Automated Temporal Reasoning about Reactive Systems

E. Allen Emerson

University of Texas at Austin, Austin, Tx 78712, USA

Abstract. There is a growing need for reliable methods of designing correct reactive systems such as computer operating systems and air traffic control systems. It is widely agreed that certain formalisms such as temporal logic, when coupled with automated reasoning support, provide the most effective and reliable means of specifying and ensuring correct behavior of such systems. This paper discusses known complexity and expressiveness results for a number of such logics in common use and describes key technical tools for obtaining essentially optimal mechanical reasoning algorithms. However, the emphasis is on underlying intuitions and broad themes rather than technical intricacies.

1 Introduction

There is a growing need for reliable methods of designing correct reactive systems. These systems are characterized by ongoing, typically nonterminating and highly nondeterministic behavior. Examples include operating systems, network protocols, and air traffic control systems. There is widespread agreement that some type of temporal logic, or related formalism such as automata on infinite objects, provides an extremely useful framework for reasoning about reactive programs.

The "classical" approach to the use of temporal logic for reasoning about reactive programs is a manual one, where one is obliged to construct by hand a proof of program correctness using axioms and inference rules in a deductive system. A desirable aspect of some such proof systems is that they may be formulated so as to be "compositional", which facilitates development of a program hand in hand with its proof of correctness by systematically composing together proofs of constituent subprograms. Even so, manual proof construction can be extremely tedious and error prone, due to the large number of details that must be attended to. Hence, correct proofs for large programs are often very difficult to construct and to organize in an intellectually manageable fashion. It is not entirely clear that it is realistic to expect manual proof construction to be feasible for large-scale reactive systems.

We have historically advocated an alternative, automated approach to reasoning about reactive systems (cf. [Em81], [CE81], [EC82]).[1] The basic idea is that certain important questions associated with (propositional) temporal logic are decidable:

1. The Model Checking Problem - Given a finite state transition graph M and a temporal logic specification formula f, is M a model of f? This is useful in automatic verification of (finite state) reactive programs.

2. The Satisfiability Problem - Given a temporal logic specification formula f, does there exist a model M of f? This is useful in automatic synthesis of (finite state) reactive programs.

Thus automated program reasoning, in various forms, is possible in principle. The limiting factors are:

a. the complexity of the decision procedures, and

b. the expressiveness of the logics.

We will discuss these factors, their interaction with each other, and their impact on the promise and pitfalls of automated program reasoning. We will sketch known complexity and expressiveness results for a number of representative logics and formalisms. In particular, we shall focus on (i) Computation Tree Logic (CTL) and its variants such as CTL*, (ii) the Mu-Calculus, and (iii) tree automata. We shall also discuss key technical tools for obtaining essentially optimal decision procedures.

In the case of model checking, the upshot is that, in practice, the complexity as a function of the structure size is usually the dominating factor, and in this regard we have for most useful logics, in effect, efficient algorithms. For this reason, model checking has been applied quite successfully to verify correctness of (and to debug) "real-world" applications. A potentially serious limitation to model checking, however, is that of state space explosion: e.g., the global state graph of a concurrent system with n individual processes can be of size exponential in n. A variety of strategies for ameliorating this state explosion problem, including symbolic model checking and state reduction techniques, have been explored in the literature and remain a topic of current research interest (cf. [Ku94]).

In the case of testing satisfiability, for the rather simple logic CTL, a tableau construction suffices to obtain the Small Model Theorem, which asserts that any satisfiable formula f has a "small" (exponential size) model M; exponential time decidability follows. This permits us to synthesize reactive programs from CTL specifications in a wholly mechanical way. The model M defines a program meeting specification f. If f is inconsistent, the decision procedure says so, and the specification must be reformulated. A potentially serious drawback here is,

[1] It seems that nowadays there is widespread agreement that some type of automation is helpful, although this opinion is not unanimous.

of course, the complexity of the decision procedure. Somewhat surprisingly, a Simplified CTL (SCTL) with radically restricted syntax is still useful (e.g., for program synthesis) and is decidable in polynomial time.

For logics with greater expressive power, such as CTL* and the Mu-Calculus, we reduce satisfiability to nonemptiness of tree automata. Here pertinent technical tools include determinization and complementation of such automata on infinite strings and trees. Finally, we remark on some more intimate connections among tree automata, temporal logics, and the Mu-Calculus. For example, Mu-calculus formulae *are* simply alternating tree automata.

The remainder of this paper is organized as follows. Section 2 discusses preliminary material including the nature of reactive systems, temporal logic in general, and manual versus mechanical reasoning; also covered in section 2 are the technical definitions of the specific logics we focus on. A brief overview of model checking is given in section 3. The tableau-based approach to decision procedures is described and illustrated on CTL in section 4. The automata-theoretic approach to decision procedures is discussed and illustrated on CTL* in section 5. Section 6 discusses expressiveness and complexity issues, including a general overview, tradeoffs, and a summary of key results. Also considered in section 6 are efficiently decidable temporal logics. Finally, some concluding remarks are given in section 7.

2 Preliminaries

2.1 Reactive Systems

The ultimate focus of our concern is the development of effective methods for designing *reactive systems* (cf. [Pn86]). These are computer hardware and/or computer software systems that usually exhibit *concurrent* or *parallel* execution, where many individual processes and subcomponents are running at the same time, perhaps competing for shared resources, yet coordinating their activities to achieve a common goal. These processes and subcomponents may be geographically dispersed so that the computation is then *distributed*. The cardinal characteristic of reactive systems, however, is the *ongoing* nature of their computation. Ideally, reactive systems exhibit *nonterminating* behavior.

There are many important practical examples of reactive systems. These include: computer operating systems; network communication protocols; computer hardware circuits; automated banking teller networks; and air traffic control systems.

The semantics of reactive systems can thus be given in terms of infinite sequences of computation states. The computation sequences may in turn be organized into infinite computation trees. The branching behavior of these reactive systems is typically highly nondeterministic, owing to a variety of factors including the varying speeds at which processes execute, uncertainty over the time required for messages to be transmitted between communicating processes, and "random" factors in the environment. Because of this high degree of non-determinism, the behavior of reactive systems is to a high degree unpredictable, and certainly irreproducible in practice. For this reason, the use of testing as a means of ascertaining correctness of a reactive system is even more infeasible than it is for sequential programs. Accordingly, the use of appropriate formal methods for precisely specifying and rigorously verifying the correct behavior of reactive systems becomes even more crucial in the case of reactive systems.

2.2 Temporal Logic

One obvious difficulty is that formalisms originally developed for use with sequential programs that are intended to terminate, and are thus based on initial state – final state semantics are of little value when trying to reason about reactive systems, since there is in general no final state. Pnueli [Pn77] was the first to recognize the importance of ongoing reactive systems and the need for a formalism suitable for describing nonterminating behavior. Pnueli proposed the use of temporal logic as a language for specifying and reasoning about change over time. Temporal logic in its most basic form corresponds to a type of modal tense logic originally developed by philosophers (cf. [RU71]). It provides such simple but basic temporal operators as Fp (sometime p) and Gp (always p), that, Pnueli argued, can be combined to readily express many important correctness properties of interest for reactive systems.

Subsequent to the appearance of [Pn77], hundreds, perhaps thousands, of papers developing the theory and applications of temporal logic to reasoning about reactive systems were written. Dozens, if not hundreds, of systems of temporal logic have been investigated, both from the standpoint of basic theory and from the standpoint of applicability to practical problems. To a large extent the trend was to enrich (and "elaborate") Pnueli's original logic thereby yielding logics of increasingly greater expressive power. The (obvious) advantage is that more expressive logics permit handling of a wider range of correctness properties within the same formalism. More recently, there has been a counter-trend toward "simplified" logics tailored for more narrowly construed applications.

In any event, there is now a widespread consensus that some type of temporal logic constitutes a superior way to specify and reason about reactive systems. There is no universal agreement on just which logics are best, but we can make some general comments. temporal logics can be classified along a number of dimensions:

1. point-based, where temporal assertions are true or false of moments in time, versus interval-based, where temporal assertions are true or false of intervals of time.

2. future-tense only versus future-and-past-tense operators.
3. linear time, where temporal assertions are implicitly universally quantified over all possible executions of a reactive system, versus branching time, where temporal assertions include explicit path quantifiers and are interpreted over the computation trees describing system execution
4. propositional versus first order

It is our sense that the majority of the work on the use of temporal logic to reason about reactive systems has focussed on propositional, point-based, future-tense systems. There are a large number of users of both the linear time and the branching time frameworks. We ourselves have some preference for branching time, since in its full generality it subsumes the linear time framework. We shall thus focus on the systems CTL, CTL* discussed below along with the "ultimate branching time logic", the Mu-calculus, and PLTL, ordinary linear temporal logic. It is to be re-emphasized we are always restricting our attention to propositional logics, which turn out to be adequate for the bulk of our needs.

