liveness, Tech. Report, Cornell Univ., 1985.

[deBS69] de Bakker, J. and Scott, D., A Theory of Programs, Unpublished notes, IBM Seminar, Vienna, 1969.

[BKP84] Barringer, H., Kuiper, R., and Pnueli, A., Now You May Compose Temporal Logic Specifications, STOC84.

[BKP86] Barringer, H., Kuiper, R., and Pnueli, A., A Really Abstract Concurrent Model and its Temporal Logic, pp. 173-183, POPL86.

[BHP82] Ben-Ari, M, Halpern, J. Y., and Pnueli, A., Deterministic Propositional Dynamic Logic: Finite Models, Complexity, and Completeness, JCSS, v. 25, pp. 402-417, 1982

[BPM81] Ben-Ari, M., Pnueli, A. and Manna, Z. The Temporal Logic of Branching Time. ACM POPL 81; jounral version in Acta Informatica vol. 20, pp. 207-226, 1983.

[BCD85] Browne, M., Clarke, E. M., and Dill, D. Checking the Correctness of sequential Circuits, Proc. 1985 IEEE INt. Conf. Comput. Design, Port Chester, NY pp. 545-548

[BCDM86a] Browne, M., Clarke, E. M., and Dill, D., and Mishra, B., Automatic verification of sequential circuits using temporal logic, IEEE Trans. Comp. C-35(12), pp. 1035-1044, 1986

[Bu74] Burstall, R., Program Proving Considered as Hand Simulation plus Induction, IFIP Congress, Amsterdam, pp. 308-312, 1974.

[CE81] Clarke, E. M., Emerson, E.A., Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic, IBM Logics of Programs Workshop, Springer LNCS #131, pp. 52-71, May 1981.

[CES83] Clarke, E. M., Emerson, E. A., and Sistla, A. P., Automatic Verification of Finite State Concurrent System Using Temporal Logic, 10th ACM Symp. on Principles of Prog. Lang., Jan. 83; journal version appears in *ACM Trans. on Prog. Lang. and Sys.*, vol. 8, no. 2, pp. 244-263, April 1986.

[CG86] Clarke, E. M., Grumberg, O. and Browne, M.C., Reasoning about Networks with Many Identical Finite State Processes, Proc. 5th ACM PODC, pp. 240-248, 1986.

[CG87] Clarke, E. M. and Grumberg, O., Avoiding the State Explosion Problem In Temporal Model Checking, PODC87.

[CG87b] Clarke, E. M. and Grumberg, O. Research on Automatic Verification of Finite State Concurrent Systems, Annual Reviews in Computer Science, 2, pp. 269-290, 1987

[CM83] Clarke, E. M., Mishra, B., Automatic Verification of Asynchronous Circuits, CMU Logics of Programs Workshop, Springer LNCS #164, pp. 101-115, May 1983.

[CVW85] Courcoubetis, C., Vardi, M. Y., and Wolper, P. L., Reasoning about Fair Concurrent Programs, Proc. 18th STOC, Berkeley, Cal., pp. 283-294, May 86.

[Di76] Dijkstra, E. W. , A Discipline of Programming, Prentice-Hall, 1976.

[[DC86] Dill, D. and Clarke, E.M., Automatic Verification of Asynchronous Circuits using Temporal Logic, IEE Proc. 133, Pt. E 5, pp. 276-282, 1986.

[DM86] Dwork, C. and Moses, Y. Knowldge and Common Knowledge in a Byzantine Environment I: Crash Failures. In Proc. of the 1st Conf. on Theoretical Aspects of Reasoning about Knowledge, J. Y. Halpern ed., Los Altos, Cal., Morgan Kaufmann, pp 149-170, 1986.

[Em83] Emerson, E. A., Alternative Semantics for Temporal Logics, Theor. Comp. Sci., v. 26, pp. 121-130, 1983.

[EC80] Emerson, E. A., and Clarke, E. M., Characterizing Correctness Properties of Parallel Programs as Fixpoints. Proc. 7th Int. Colloquium on Automata, Languages, and Programming, Lecture Notes in Computer Science #85, Springer-Verlag, 1981.

[EC82] Emerson, E. A., and Clarke, E. M., Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons, *Science of Computer Programming*, vol. 2, pp. 241-266, Dec. 1982.