2.3 Manual versus Mechanical Reasoning

Because of the difficulty of establishing correctness of computer programs in general and the special subtleties associated with reactive systems in particular a great deal of of effort has gone into developing formal methods for reasoning about program correctness. There is a vast literature (cf. [MP92]) on the use of *manual* "proof systems", where a program's behavior is specified in some formal assertion language or logic, often a dialect of temporal logic, and then a rigorous mathematical proof is given, using axioms and inference rules in a deductive system for the logic, that the program meets its specification.

Manual proofs of program correctness offer both advantages and disadvantages. The advantages are rather obvious. In principle, it is possible to be completely formal in conducting the proof, thereby obtaining an absolute guarantee of soundness. In practice, working with complete, true formality is more problematic to achieve, and some level of quasi-formal argument is substituted instead.[2] These quasi-formal arguments can themselves still be useful, providing some degree of assurance that the program is – more or less – correct. Moreover, the quasi-formal arguments can mirror the informal style of a good human mathematician. Unfortunately, this does permit errors in the argument to creep in,

[2] This is more along the line of the working mathematician's notion of rigorous, but not strictly formal. To us, a strictly formal proof is conducted by simply performing symbol manipulations. There should exist an algorithm which, given a text that is alleged to constitute a strictly formal proof of an assertion, mechanically checks that each step in the proof is legitimate, say, an instance of an axiom or results from previous steps by application of an inference rule.

and it seems to us that complete formalization is, given the current state of knowledge about formal systems and notations, not likely to be practical in the manual setting.

We point out that there are additional drawbacks to manual proof construction of program correctness. Just as for any proof, ingenuity and insight are required to develop the proof. The problem is complicated by the vast amount of tedious detail that must be coped with, which must in general be organized in subtle ways in formulating "loop invariants", and so forth. There may be so much detail that it is difficult to organize the proof in an intellectually manageable fashion.

The upshot is that the whole task of manual proof construction becomes extremely error prone. One source of errors is that strong temptation to replace truly formal reasoning by quasi-formal reasoning. This is seen through-out the literature on manual program verification, and it frequently leads to errors.

We feel compelled to assert the following (perhaps controversial):

Claim. Manual verification will not work for large-scale reactive systems.

Our justification is that the task is error-prone to an overwhelming degree. Even if strictly formal reasoning were used throughout, the plethora of technical detail would be overwhelming. By analogy, consider the task of a human adding 100000 decimal numbers of 1000 digits each. This is rudimentary in principle but likely impossible in practice for any human to perform accurately. Similarly, the verification of 1000000 or even 100000 line programs by hand will not be feasible. The transcription errors alone will be prohibitive.

For these reasons plus the convenience of automation, we therefore believe that it is important to focus on mechanical reasoning about program correctness using temporal logic and related formalisms. There are at least four approaches to explore:

0. Mechanical *assistance* for verification of programs using a *validity* tester applied to assertions of the form $p_1 \wedge \ldots \wedge p_k \Rightarrow q$. Intuitively, $p_1, \ldots, p_k$ are assertions already proved to hold for the program and q is a new assertion to be established for the program.
1. Mechanical *verification* that a given finite state program M meets a specification p formulated in temporal logic, using a *model checking* algorithm (cf. [CE81], [QS82], [CES86], [Ku94]).
2. Mechanical *synthesis* of a program M meeting a temporal specification p using a decision procedure for testing *satisfiability* (cf. [EC82, MW84, PR89]).
3. *Executable* temporal logic specifications. This approach [BFGGO89] may be viewed as an elegant variation of the synthesis approach. While synthesis might be seen as a process of compiling temporal logic specifications, in contrast, this approach amounts to interpreting the specifications on-the-fly.

We note that approach 0, while less ambitious than approach 2, relies on the technical machinery of approach 2, a decision procedure for satisfiability/validity. Actually, it can be argued that approaches 1 and 3 also rely heavily on approach 2. In any event, in the sequel, we shall focus on approach 1, model checking, and approach 2, decision procedures for satisfiability.

2.4 CTL*, CTL, and PLTL

In this section we provide the formal syntax and semantics for three representative systems of propositional temporal logic. Two of these are branching time temporal logics: CTL and its extension CTL*. The simpler branching time 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], [QS82], [BPM83], and was itself proposed in [CE81]. However, its syntactic restrictions limit its expressive power so that, for example, correctness under fair scheduling assumptions cannot be expressed. 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 a number of other systems, including both CTL and PLTL.

The system PLTL (Propositional Linear temporal logic) is the "standard" linear time temporal logic widely used in applications (cf. [Pn77], [MP92]).

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, pUq

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: $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, $p \Leftrightarrow q$ abbreviates $p \Rightarrow q \wedge q \Rightarrow p$, Fp abbreviates $trueUq$, and Gp abbreviates $\neg F \neg p$. We also let $\overset{\infty}{F} p$ abbreviate GFp ("infinitely often"), $\overset{\infty}{G} p$ abbreviate FGp ("almost everywhere"), and (pBq) abbreviate $\neg((\neg p)Uq)$ ("before").

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, pUq 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 $E(trueUp)$, AGp abbreviates $\neg EF \neg p$, AFp abbreviates $A(trueUp)$, and EGp abbreviates $\neg AF \neg p$. (Note: this definition can be seen to be consistent with that of CTL*.)

Finally, the set of path formulae generated by rules S1,P1-3 define the syntax of the linear time logic PLTL.

Semantics

A formula of CTL* is interpreted with respect to a 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: $S \to 2^{AP}$ is a labeling which associates with each state s a set $L(s)$ consisting of all atomic proposition symbols in the underlying set of atomic propositions AP intended to be true at state s.

We may view M as a labeled, directed graph with node set S, arc set R, and node labels given by L.

A *fullpath* of M 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 it is not the case that $M, s_0 \models p$
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 it is not the case that $M, x \models \neg p$
P3 $M, x \models pUq$ 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.

Similarly, a formula of PLTL, which is a "pure path formula" of CTL* is interpreted using the above CTL* semantics.

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., $M, x \models p$).

Alternative and Generalized Semantics

We can define CTL* and other logics over various generalized notions of structure. For example, we could consider more general structures $M = (S, \mathbf{X}, L)$ where S is a set of states and L a labeling of states as usual, while $\mathbf{X} \subseteq S^\omega$ 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 $\mathbf{X}$, provided that "a fullpath x in M" is understood to refer to a fullpath x in $\mathbf{X}$. Usually, we want $\mathbf{X}$ to be the set of paths generated by a binary relation R (cf. [Em83]).

Another generalization is to define a *multiprocess structure*, which is a refinement of the above notion of a "monolithic" structure that distinguishes between different processes. Formally, a multiprocess 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 the proposition symbols at the state.

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

Now we can define nexttime operators relativized to process indices. For example, we can extend CTL to allow

$M, s \models EX_i p$ iff $\exists t$ ($(s,t) \in R_i$ and $M,t \models p$)
$M, s \models AX_i p$ iff $\forall t$ ($(s,t) \in R_i$ implies $M,t \models p$)

We can further generalize the semantics for CTL*. The previous formulation of CTL* over uniprocess structures refers only to the atomic formulae labeling 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 constraints, 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 $en_1, \ldots, en_k, ex_1, \ldots, ex_k$, where intuitively en_j is true of a state exactly when process j is enabled, i.e., when a transition by process j is possible, and ex_j is true of a transition when it is performed by process j. Technically, each en_j is an atomic proposition—and hence a state formula—true of exactly those states in domain R_j:

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

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

$M, x \models ex_j$ iff $d_1 = j$.

Fairness constraints (cf. [Fr86], [EL87]) can now be precisely captured:

- *Unconditional fairness*, which asserts that each process is executed infinitely often, can be expressed as $\wedge_{i \in [1:k]} (\overset{\infty}{F} ex_i)$.
- *Weak fairness*, which asserts that each process which is continuously enabled is repeatedly executed, can be expressed as $\wedge_{i \in [1:k]} (\overset{\infty}{G} en_i \Rightarrow \overset{\infty}{F} ex_i)$
- *Strong fairness*, which asserts that each process which is enabled infinitely often is executed infinitely often can be expressed as $\wedge_{i \in [1:k]} (\overset{\infty}{F} en_i \Rightarrow \overset{\infty}{F} ex_i)$

We can define a single type of general structure which subsumes all of those above. We assume an underlying set of *state symbols* 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* $\mathrm{M} = (S, R, \mathbf{X}, L)$ where

S is a set of *states*,
R is a total *binary relation* $\subseteq S \times S$,
$\mathbf{X}$ 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 $\mathbf{X}$, let $R = \{(s,t) \in S \times S$: there is a fullpath of the form $ystz$ in $\mathbf{X}$ 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 $\mathbf{X}$ 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 $\mathbf{X}$. If d is an arc assertion we have that:
$M, x \models d$ iff $d \in L((s_0, s_1))$

Alternative Syntax

Here the essential idea is that of a *basic modality*. A formula of CTL* is a basic modality provided that it is of the form Ap or Ep where p itself contains no A's or E's, i.e., p is an arbitrary formula of PLTL. Similarly, a basic modality of CTL is of the form Aq or Eq where q is one of the *single* linear temporal operators F, G, X, or U applied to pure propositional arguments. A CTL* (respectively, CTL) formula can now be thought of as being built up out of boolean combinations and nestings of basic modalities (and atomic propositions).

2.5 Mu-calculus

CTL* provides one way of extending CTL. In this section we describe another way of extending CTL. We can view CTL as a sublanguage of the *propositional Mu-Calculus* $L\mu$ (cf. [Ko83], [EC80]). The propositional Mu-Calculus provides a *least fixpoint* operator (μ) and a *greatest fixpoint* operator (ν) which make it possible to give *fixpoint characterizations* of the branching time modalities. Intuitively, the Mu-Calculus makes it possible to characterize the modalities in terms of recursively defined tree-like patterns. For example, the CTL assertion

EFp (along some computation path p will become true eventually) can be characterized as $\mu Z.p \vee EXZ$, the least fixpoint of the functional $p \vee EXZ$ where Z is an atomic proposition variable (intuitively ranging over sets of states) and EX denotes the existential nexttime operator.

We first give the formal definition of the Mu-Calculus.

Syntax

The formulae of the propositional Mu-Calculus $L\mu$ are those generated by rules (1)-(6):

(1) Atomic proposition constants P, Q
(2) Atomic proposition variables $Y, Z, \ldots$
(3) EXp, where p is any formula.
(4) $\neg p$, the negation of formula p.
(5) $p \wedge q$, the conjunction of formulae p, q.
(6) $\mu Y.p(Y)$, where $p(Y)$ is any formula syntactically monotone in the propositional variable Y, i.e., all free[3] occurrences of Y in $p(Y)$ fall under an even number of negations.

The set of formulae generated by the above rules forms the language $L\mu$. The other connectives are introduced as abbreviations in the usual way: $p \wedge q$ abbreviates $\neg(\neg p \wedge \neg q)$, AXp abbreviates $\neg EX \neg p$, $\nu Y.p(Y)$ abbreviates $\neg \mu Y. \neg p(\neg X)$, etc. Intuitively, $\mu Y.p(Y)$ ($\nu Y.p(Y)$) stands for the least (greatest, resp.) fixpoint of $p(Y)$, EXp (AXp) means p is true at some (every) successor state reachable from the current state, $\wedge$ means "and", etc. We use $|p|$ to denote the *length* (i.e., number of symbols) of p.[4]

We say that a formula q is a *subformula* of a formula p provided that q, when viewed as a sequence of symbols, is a substring of p. A subformula q of p is said to be *proper* provided that q is not p itself. A *top-level* (or *immediate*) subformula is a maximal proper subformula. We use $SF(p)$ to denote the set of subformulae of p.

The fixpoint operators μ and ν are somewhat analogous to the quantifiers $\exists$ and $\forall$. Each occurrence of a propositional variable Y in a subformula $\mu Y.p(Y)$ (or $\nu Y.p(Y)$) of a formula is said to be *bound*. All other occurrence are *free*. By renaming variables if necessary we can assume that the expression $\mu Y.p(Y)$ (or $\nu Y.p(Y)$) occurs at most once for each Y.

A *sentence* (or *closed* formula) is a formula that contains no free propositional variables, i.e., every variable is bound by either μ or ν. A formula is said to be in *positive normal form* (PNF) provided that no variable is quantified twice

[3] Defined below.

[4] Alternatively, we can define $|p|$ as the size of the syntax diagram for p.

and all the negations are applied to atomic propositions only. Note that every formula can be put in PNF by driving the negations in as deep as possible using DeMorgan's Laws and the dualities $\neg\mu Y.p(Y) = \nu Y.\neg p(\neg Y)$, $\neg\nu Y.p(Y) = \mu Y.\neg p(\neg Y)$. (This can at most double the length of the formula). *Subsentences* and *proper subsentences* are defined in the same way as subformulae and proper subformulae.

Let σ denote either μ or ν. If Y is a bound variable of formula p, there is a unique μ or ν subformula $\sigma Y.p(Y)$ of p in which Y is quantified. Denote this subformula by σY. Y is called a *μ-variable* if $\sigma Y = \mu Y$; otherwise, Y is called a *ν-variable*. A *σ-subformula* (*σ-subsentence*, resp.) is a subformula (subsentence) whose main connective is either μ or ν. We say that q is a *top-level* σ-subformula of p provided q is a proper σ-subformula of p but not a proper σ-subformula of any other σ-subformula of p. Finally, a *basic modality* is a σ-sentence that has no proper σ-subsentences.

Semantics

We are given a set Σ of atomic proposition constants and a set Γ of atomic proposition variables. We let AP denote $\Sigma \cup \Gamma$. Sentences of the propositional Mu-Calculus $L\mu$ are interpreted with respect to a structure $M = (S, R, L)$ as before.

The power set of S, 2^S, may be viewed as the complete lattice (2^S, $S, \emptyset$, $\subseteq$, $\cup$, $\cap$). Intuitively, we identify a predicate with the set of states which make it true. Thus, *false* which corresponds to the empty set is the bottom element, *true* which corresponds to S is the top element, and implication ($\forall s \in S[P(s) \Rightarrow Q(s)]$) which corresponds to simple set-theoretic containment ($P \subseteq Q$) provides the partial ordering on the lattice.

Let $\tau : 2^S \to 2^S$ be given; then we say that τ is *monotonic* provided that $P \subseteq Q$ implies $\tau(P) \subseteq \tau(Q)$.

Theorem (Tarski-Knaster). Let $\tau : 2^S \to 2^S$ be a monotonic functional. Then

(a) $\mu Y.\tau(Y) = \cap \{Y : \tau(Y) = Y\} = \cap \{Y : \tau(Y) \subseteq Y\}$,
(b) $\nu Y.\tau(Y) = \cup \{Y : \tau(Y) = Y\} = \cup \{Y : \tau(Y) \supseteq Y\}$,
(c) $\mu Y.\tau(Y) = \cup_i \tau^i(\mathit{false})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$, and
(d) $\nu Y.\tau(Y) = \cap_i \tau^i(\mathit{true})$ where i ranges over all ordinals of cardinality at most that of the state space S, so that when S is finite i ranges over $[0{:}|S|]$.

A formula p with free variables $Y_1, \ldots, Y_n$ is thus interpreted as a mapping p^M from $(2^S)^n$ to 2^S, i.e., it is interpreted as a predicate transformer. We write

$p(Y_1, \ldots, Y_n)$ to denote that all free variables of p are among $Y_1, \ldots, Y_n$. A *valuation* $\mathcal{V}$, denoted $(V_1, \ldots, V_n)$, is an assignment of the subsets of S, $V_1, \ldots, V_n$, to free variables $Y_1, \ldots, Y_n$, respectively. We use $p^M(\mathcal{V})$ to denote the value of p on the (actual) arguments $V_1, \ldots, V_n$ (cf. [EC80], [Ko83]). The operator p^M is defined inductively as follows:

(1) $P^M(\mathcal{V}) = \{s : s \in S \text{ and } P \in L(s)\}$ for any atomic propositional constant $P \in AP$
(2) $Y_i^M(\mathcal{V}) = V_i$
(3) $(p \wedge q)^M(\mathcal{V}) = p^M(\mathcal{V}) \cap q^M(\mathcal{V})$
(4) $(\neg p)^M(\mathcal{V}) = S \backslash (p^M(\mathcal{V}))$
(5) $(EXp)^M(\mathcal{V}) = \{s : \exists t \in p^M(\mathcal{V}), (s,t) \in R\}$
(6) $\mu Y_1.p(Y_1)^M(\mathcal{V}) = \cap\{S' \subseteq S : p(Y_1)^M(S', V_2, \ldots, V_n) \subseteq S'\}$

Note that our syntactic restrictions on monotonicity ensure that least (as well as greatest) fixpoints are well-defined.

Usually we write $M, s \models p$ (respectively, $M, s \models p(\mathcal{V})$) instead of $s \in p^M$ (respectively, $s \in p^M(\mathcal{V})$) to mean that sentence (respectively, formula) p is true in structure M at state s (under valuation $\mathcal{V}$). When M is understood, we write simply $s \models p$.

Extensions

Just as for CTL and CTL*, we have the multiprocess versions of the Mu-calculus. One possible formulation is to use $EX_i p$ for "there exists a successor state satisfying p, reached by some step of process i". Dually, we then also have $AX_i p$. The classical notation, going back to PDL [FL79], would write <i> p and [i]p, respectively.

Discussion

We can get some intuition for the the Mu-Calculus by noting the following extremal fixpoint characterizations for CTL properties:

$$EFP \equiv \mu Z.P \vee EXZ$$
$$AGP \equiv \nu Z.P \wedge AXZ$$
$$AFP \equiv \mu Z.P \vee AXZ$$
$$EGP \equiv \nu Z.P \wedge EXZ$$
$$A(P\ U\ Q) \equiv \mu Z.Q \vee (P \wedge AXZ)$$
$$E(P\ U\ Q) \equiv \mu Z.Q \vee (P \wedge EXZ)$$

For these properties, as we see, the fixpoint characterizations are simple and plausible. It is not too difficult to give rigorous proofs of their correctness [EC80],

[EL86]. However, it turns out that it is possible to write down highly inscrutable Mu-calculus formulae for which there is no readily apparent intuition regarding their intended meaning. As discussed subsequently, the Mu-calculus is a very rich and powerful formalism so perhaps this should come as no surprise. We will comment here that Mu-calculus formulas are really representations of alternating finite state automata on infinite trees (see section 6.5). Since even such basic automata as *deterministic* finite state automata on *finite strings* can be quite complex "cans of worms", we should again not be so surprised at potential inscrutability. On the other hand, many Mu-calculus characterization of correctness properties are elegant, and the formalism seems to have found increasing favor, especially in Europe, owing to its simple and elegant underlying mathematical structure.