[EH85] Emerson, E. A., and Halpern, J. Y., Decision Procedures and Expressiveness in the Temporal Logic of Branching Time, *Journal of Computer and System Sciences*, vol. 30, no. 1, pp. 1-24, Feb. 85.

[EH86] Emerson, E. A., and Halpern, J. Y., 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time Temporal Logic, *JACM*, vol. 33, no. 1, pp. 151-178, Jan. 86.

[EL85] Emerson, E. A. and Lei, C. L., Modalities for Model Checking: Branching Time Strikes Back, pp. 84-96, ACM POPL85; journal version appears in Sci. Comp. Prog. vol. 8, pp 275-306, 1987.

[ES84] Emerson, E. A., and Sistla, A. P., Deciding Full Branching Time Logic, *Information & Control*, vol. 61, no. 3, pp. 175-201, June 1984.

[ESS89] Emerson, E. A., Sadler, T. H. , and Srinivasan, J. Efficient Temporal Reasoning, pp 166-178, 16th ACM POPL, 1989.

[En72] Enderton, H. B., A Mathematical Introduction to Logic, Academic Press, 1972.

[FL79] Fischer, M. J., and Ladner, R. E, Propositional Dynamic Logic of Regular Programs, JCSS vol. 18, pp. 194-211, 1979.

[Fr86] Francez, N., Fairness, Springer-Verlag, NY, 1986.

[FK84] Francez, N., and Kozen, D., Generalized Fair Termination, 11th Annual ACM Symp. on Principles of Programming Languages, 1984, pp. 46-53.

[GPSS80] Gabbay, D., Pnueli A., Shelah, S., Stavi, J., On The Temporal Analysis of Fairness, 7th Annual ACM Symp. on Principles of Programming Languages, 1980, pp. 163-173.

[Ha87] Halpern, J. Y., Using Reasoning about Knowledge to Analyze Distributed Systems, Annual Reviews in Computer Science, 2, pp. 37-68, 1987.

[HM84] Halpern, J. Y. and Moses, Y. Knowledge and Common Knowledge in a Distributed Environment, Proc. 3rd ACM Symp. PODC, pp. 50-61.

[Ha86] Halpern, J. Y. and Shoham, Y., A Propositional Modal Logic of Time Intervals, IEEE LICS, pp. 279-292, 1986.

[Ha79] Harel, D., Dynamic Logic: Axiomatics and Expressive Power, PhD Thesis, MIT, 1979; also available in Springer LNCS Series no. 68, 1979.

[HA84] Harel, D., Dynamic Logic, in Handbook of Philosophical Logic vol. II: Extensions of Classical Logic, ed. D. Gabbay and F. Guenthner, D. Reidel Press, Boston, 1984, pp. 497-604. Applications, 16th STOC, pp. 418-427, May 84.

[HS84] Hart, S. and Sharir, M. Probabalistic Temporal Logics for Finite and Bounded Models, 16th ACM STOC, pp. 1-13, 1984.

[Ho78] Hoare, C. A. R., Communicating Sequential Processes, CACM, vol. 21, no. 8, pp. 666-676, 1978.

[Ha82] Hailpern, B., Verifying Concurrent Processes Using Temporal Logic, Springer-Verlag LNCS no. 129, 1982.

[HO80] Hailpern, B. T., and Owicki, S. S., Verifying Network Protocols Using Temporal Logic, In Proceedings Trends and Applications 1980: Computer Network Protocols, IEEE Computer Society, 1980, pp. 18-28.

[HR74] Hossley, R., and Rackoff, C, The Emptiness Problem For Automata on Infinite Trees, Proc. 13th IEEE Symp. Switching and Automata Theory, pp. 121-124, 1972.

[HS84] Hart, S., and Sharir, M., Probabilistic Temporal Logics for Finite and Bounded Models, 16th STOC, pp. 1-13, 1984.

[HT87] Hafer, T., and Thomas, W., Computation Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree, ICALP87.

[Ka68] Kamp, Hans, Tense Logic and the Theory of Linear Order, PhD Dissertation, UCLA 1968.

[Ko87] Koymans, R., Specifying Message Buffers Requires Extending Temporal Logic, PODC87.