One interesting measure of the "structural complexity" of a Mu-calculus formula is its *alternation depth*. Intuitively, the alternation depth refers to the depth of nesting of alternating μ's and ν's. The alternation must be "significant", entailing a subformula of the forms

$$(*)\ \mu Y.p(\nu Z.q(Y,Z)) \text{ or } \nu Y.p(\mu Z.q(Y,Z))$$

where, for example, a μ'd Y occurs within the scope of νZ or ν'd Y occurs within the scope of a μZ.

All the basic modalities AFq, AGq, EFq, etc. of CTL can be expressed in the Mu-Calculus with alternation depth 1 as illustrated above. So can all CTL formula. For example, $EFAGq$ has the Mu-Calculus characterization $\mu Y.(EXY \vee \nu Z.(P \wedge AX\,Z))$, which is still of alternation depth 1 since while νZ appears inside μY, the "alternation" does not have Y inside νZ and does not match the above form (*). A property such as $E(P^*Q)^*R$, meaning there exists a path matching the regular expression $(P^*Q)^*R$, can be expressed by $\mu Y.\mu Z.(P \wedge EXZ \vee (Q \wedge EXY) \vee R$, which is still of alternation depth 1.

On the other hand, properties associated with fairness require alternation depth 2. For example, $E\overset{\infty}{F}P$ (along some path P occurs infinitely often) can be characterized by $\nu Y.\mu Z.EX(P \wedge Y \vee Z)$. It can be shown that $E\overset{\infty}{F}P$ is not expressible by any alternation depth 1 formula (cf. [EC80] [EL86]).

Let $L\mu_k$ denote the Mu-Calculus $L\mu$ restricted to formulas of alternation depth at most k. It turns out that most all modal or temporal logics of programs can be translated into $L\mu_1$ or $L\mu_2$, often succinctly (cf. [EL86]). Interestingly, it is not known if higher alternation depths form a true hierarchy of expressive power. The question has some bearing on the complexity of model checking in the overall Mu-calculus as discussed in section 7.

3 Model Checking

One of the more promising techniques in automated temporal reasoning about reactive systems began with the advent of efficient temporal logic *model checking* [CE81] (cf. [Em81], [QS82], [CES86]). The basic idea is that the global state transition graph of a finite state reactive system defines a (Kripke) structure in the sense of temporal logic (cf. [Pn77]), and we can give an *efficient* algorithm for checking if the state graph defines a model of a given specification expressed in an appropriate temporal logic. While earlier work in the protocol community had addressed the problem of analysis of simple reachability properties, model checking provided a powerful, uniform specification language in the form of temporal logic along with a single, efficient verification algorithm which automatically handled a wide variety of correctness properties, including both safety and liveness properties, with equal ease.

Technically, the Tarski-Knaster theorem can be understood as providing a systematic basis for model checking. The specifications can be formulated in the Mu-calculus or in other logics such as CTL which, as noted above, are readily translatable into the Mu-calculus. For example, to calculate the states where the CTL basic modality EFP holds in structure $M = (S, R, L)$, we use the fixpoint characterization $EFP \equiv \mu Z.\tau(Z)$, with $\tau(Z) \equiv P \vee EXZ$. We successively calculate the ascending chain of approximations

$$\tau(false) \subseteq \tau^2(false) \subseteq \ldots \subseteq \tau^k(false)$$

for the least $k \leq |S|$ such that $\tau^k(false) = \tau^{k+1}(false)$. The intuition here is just that each $\tau^i(false)$ corresponds to the set of states which can reach P within at most distance i; thus, P is reachable from state s iff P is reachable within i steps from s for some i less than the size of M iff $s \in \tau^i(false)$ for some such i less than the size of M. This idea can be easily generalized to provide a straightforward model checking algorithm for all of CTL and even the entire Mu-calculus.[5] The Tarski-Knaster theorem handles the basic modalities. Compound formulae built up by nesting and boolean combinations are handled by recursive descent. A naive implementation runs in time complexity $O((|M||p|)^{k+1})$ for input structure M and input formula p with μ, ν formulas nested k deep. Some improvements are possible as described in section 6.3.

The above model checking algorithm has been dubbed a *global* algorithm because it computes for the (closed) formula f over structure M the set f^M of all states of M where f is true. A technical characteristic of global model checking algorithms is that with each subformula g of the specification f there is calculated the associated set of state g^M. A potential practical drawback is that all the states of the structure M are examined.

[5] Model checking for PLTL and CTL* can be performed as discussed in section 6.3.

In practice, we are often only interested in checking truth of f at a particular state s_0. This gives rise to *local model checking* algorithms which in the best case may not examine all states of the structure M, although in the worst case they may have to (cf. [SW89]).

In this connection, note that model checkers are a type of decision procedure and provide yes/no answers. It turns out that, in practice, model checkers are often used for debugging as well as verification. In industrial environments it seems that the capacity of a model checker to function as a debugger is perhaps better appreciated than their utility as a tool for verifying correctness.

Consider the empirical fact that most designs are initially wrong and must go through a sequence of corrections/refinements before a truly correct design is finally achieved. Suppose one aspect of correctness that we wish to check is that a simple invariance property of the form $AGgood$ holds provided the system M is started in the obviously *good* initial state s_0. It seems quite likely that the invariance may in fact not hold of the initial faulty design due to conceptually minor but tricky errors in the fine details.[6] Thus, during many iterations of the design process, we have that in fact $M, s_0 \models EF\neg good$.

It would be desirable to circumvent the global strategy of examining all of M to calculate the set $EF\neg good^M$ and then checking whether s_0 is a member of that set. If there does exist a $\neg good$ state reachable from s_0, once it is detected it is no longer necessary to continue the search examining M. This is the heuristic underlying local model checking algorithms. Many of them involve depth first search from the start state s_0 looking for confirming or refuting states or cycles; once found, the algorithm can terminate often "prematurely" having determined that the formula must be true or must be false at s_0 on the basis of the portion M examined during the limited search.

Of course, it may be that all states must be examined before finding a refutation to $AGgood$. Certainly, once a truly correct design is achieved, all states reachable from s_0 must be examined. But in many practical cases, a refutation may be found quickly after limited search.

All the above discussion concerns what is referred to as *extensional* model checking or as *explicit state* model checking, since it is assumed that the structure M including all of its nodes and arcs explicitly represented using data structures such as adjacency lists or adjacency matrices. An obvious limitation then is the *state explosion* problem. Given a reactive system composed on n sequential processes running in parallel, its global state graph will be essentially the product of the individual local process state graphs. The number of global states thus grows exponentially in n. For particular systems it may happen that the final global state graph is of a tractable size, say a few hundred thousand states plus transitions. A number of practical systems can be modeled at a useful level of

[6] This is a scenario where model checkers should be particularly useful.

abstraction by state graphs of this size, and extensional model checking can be a helpful tool. On the other hand, it can quickly become infeasible to represent the global state graph for large n. Even a banking network with 100 automatic teller machines each having just 10 local states, could yield a global state graph of astronomical size amounting to about 10^{100} states.

A major advance has been the introduction of *symbolic* model checking techniques (cf. [McM92], [Pi90], [CM90]) which are – in practice– often able to succinctly represent and model check over state graphs of size 10^{100} states and even considerably larger. The key idea is represent the state graph in terms of a boolean characteristic function which is in turn represented by a Binary Decision Diagram (BDD) (cf. [Br86]). BDD-based model checkers have been remarkably effective and useful for debugging and verification of hardware circuits. For reasons not well understood, BDDs are able to exploit the regularity that is readily apparent even to the human eye in many hardware designs. Because software typically lacks this regularity, BDD-based model checking seems less helpful for software verification. We refer the reader to [McM92] for an extended account of the utility of BDDs in hardware verification.

There has thus been great interest in model checking methods that avoid explicit construction of the global state graph. In some cases, it is possible to give methods that work directly on the program text itself as an implicit representation of the state graph. Some approaches use process algebras such as CCS or formalisms such as Petri-nets to represent programs. This facilitates succinct representation of (possibly infinite) families of states and exploitation of the compositional structure of programs (cf. [BS92]) in performing a more general type of local model checking. The drawback is that the general method can no longer be fully automated, for such basic reasons as the unsolvability of the halting problem over infinite state spaces. Still, such partially automated approaches are intriguing.

4 Decision Procedures I: Tableaux-theoretic Approach

4.1 Overview

In this section we discuss decision procedures for testing satisfiability of temporal logic formulas.

Basic Idea 0: Prove the *small model property* for the logic, which asserts that if a formula f is satisfiable then it has a finite model of size bounded by some function of the length of f. This then yields a nondeterministic algorithm:

- Guess a candidate model of bounded size.
- Check it for genuineness.

For many logics the process can be improved:
Basic Idea 1: Eliminate the nondeterminism:

- Build a small *tableau* graph encoding (essentially all) potential models of f.
- "Pseudo-model-check" the tableau, ensuring that it contains a structure in which all eventualities are fulfillable; since invariances turn out to take care of themselves in tableau, this structure defines a genuine model of f.

For many logics the tableau method is natural, powerful, and appropriate. CTL is such a logic. A key advantage is that there is a fairly direct correspondence between the organization of the tableau and the syntax of the formula. However, for some logics (such as CTL*) the tableau method, as normally conceived, breaks down. There is no apparent way to derive a natural tableau graph encoding all possible models of a formula from its syntax. Instead we can fall back upon what is widely viewed as the standard paradigm today:
Basic Idea 2: The automata-theoretic approach:

- Build a finite state automaton on infinite objects (strings or trees) accepting (essentially all) models of f.
- Test it for nonemptiness.

Basic Idea 2 subsumes Basic Idea 1. For logics to which the tableau method is applicable, the tableau can be viewed as defining an automaton. For other logics, we can build an automaton when it is not possible to build a tableau because we can appeal to certain difficult combinatorial constructions from the theory of automata on infinite objects. The most important of these is *determinization* of automata.[7] The power of automata theory seems to lie in a reservoir of deep combinatorial results that show how to construct an automaton corresponding to a formula, even though that correspondence is by no means apparent. Of course, this opacity could be viewed as a drawback, but it seems to be inherent in the problem.

4.2 Tableau-based Decision Procedure for CTL

The basic idea underlying the tableau method is as follows. Given a formula f, we initially construct a tableau graph T, based on systematic case analysis. The nodes of T are essentially sets of subformulae of f. The "meaning" of a node is the conjunction of all its associated subformulas, and a node is thought of as a state satisfying those subformulas in a potential model of f. The nodes of T

[7] It seems that in most cases complementation is what is really needed. Complementation for infinite string automata can be accomplished without determinization (cf. e.g. [SVW87]). However, for automata on infinite trees it is not clear that complementation can be performed without determinization of ω-string automata.

thus correspond to a partitioning/covering of the state space of potential models, based on which subformulae do and do not hold at a state. We then successively prune from T nodes which are inconsistent; for example, nodes which assert that a formula is presently both true and false, or nodes which assert that something eventually happens while it really does not. After all such pruning is performed whatever remains is the final tableau which encodes all potential models, if any, of f. Below we describe the method in greater detail.

We build an initial tableau T_0 which is a bipartite graph consisting of OR-nodes D and AND-nodes C, where each node is a set of formulae whose "meaning" is their conjunction. We will then prune the tableau, deleting inconsistent nodes.

T_0 is constructed starting with $D_0 = \{f_0\}$. In general, we have a method, described below, of decomposing an OR-node D to get its set of AND-node successors C_i, and another method for constructing the OR-node successors D_j of AND-node C. If any set C_i already appears in the the tableau, we make it a successor of D. Thus, a set of formulae appears at most once as an AND-node in the tableau. Similarly for OR-nodes. This bounds the size of T_0.

We perform *α–β-expansion* to systematically decompose a given OR-node D that we are testing for satisfiability into boolean combinations of *elementary* formulae, which are atomic propositions P, their negations $\neg P$, and nexttime formulae AXf, EXf. In this way, D is shown to be equivalent to certain boolean combinations of assertions about *now* and *nexttime*. To do this decomposition, we note that each *nonelementary* formula may be classified as either a conjunctive formula $\alpha \equiv \alpha_1 \wedge \alpha_2$ or a disjunctive formula $\beta \equiv \beta_1 \vee \beta_2$. Plainly, $f \wedge g$ is an α formula, while $f \vee g$ is a β formula. A temporal modality is classified as α or β on the basis on its fixpoint characterization in the Mu-calculus. For example, (assuming the structure is total) $AFh \equiv h \vee AXAFh$ is a β formula, while $AGh \equiv h \wedge AXAGh$ is an α formula.

Thus, given D we perform α–β–expansion to its formula and the resulting subformulae as long as possible. This yields a collection $\{C_1, \ldots, C_k\}$ of sets of formula such that

$$D \equiv C_1 \vee \ldots \vee C_k$$

thereby justifying the term OR-node for D.

Let C be such a C_i. C is *downward closed*: $\alpha \in C$ implies $\alpha_1, \alpha_2 \in C$; $\beta \in C$ implies $\beta_1 \in C$ or $\beta_2 \in C$. It follows that C is equivalent to its subset of elementary formula which is of the general form

$$\{P, \ldots, \neg Q, AXg_1, \ldots, AXg_k, EXh_1, \ldots, EXh_\ell\}.$$

Hence, to create the successors of C, we let for $j \in [1 : \ell]$, $D_j = \{g_1, \ldots, g_k, h_j\}$. We have that

C is satisfiable iff D_1 is satisfiable and ... and D_k is satisfiable.

This justifies the use of the term AND-node for C.

After completing the construction of the initial tableau T_0, it is pruned by repeatedly applying the following deletion rules until the tableau stabilizes, yielding the final tableau T_1 which is conceivably empty.

1. Delete any node C which contains both a proposition P and its negation $\neg P$.
2. Delete any node C one of whose original successors D_i has been deleted, or which has no successors to begin with.
3. Delete any node D all of whose original successors have been deleted.
4. Delete any node C containing an eventuality which is not "fulfillable" within the current version of the tableau (as described in detail below).

We explain rule 4 in greater detail. An eventuality is a formula such as EFh, $E(gUh)$, AFh, or $A(gUh)$ which asserts that something does eventually happen. It is necessary to ensure that there is a certificate of its indeed happening present in the tableau. If, e.g., EFh is in C_0, then there should be a path in the tableau from C_0 to a node C with $h \in C$. This path defines a directed acyclic subgraph, DAG[C_0, EFh], rooted at C_0 certifying fulfillment of EFh starting from C_0. $E(gUh)$ is handled similarly. Fulfillment of AFh in C_0 means, roughly speaking, that there should be a subtree rooted at C in the tableau all of whose frontier nodes C contain h. More precisely, there should be a finite, directed acyclic graph, DAG[C_0, AFh], rooted at C_0 all of whose frontier nodes are AND-nodes C' containing h with the following stipulations on interior nodes: if interior OR-node D is in DAG[C_0,AFh] precisely one of its tableau successors C is also in DAG[C_0,AFh]; if interior AND-node C is in DAG[C_0, AFh], all of its tableau successors $D_1, \ldots, D_k$ are in DAG[C_0,AFh]. To handle $A(gUh)$, DAG[C_0,$A(gUh)$] is defined similarly.

The existence of such DAG's can be checked efficiently by an iterative process amounting to model checking, taking into account the OR-nodes intervening between AND-nodes.

Proposition 4.1. The original formula f_0 is satisfiable iff the final tableau T_1 contains a node C containing f_0.

Proof (Sketch). To facilitate the argument, we define for every AND-node C and every eventuality e, DAGG[C,e] to be DAG[C,e] with interior OR-nodes elided if $e \in C$, and, otherwise, to be some fixed, acyclic graph with exactly one C successor for each D successor of C. DAGG[C,e] is thus a potential "chunk" of a model certifying that (i) C has sufficient successors to satisfy its subformulae of the form EXh_j and (ii) e, if present in C is fulfilled by the time the frontier is reached.

Note that these DAGGs can be spliced together by identifying nodes in the frontier of one with the roots of others. Suppose this is done in an exhaustive and systematic way as shown in Figure 1. Here $S_1, \ldots, S_N$ are all the AND-nodes, i.e. the "states" of the tableau and $e_1, \ldots, e_m$ are all the eventualities. We have a matrix whose entries are all the DAGG$[S_i,e_j]$'s spliced together as shown.