[Ko83] Kozen, D., Results on the Propositionnal Mu-Calculus, Theor. Comp. Sci., pp. 333-354, Dec. 83

[KP81] Kozen, D. and Parikh, R. An Elementary Proof of Completeness for PDL, Theor. Comp. Sci., v. 14, pp. 113-118, 1981

[KT87] Kozen, D. and Tiuryn, J., Logics of Programs, this volume

[Ku86] Kurshan, R. P. , Testing containment of omega regular languages, Tech. Report 1121-861010-33-TM, ATT Bell Labs, Murray Hill, NJ, 1986.

[LR86] Ladner, R. and Reif, J. The Logic of Distributed Protocols, in Proc. of Conf. On Theor. Aspects of reasoning about Knowledge, ed. J Halpern, pp. 207-222, Los Altos, Cal., Morgan Kaufmann

[LA80] Lamport, L., Sometimes is Sometimes "Not Never"—on the temporal logic of programs, 7th Annual ACM Symp. on Principles of Programming Languages, 1980, pp. 174-185.

[La83] Lamport, L., What Good is Temporal Logic?, Proc. IFIP, pp. 657-668, 1983.

[La78] Laventhal, M. Synthesis of Synchronization Code for Data Abstractions, PhD Thesis, MIT, 1978.

[Le84] Lehmann, D., Knowledge, Common Knowldge, and Related Puzzles, 3rd ACM Symp. PODC 84, pp 62-67.

[LPS81] Lehmann. D., Pnueli, A., and Stavi, J., Impartiality, Justice and Fairness: The Ethics of Concurrent Termination, ICALP 1981, LNCS Vol. 115, pp 264-277.

[LS82] Lehmann, D., and Shelah, S. Reasoning about Time and Chance, Inf. and Control, vol. 53, no. 3, pp. 165-198, 1982.

[LP85] Lichtenstein, O. and Pnueli, A., Checking that Finite State Concurrent Programs Satisfy their Linear Specification, POPL85, pp. 97-107, Jan. 85.

[LPZ85] Lichtenstein, O, Pnueli, A. ,and Zuck, L. The Glory of the Past, Brooklyn College Conference on Logics of Programs, Springer-Verlag LNCS, June 1985.

[MP82a] Manna, Z., and Pnueli, A., Verification of Concurrent Programs: The Temporal Framework, in The Correctness Problem in Computer Science, Boyer & Moore (eds.), Academic Press, pp. 215-273, 1982.

[MP81] Manna, Z. and Pnueli, A., Verification of Concurrent Programs: Temporal Proof Principles, in Proc. of Workshop on Logics of Programs, D. Kozen (ed.), Springer LNCS #131, pp. 200-252, 1981.

[MP82] Manna, Z. and Pnueli, A., Verification of Concurrent Programs: A Temporal Proof System, Proc. 4th School on Advanced Programming, Amsterdam, The Netherlands, June 82.

[MP83] Manna, Z. and Pnueli, A., How to Cook a Proof System for your Pet Language, ACM Symp. on Princ. of Prog. Languages, pp. 141-154, 1983.

[MP84] Manna, Z. and Pnueli, A., Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs, Science of Computer Programming, vol. 4, no. 3, pp. 257-290, 1984.

[MP87a] Manna, Z. and Pnueli, A. Specification and Verification of Concurrent Programs by $\forall$-automata, Proc. 14th ACM POPL, 1987

[MP87b] Manna, Z. and Pnueli, A. A Hierarchy of Temporal Properties, PODC7.

[MW78] Manna, Z., and Waldinger, R., Is "sometimes" sometimes better than "always"?: Intermittent assertions in proving program correctness, CACM, vol. 21, no. 2, pp. 159-172, Feb. 78

[MW84] Manna, Z. and Wolper, P. L., Synthesis of Communicating Processes from Temporal Logic Specifications, vol. 6, no. 1, pp. 68-93, Jan. 84.

[McN66] McNaughton, R., Testing and Generating Infinite Sequences by a Finite Automaton, Information and Control, Vol. 9, 1966.

[MP62] McNaughton, R. and Pappert, S. Counter-Free automata, MIT Press, 1962.

[MC80] Mead, C. and Conway, L., Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980.

[Me74] Meyer, A. R., Weak Monadic Second Order Theory of One Successor is Not Elementarily Recursive, Boston Logic Colloquium, Springer-Verlag Lecture Notes in Math. no. 453, Berlin/New York, 1974.