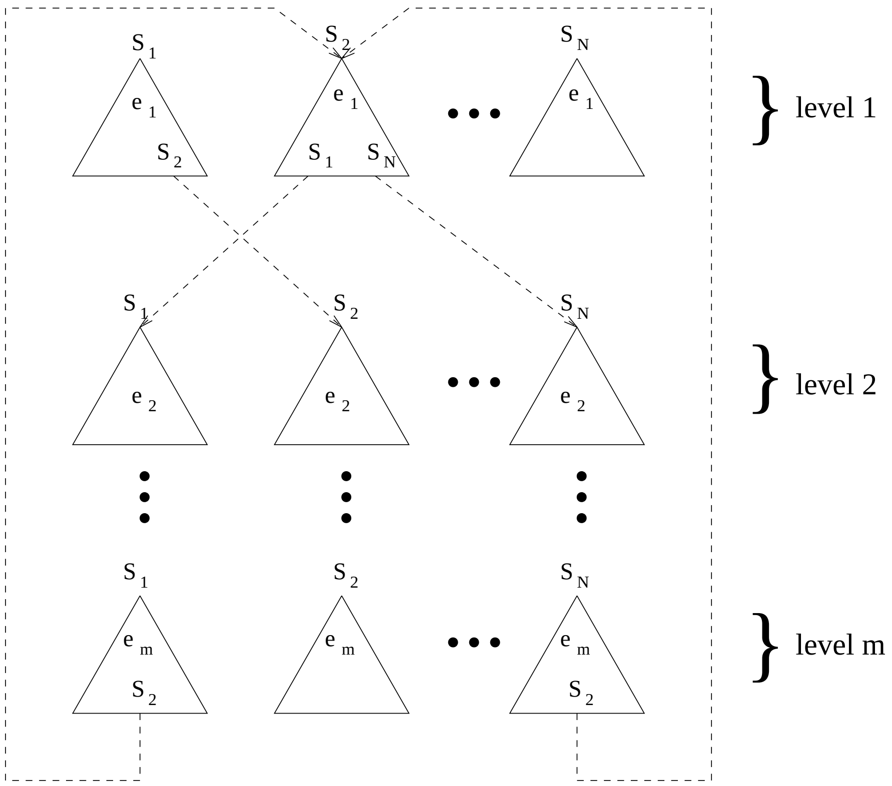

Fig. 1. Model formed from DAGGs

In the resulting graph, observe that each node is propositionally consistent, satisfies its EXh formulas, its AXg formulas, and its invariances such as AGh are also satisfied. The latter follows by downward closure and correctness of AX's: if AGh labels a node S, by downward closure, so do h and $AXAGh$. Hence all successors $S', \ldots, S''$ are labeled AGh and so on. In this way, invariances may be seen to take care of themselves. The only possible problem is that eventualities may not be fulfilled. But, if an eventuality, such as $e_j = AFh$, is not fulfilled along some path, the formula AFh is propagated along in the label of all the nodes of that path. Eventually, the path must hit (the root S of) some DAGG$[S,AFh]$ at level j and go through its frontier where h occurs thereby fulfilling the eventuality.

Thus we see, that in this systematically constructed graph, call it M_1, for any node C, we have $M_1, C \models \wedge C$. If there is a C containing f_0 then f_0 is satisfiable. This establishes "soundness" of the decision procedure.

Completeness of the algorithm may be established as follows. If f_0 is satisfiable then there is a model M and state s_0 such that $M, s_0 \models f_0$. Without loss of generality, we may assume M has been unwound into a tree-like model. Let M' be the *quotient structure* obtained from M by identifying all states of M that satisfy exactly the same AND-node label. It follows that M' defines a *pseudo-model* of f_0 that is contained within the tableau throughout its pruning. The essential point is that if an eventuality such as AFq holds at a state s in M there is a subtree of M rooted at s whose frontier nodes contain q. This subtree can be collapsed to yield DAG[C_s,AFq] in the tableau where C_s is the AND-node for s. Hence, C_s will never be eliminated on account of its eventuality AFq being unfulfilled. $\square$

The size of the tableau T_0 is exponential in $n = |f_0|$ since there are at most $O(n)$ different subformulas that can appear in AND-nodes and OR-nodes. The pruning procedure can be implemented to run in time polynomial in $|T_0|$, yielding an exponential time upper bound. A matching lower bound can be established by simulating alternating polynomial space Turing machines (cf. [FL79]) Thus, we have (cf. [EH85], [EC82])

Theorem 4.2. CTL satisfiability is deterministic exponential time complete.

5 Decision Procedures II: Automata-theoretic Approach

There has been a resurgence of interest in finite state automata on infinite objects, due to their close connection to temporal logic. 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 state automaton on infinite strings that accepts a string iff it defines a model of the formula p_0. The satisfiability problem for linear logics is thus reduced to the nonemptiness 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 has been a fruitful and active area of research.

5.1 Linear Time and Automata on Infinite Strings

We first review the basics of automata-theoretic approach for linear time. (See [Va9?] for a more comprehensive account.) The tableau construction for CTL can be specialized, essentially by dropping the path quantifiers to define a tableau construction for PLTL, remembering that in a linear structure, $Ep_0 \equiv Ap_0 \equiv p_0$. The extended closure of a PLTL formula p_0, $ecl(p_0)$, is defined to be $\{q, \neg q :$ q appears in some AND-node C $\}$. The (initial) tableau for p_0 can then be simplified to be a structure $T = (S, R, L)$ where S is the set of AND-nodes, i.e., states, $R \subseteq S \times S$ 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 $\Sigma = 2^{AP}$ that are models of p_0, by letting the arc (u, v) be labeled with $AtomicPropositions(v)$, i.e., the set of atomic propositions in v (cf. [ES83]). Technically, $\mathcal{A}$ is a tuple of the form $(Q, \Sigma, \delta, s_0, \Phi)$ where $Q = S \cup \{s_0\}$ is the state set, $s_0 \notin S$ 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 $AtomicPropositions(t)$ $= a\}$. The acceptance condition Φ is described below below. A *run* r of $\mathcal{A}$ on input $x = a_1 a_2 a_3 \ldots \in \Sigma^\omega$ is an infinite sequence of states $s_0 s_1 s_2 \ldots$ such that $\forall i \geq 0 \quad \delta(s_i, a_{i+1}) \supseteq \{s_{i+1}\}$. Note that $\forall i \geq 1$ $AtomicPropositions(s_i) = a_i$. The automaton $\mathcal{A}$ *accepts* input x iff there is a run r on x that satisfies the acceptance condition Φ.

Several different types of acceptance conditions Φ may be used. For *Muller* acceptance, we are given a family $\mathcal{F}$ of sets of states. Letting $In\ r$ denote the set of states in Q that appear infinitely often along r, we say that run r meets the Muller condition provided that $In\ r \in \mathcal{F}$.

For a *pairs* automaton (cf. [McN66], [Ra69]) acceptance is defined in terms of a finite list $((\text{RED}_1, \text{GREEN}_1), \ldots, (\text{RED}_k, \text{GREEN}_k))$ of pairs of sets of automaton states (which may be thought of as pairs of colored lights where $\mathcal{A}$ flashes the red light of the first pair upon entering any state of the set RED_1, etc.): r satisfies the pairs condition iff there exists a pair $i \in [1..k]$ such that RED_i flashes finitely often and GREEN_i flashes infinitely often. It is often convenient to assume the pairs acceptance condition is given formally by a temporal logic formula $\Phi = \bigvee_{i \in [1..k]} (\overset{\infty}{F}\ \text{GREEN}_i \wedge \neg \overset{\infty}{F}\ \text{RED}_i)$. Similarly, a *complemented pairs* (cf. [St81]) automaton has the negation of the pairs condition as its acceptance condition; i.e., for all pairs $i \in [1..k]$, infinitely often GREEN_i flashes implies that RED_i flashes infinitely often too. The complemented pairs acceptance condition can given formally by a temporal logic formula $\Phi = \bigwedge_{i \in [1:k]} \overset{\infty}{F} \text{GREEN}_i \Rightarrow \overset{\infty}{F} \text{RED}_i$. A special case of both pairs and complemented pair conditions is the *Buchi* [Bu62] acceptance condition. Here there is a single GREEN light and $\phi = \overset{\infty}{F}$GREEN.

A final acceptance condition that we mention is the *parity* acceptance condition [Mo84] (cf. [EJ91]). Here we are given a finite list $(C_1, \ldots, C_k)$ of sets of states which we think of as colored lights. The condition is that the highest index color C_i which flashes infinitely often should be of even parity.

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. If $ecl(p_0)$ has m eventualities $(p_1 U q_1), \ldots, (p_m U q_m)$, we let $\mathcal{A}$ have m pairs $(\text{RED}_i, \text{GREEN}_i)$ of lights. Each time a state containing $(p_i U q_i)$ 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 (pUq) is *pending* at state s of run r provided that $(pUq) \in s$ 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 $(pUq) \in ecl(p_0)$, (pUq) is pending almost everywhere along r) iff $\forall$ eventuality $(pUq) \in ecl(p_0)$, (pUq) 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, $(p_i U q_i)$ 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 in every state.) Now observe that there are infinitely many GREEN flashes iff $\forall i \in [1{:}m]$ $(p_i U q_i)$ 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 $exp(\mid p_0 \mid) \cdot \mathrm{O}(\mid p_0 \mid) = exp(\mid p_0 \mid)$ states.

5.2 Branching Time and Tree Automata

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]. However, it is for richer logics such as CTL* that the use of tree automata become essential.

Tree Automata

We describe finite automata on labeled, infinite binary trees (cf. [Ra69]).[8] The set $\{0,1\}^*$ may be viewed as an infinite binary tree, where the empty string λ is the root node and each node u has two successors: the 0-successor $u0$ and the 1-successor $u1$. A finite (infinite) *path* through the tree is a finite (resp., infinite) sequence $x = u_0, u_1, u_2, \ldots$ such that each node u_{i+1} is a successor of node u_i. If Σ is an alphabet of symbols, an infinite binary Σ-*tree* is a labeling L which maps $\{0,1\}^* \longrightarrow \Sigma$, i.e., a binary tree where each node is labeled with a symbol from Σ.