[Mo83] Moszkowski, B., Reasoning about Digital Circuits, PhD Thesis, Stanford Univ, 1983.

[MU63] Muller, D. E., Infinite Sequences and Finite Machines, 4th Ann. IEEE Symp. of Switching Theory and Logical Design, pp. 3-16, 1963.

[NDOG86] Nguyen, V., Demers, A., Owicki, S., and Gries, D., A Model and Temporal Proof System for Networks of Processes, Distr. Computing, vol. 1, no. 1, pp 7-25, 1986

[OL82] Owicki, S. S., and Lamport, L., Proving Liveness Properties of Concurrent Programs, ACM Trans. on Programming Languages and Syst., Vol. 4, No. 3, July 1982, pp. 455-495.

[Pa78] Parikh, R., A Decidability Result for Second Order Process Logic, Proc. 19th IEEE FOCS, pp. 177-183, 1978.

[Pa70] Park, D., Fixpoint Induction and Proof of Program Semantics, in Machine Intelligence, eds. B. Meltzer and D. Michie, vol. 5, pp. 59-78, Edinburgh Univ. Press, Edinburgh, 1970.

[PA80] Park, D., On The Semantics of Fair Parallelism, Abstract Software Specification, LNCS Vol. 86, Springer Verlag, 1980, pp. 504-524.

[PW84] Pinter, S., and Wolper, P. L., A Temporal Logic for Reasoning about Partially Ordered Computations, Proc. 3rd ACM PODC, pp. 28-37, Vancouver, Aug. 84

[Pe81] Peterson, G. L., Myths about the Mutual Exclusion Problem, Inform. Process. Letters, vol. 12, no. 3, pp. 115-116, 1981.

[PN77] Pnueli, A., The Temporal Logic of Programs, 18th annual IEEE-CS Symp. on Foundations of Computer Science, pp. 46-57, 1977.

[Pn81] Pnueli, A., The Temporal Semantics of Concurrent Programs, Theor. Comp. Sci., vol. 13, pp 45-60, 1981.

[PN83] Pnueli, A., On The Extremely Fair Termination of Probabilistic Algorithms, 15 Annual ACM Symp. on Theory of Computing, 1983, 278-290.

[Pn84] Pnueli, A., In Transition from Global to Modular Reasoning about Concurrent Programs, in Logics and Models of Concurrent Systems, ed. K. R. Apt, Springer, 1984.

[Pn85] Pnueli, A., Linear and Branching Structures in the Semantics and Logics of Reactive Systems, Proceedings of the 12th ICALP, pp. 15-32, 1985.

[Pn86] Pnueli, A., Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, in Current Trends in Concurrency: Overviews and Tutorials, ed. J. W. de Bakker, W.P. de Roever, and G. Rozenberg, Springer LNCS no. 224, 1986.

[PR76] Pratt, V., Semantical Considerations on Floyd-Hoare Logic, 17th FOCS, pp. 109-121, 1976.

[Pr79] Pratt, V., Models of Program Logics, Proc. 20th IEEE FOCS, pp. 115-122, 1979.

[PR81] Pratt, V., A Decidable Mu-Calculus, 22nd FOCS, pp. 421-427, 1981.

[Pr] Prior, A., Past, Present, and Future, Oxford Press.

[RU71] Rescher, N., and Urquhart, A., Temporal Logic, Springer-Verlag, 1971.

[QS82] Queille, J. P., and Sifakis, J., Specification and verification of concurrent programs in CESAR, Proc. 5th Int. Symp. Prog., Springer LNCS no. 137, pp. 195-220, 1982.

[QS83] Queille, J. P., and Sifakis, J., Fairness and Related Properties in Transition Systems, Acta Informatica, vol. 19, pp. 195-220, 1983.

[RK80] Ramarithram, K., and Keller, R., Specification and Synthesis of Synchronizers, Porc. 9th Int. Conf. on Par. Processing, pp. 311-321, 1980.

[deR76] de Roever, W. P., Recursive Program Schemes: Semantics and Proof Theory, Math. Centre Tracts no. 70, Amsterdam, 1976.

[Sa88] Safra, S., On the complexity of omega-automata, Proc. 29th IEEE FOCS, pp. 319-327, 1988.