A *finite automaton* $\mathcal{A}$ on infinite binary Σ-trees is a tuple $(\Sigma, Q, \delta, q_0, \Phi)$ where

Σ is the finite, nonempty *input alphabet* for the input tree,
Q is the finite, nonempty set of *states* of the automaton,
$\delta : Q \times \Sigma \rightarrow 2^{Q \times Q}$ is the nondeterministic *transition function*,
$q_0 \in Q$ is the *start state* of the automaton, and
Φ is an *acceptance condition* as described previously.

A *run* of $\mathcal{A}$ on the input Σ-tree L is, intuitively, an annotation of the input tree with automaton states consistent with the transition function δ. Formally, a run is a function $\rho : \{0,1\}^* \rightarrow Q$ such that for all $v \in \{0,1\}^*$, $(\rho(v0), \rho(v1)) \in \delta(\rho(v), L(v))$ and $\rho(\lambda) = q_0$. We say that $\mathcal{A}$ *accepts* input tree L iff there exists a run ρ of $\mathcal{A}$ on L such that for all infinite paths x starting at the root of L if $r = \rho \circ x$ is the sequence of states $\mathcal{A}$ goes through along path x, then the acceptance condition Φ holds along r.

Tree Automata Running on Graphs

Note that an infinite binary tree L' may be viewed as a "binary" structure $M = (S, R, L)$ where $S = \{0,1\}^*$, $R = R_0 \cup R_1$ with $R_0 = \{(s, s0) : s \in S\}$ and $R_1 = \{(s, s1) : s \in S\}$, and $L = L'$. We could alternatively write $M = (S, R_0, R_1, L)$.

We can also define a notion of a tree automaton running on an appropriately labeled "binary" directed graphs, that are not trees. Such graphs, if accepted, serve as witnesses to the nonemptiness of tree automata. We make the following definitions.

[8] CTL^* and the other logics we study, have the property that their models can be unwound into an infinite tree. In particular, in [ESi84] it was shown that a CTL^* formula of length k is satisfiable iff it has an infinite tree model with finite branching bounded by k, i.e. iff it is satisfiable over a k-ary tree. Our exposition of tree automata can be easily generalized k-ary trees. We consider only binary trees to simplify the exposition, and for consistency with the classical theory of tree automata.

A *binary structure* $M = (S, R_0, R_1, L)$ consists of a state set S and labeling L as before, plus a transition relation $R_0 \cup R_1$ decomposed into two functions: $R_0 : S \longrightarrow S$, where $R_0(s)$ specifies the 0-successor of s, and $R_1 : S \longrightarrow S$, where $R_1(s)$ specifies the 1-successor of s.

A *run* of automaton $\mathcal{A}$ on binary structure $M = (S, R_0, R_1, L)$ starting at $s_0 \in S$ is a mapping $\rho : S \rightarrow Q$ such that $\forall s \in S$, $(\rho(R_0(s)), \rho(R_1(s))) \in \delta(\rho(s), L(s))$ and $\rho(s_0) = q_0$. Intuitively, a run is a labeling of M with states of $\mathcal{A}$ consistent with the local structure of $\mathcal{A}$'s "transition diagram".

The Transition Diagram of a Tree Automaton

The transition function δ of a tree automaton $\mathcal{A}$ as above can be viewed in a natural way as defining a transition diagram T, which facilitates the development of algorithms for testing nonemptiness. The transition diagram T of $\mathcal{A}$ is a bipartite AND/OR-graph where the set Q of states of $\mathcal{A}$ comprises the set of OR-nodes, while the AND-nodes define the allowable moves of the automaton. Intuitively, OR-nodes indicate that a nondeterministic choice has to be made (depending on the input label), while the AND-nodes force the automaton along all directions. For example, suppose that for automaton $\mathcal{A}$, $\delta(s, a) = \{(t_1,\ u_1), \ldots,\ (t_m, u_m)\}$ and $\delta(s, b) = \{(v_1,\ w_1),\ \ldots,\ (v_n,\ w_n)\}$ then the transition diagram contains the portion shown in Figure 2.

Remark: The resemblance of the transition diagram of a tree automaton to a CTL tableau is striking. In fact, they are really the same except that in CTL the models are built out of AND-nodes, while for tree automata the models are built out of OR-nodes as we shall see below. In fact, an alternative, equivalent formulation of tree automata [GH82] has the transition function information presented by a left transition function $\delta_0 : Q \times \Sigma \longrightarrow 2^Q$ giving the possible left successor states together with an independent right transition function $\delta_1 : Q \times \Sigma \longrightarrow 2^Q$ giving the possible right successor states. For this dual formulation of tree automaton the transition diagram is just as for CTL with models built out of OR-nodes.

One Symbol Alphabets

For purposes of testing nonemptiness, without loss of generality, we can restrict our attention to tree automata over a single letter alphabet, and, thereby, subsequently ignore the input alphabet. Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, \Phi)$ be a tree automaton over input alphabet Σ. Let $\mathcal{A}' = (Q, \Sigma', \delta', q_0, \Phi)$ be the tree automaton over one letter input alphabet $\Sigma' = \{c\}$ obtained from $\mathcal{A}$ by, intuitively, taking the same transition diagram but now making all transitions on symbol c. Formally, $\mathcal{A}'$ is identical to $\mathcal{A}$ except that the input alphabet is Σ' and the transition function δ' is defined by $\delta'(q, c) = \bigcup_{a \in \Sigma} \delta(q, a)$. Then $\mathcal{A}$ is nonempty iff $\mathcal{A}'$ is nonempty.

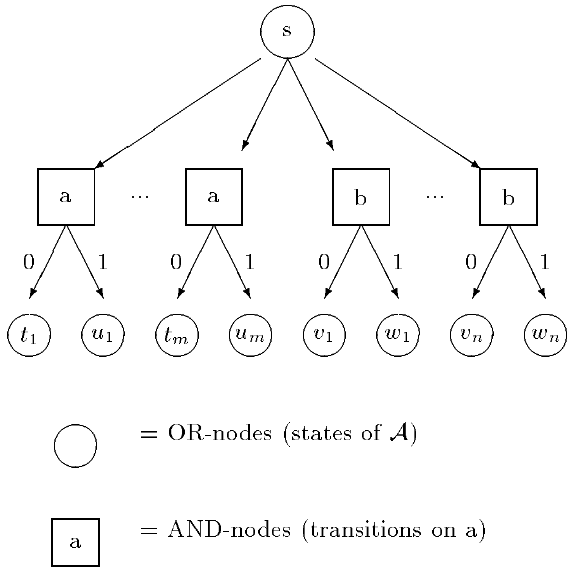

Fig. 2. Portion of tree automaton transition diagram

Henceforth, we therefore assume that we are dealing with tree automata over a one symbol alphabet.

Generation and Containment

It is helpful to reconsider the notion of run to take advantage of the organization of the transition diagram of an automaton.

Intuitively, there is a run of tree automaton $\mathcal{A}$ on binary structure M starting at s_0 in M provided M is *generated* from $\mathcal{A}$ by unwinding $\mathcal{A}$'s transition diagram so that (a) s_0 in M corresponds to q_0 in $\mathcal{A}$, and (b) each state of M is a copy of an OR-node of $\mathcal{A}$, and, moreover, each state's successors are consistent with the transition diagram. We say that a binary structure M is *contained in* $\mathcal{A}$ provided M is generated by unwinding $\mathcal{A}$ starting starting at $s_0 \in M$ and $q_0 \in \mathcal{A}$ and, moreover, (a copy of) M is a subgraph of T.

Linear Size Model Theorem

The following theorem is from [Em85] (cf. [VS85]). Its significance is that it provides the basis for our method of testing nonemptiness of pairs automata;

it shows the existence of a small binary structure accepted by the automaton contained in its transition diagram, provided the automaton is nonempty.

Theorem 5.1 (Linear Size Model Theorem). Let $\mathcal{A}$ be a tree automaton over a one symbol alphabet with pairs acceptance condition $\Phi = \bigvee_{i\in[1:k]}(\overset{\infty}{F}\text{GREEN}_i \wedge \overset{\infty}{G}\neg\text{RED}_i)$.
Then automaton $\mathcal{A}$ accepts some tree T iff $\mathcal{A}$ accepts some binary model M, of size linear in the size of $\mathcal{A}$, which is a structure contained in the transition diagram of $\mathcal{A}$.

Proof. ($\Rightarrow$) For Φ a pairs condition, [HR72] shows that, if $\mathcal{A}$ accepts some tree M_0, then it accepts some finite binary model M_1 starting at some state $s_0 \in M_1$. We will explain, roughly, how M_1 is obtained in the case of Φ involving a single pair. Along every path x of M_0 there is a first node u with these properties: (i) u is labeled with automaton state q, (ii) there is a first node v along x strictly before u also labeled with automaton state q, and (iii) the set of automaton states in the interval along x from v to u coincides with the set of states that appear infinitely often along x. The set of all such nodes u forms the frontier of a finite subtree of M_0. M_1 is formed from this finite subtree by identifying each u with its v (i.e., redirecting the predecessor of each u into its v). M_1 is a structure generated by $\mathcal{A}$. In the case of one pair, it follows that $M_1, s_0 \models A\Phi$.[9]

Given any such finite structure M_1 of $A\Phi$ generated by $\mathcal{A}$ we can obtain a (necessarily finite) structure M contained in $\mathcal{A}$. If two distinct nodes s and t of M_1 have the same labeling with states of $\mathcal{A}$, then we can eliminate one of them as follows. Attempt to delete s by redirecting all its predecessors u to have t as a successor instead. More precisely, delete all edges of the form (u, s) and replace them by edges of the form (u, t). If the resulting structure, call it M^t, is a model of $A\Phi$, we have reduced the number of "duplicates", such as s and t by one. If not, try replacing t by s instead. If M^s, the resulting structure is a model of $A\Phi$ we are done.