[Si83] Sistla, A. P., Theoretical Issues in the Design of Distributed and Concurrent Systems, PhD Thesis, Harvard Univ., 1983.

[SC85] Sistla, A. P., and Clarke, E. M., The Complexity of Propositional Linear Temporal Logic, J. ACM, Vol. 32, No. 3, pp.733-749.

[SCFM84] Sistla, A. P., Clarke, E. M., Francez, N., and Meyer, A. R., Can Message Buffers be Axiomitized in Temporal Logic?, Information & Control, vol. 63., nos. 1/2, Oct./Nov. 84, pp. 88-112.

[SG87] Sistla, A. P., and German, S. M., Reasoning about Many Processes, LICS87.

[Si85] Sistla, A. P., Characterization of Safety and Liveness Properties in Temporal Logic, PODC85.

[SVW87] Sistla, A. P., Vardi, M. Y., and Wolper, P. L., The Complementation Problem for Buchi Automata with Applications to Temporal Logic, Theor. Comp. Sci., v. 49, pp 217-237, 1987.

[Si87] Sifakis, J., Personal Communication, April 87.

[SMS82] Schwartz, R., and Melliar-Smith, P. From State Machines to Temporal Logic: Specification Methods for Protocol Standards, IEEE Trans. on Communication, COM-30, 12, pp. 2486-2496, 1982.

[SMV83] Schwartz, R., Melliar-Smith, P. and Vogt, F. An Interval Logic for Higher-Level Temporal Reasoning, Proc. 2nd ACM PODC, Montreal, pp. 173-186, Aug. 83.

[ST82] Streett, R., Propositional Dynamic Logic of Looping and Converse, Information and Control 54, 121-141, 1982. (Full version: Propositional Dynamic Logic of Looping and Converse, PhD Thesis, MIT Lab for Computer Science, 1981.)

[SE84] Streett, R., and Emerson, E. A., The Propositional Mu-Calculus is Elementary, ICALP84, pp 465 -472, July 84.

[Ta55] Tarksi, A., A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific. J. Math., 55, pp. 285-309, 1955.

[Th79] Thomas, W., Star-free regular sets of omega-sequences, Information and Control, v. 42, pp. 148-156, 1979

[Th89] Thomas, W., Automata on Infinite objects, this volume

[Va85] Vardi, M., The Taming of Converse: Reasoning about Two-Way Computations, Proc. Workshop on Logics of Programs, Brooklyn, NY, LNCS no. 193, Springer-Verlag, pp. 413-424, 1985.

[Va87] Vardi, M., Verification of Concurrent Programs: The Automata-theoretic Framework, Proc. IEEE LICS, pp. 167-176, June 87

[VW83] Vardi, M. and Wolper, P., Yet Another Process Logic, in Proc. CMU Workshop on Logics of Programs, Springer LNCS no. 164, pp. 501-512, 1983.

[VW84] Vardi, M. and Wolper, P., Automata Theoretic Techniques for Modal Logics of Programs, STOC 84; journal version in *JCSS*, vol. 32, pp. 183-221, 1986.

[VW86] Vardi, M., and Wolper, P. , An Automata-theoretic Approach to Automatic Program Verification, Proc. IEEE LICS, pp. 332-344, 1986.

[Wo83] Wolper, P., Temporal Logic can be More Expressive, FOCS 81; journal version in Information and Control, vol. 56, nos. 1-2, pp. 72-93, 1983.

[Wo82] Wolper, P., Synthesis of Communicating Processes from Temporal Logic Specifications, Ph.D. Thesis, Stanford Univ., 1982.

[Wo85] Wolper, P., The Tableau Method for Temporal Logic: An Overview, Logique et Analyse, v. 28, June-Sept. 85, pp. 119-136, 1985.

[Wo86] Wolper, P., Expressing Interesting Properties of Programs in Propositional Temporal Logic, ACM Symp. on Princ. of Prog. Lang., pp. 184-193, 1986.

[Wo88] Wolper, P., On the Relation of Programs and Computations to Models of Temporal Logic, manuscript, Univ. of Liege, Belgium, Feb. 88

[Zu86] Zuck, L., Past Temporal Logic, PhD Dissertation, Weizmann Institute, 1986.

Contents