However, if both these replacements fail to yield a model of $A\Phi$, each must yield a model of $E\neg\Phi$, i.e. each introduces a bad cycle. In M^t the bad cycle is of the form (where u is a predecessor of s in M_1) $u \to t \to \ldots \to u$, where except for the first transition (u, t) the suffix path from t to u is in the original M_1. In M^s the bad cycle is of the form (where v is a predecessor of t in M_1) $v \to s \to \ldots \to v$, where except for the first transition (v, s) the suffix path from s to v is in the original M_1.

But, these two suffix paths in M_1 together with the edges (u, s), and (v, t) in M_1 form a bad cycle in M_1: $u \to s \to \ldots \to v \to t \to \ldots \to u$. This contradicts that M_1 was a model of $A\Phi$.

[9] The technical fine point is that cycles satisfying the one pair condition are closed under union. This is not so for multiple pairs. However, a slightly more subtle construction can be used to get M_1 in the case of multiple pairs (cf. [HR72]).

By repeatedly eliminating duplicates in this way we eventually get the desired model M contained in $\mathcal{A}$.

($\Leftarrow$) Any model M contained in $\mathcal{A}$ such that $M, s_0 \models A\Phi$ can plainly be unwound into a tree that is accepted by $\mathcal{A}$. □

We can now use the Linear Size Model Theorem to establish the following result (cf. [EJ88]).

Theorem 5.2. The problem of testing nonemptiness of pairs tree automata is NP-complete.

proof sketch: Membership in NP (cf. [Em85], [VS85]): Given a pairs tree automaton, if it accepts some tree, then a linear size modelexists contained within its transition diagram. Guess that model. Use the (efficient FairCTL) model checking algorithm of [EL87] to verify in deterministic polynomial time that $A(\vee_i(\overset{\infty}{F}\text{GREEN}_i \wedge \overset{\infty}{G}\neg\text{RED}_i))$ holds.

NP-hardness is established by reduction from 3-SAT (cf. [EJ88]). □

Our overall approach to testing nonemptiness is formulated in terms of "pseudo-model checking". We write $\mathcal{A}, q_0 || - f$ to indicate that $\mathcal{A}$ is a pseudo-model of f at q_0; the technical definition is that there is a binary model M contained in $\mathcal{A}$ such that $M, q_0 \models f$.

Observe that for pairs automaton $\mathcal{A}$ with acceptance condition Φ, $\mathcal{A}$ is nonempty iff $\mathcal{A}, q_0 || - A\Phi$.

For simplicity, we illustrate how pseudo-model checking can be performed in a very rudimentary case. Suppose that $\Phi = F\text{GREEN}$, so that we are dealing with a finitary acceptance condition where along each path the GREEN light should flash (at least) once.

We wish to pseudo- model check $\mathcal{A}, q_0 || - AF\text{GREEN}$. We will reduce this here to ordinary model checking by fixpoint calculation. We start with the fixpoint characterization $AF\text{GREEN} \equiv \mu Z.\text{GREEN} \vee AXZ$. But notice that we cannot directly use this fixpoint characterization, on account of the presence of both OR-nodes and intervening AND-nodes in the diagram of $\mathcal{A}$. Instead, we use $\mu Y.\text{GREEN} \vee EXAXY$. The inserted "$EX$" skips over the intervening AND-nodes appropriately. Now, $\mathcal{A}, q_0 || - AF\text{GREEN}$ iff $\mathcal{A}, q_0 \models \mu Y.\text{GREEN} \vee EXAXY$.

This simple idea can be generalized considerably to handle pairs acceptance (cf [EJ88]). Let Φ be the pairs condition $\vee_{i \in I}(\neg\overset{\infty}{F}\text{RED}_i \wedge \overset{\infty}{F}\text{GREEN}_i)$ where $I = [1{:}n]$ is the index set for the pairs and I_{-i} denotes $I \setminus \{i\}$. We pseudo-model check $A\Phi$ based on a fixpoint characterization in terms of simple formulae AF and AG and by induction on the number of pairs. $A\Phi = \mu Y.\tau(Y)$ where

$$\tau(Y) = \bigvee_{i \in I} AFAG((\neg\text{RED}_i \vee Y) \wedge A(F\text{GREEN}_i \vee \Phi_{-i}))$$

The pseudo-model checking algorithm of the transition diagram of $\mathcal{A}$ must simultaneously search for a graph contained within $\mathcal{A}$ and check that it defines a model of the pairs condition $A\Phi$. It successively calculates the set of states where $\tau^i(false)$ is "satisfiable" in the transition diagram using Tarski-Knaster approximation. The effective size of the above fixpoint characterization is exponential in the number of pairs. The pseudo-model checking algorithm runs in time proportional to the size of the fixpoint characterization and polynomial in the size of the transition diagram. It is shown in [EJ88] that this yields a complexity of $(mn)^{O(n)}$ for an automaton with transition diagram of size m and n pairs. This bound is thus polynomial in the number of automaton states, with the degree of the polynomial proportional to the number of pairs. This polynomial complexity in the size of the state diagram turns out to be significant in applications to testing satisfiability as explained below.

Related results on the complexity of testing nonemptiness of tree automata may be found in [EJ88], [PR89], [EJ91].

Decision Procedure for CTL*

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 that accepts some tree iff the formula is satisfiable. 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 delicate automata-theoretic results such as McNaughton's construction ([McN66]) for determinizing automata on infinite strings, or subsequent improvements [ES83], [Sa88], [EJ89].

The original CTL* formula f_0 can be converted, by the introduction of auxiliary propositions, into a *normal form* f_1 that is satisfiable iff the original formula is, but where path quantifiers are nested to depth at most 2. For example,

$$\begin{gathered} EFAFEGP \approx \\ EFAFQ_1 \wedge AG(Q_1 \equiv EGP) \approx \\ EFQ_2 \wedge AG(Q_2 \equiv AFQ_1) \wedge AG(Q_1 \equiv EGP) \end{gathered}$$

Here $g \approx h$ means that g is satisfiable iff h is satisfiable.

By using propositional reasoning, we can further the simplify our task of testing satisfiability of f_1 to testing satisfiability of boolean combinations of subformulae of the form Ap_0, Ep_0, and $AGEp_0$ where p_0 is a pure linear time formula. More precisely, we build tree automata for each of these three forms. A composite product automaton can then be readily obtained for f_1 and tested for satisfiability.

It turns out that it is easy to build tree automata for Ep_0 and $AGEp_0$. Rather surprisingly, the crucial and difficult step is building the tree automaton the branching time modalities of the form Ap_0, in terms of the ω–string automaton for the corresponding linear time formula p.

We explain the difficulty that manifests itself with just the simple modality Ap_0. The naive approach to get a tree automaton for Ap_0 would be to simply build the ω–string automaton for p_0 and then run it it down all paths of the input tree. However, while this seems very natural, it does *not*, in fact, work. To see this, consider two infinite paths xy and xz in the input tree which start off with the same common finite prefix x but eventually separate to follow two different infinite suffixes y or z. It is possible that p_0 holds along both paths xy and xz, but in order for the nondeterministic automaton to accept, it might have to "guess" while reading a particular symbol of the finite prefix x whether it will eventually read the suffix y or the suffix z. The state the string automaton 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.

Of course, if the string automaton is *deterministic* the above difficulty vanishes. We should therefore ensure that the string automaton for p_0 is determinized before constructing the tree automaton. The drawback is that determinization is an expensive operation. However, it appears to be unavoidable.

For a linear temporal logic formula p_0 of length n we can construct an equivalent Buchi nondeterministic finite state automaton on ω–strings of size $exp(n)$. We can then get tree automata for Ep_0 and $AGEp_0$ of size $exp(n)$. However, for Ap_0, use of classical automata-theoretic results yields a tree 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_0 plus the double exponential cost of McNaughton's construction to determinize it. For a CTL* formula of length n, nonemptiness of the composite tree automaton can be tested in exponential time to give a decision procedure of deterministic time complexity quadruple exponential in n.

An improvement in the determinization process makes an exponential improvement possible. In [ES83] it was shown that, due to the special structure of the string automata derived from linear temporal logic formulae, such string automata could be determinized 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 a double exponential number of states and a single exponential number of pairs. [10] As described in the previous section, 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].

For the Mu-calculus (and PDL–Δ (cf. [St81])) similar techniques techniques can be applied to get a single exponential decision procedure (cf. [EJ88], [SE84]).

Other Comments on Automata

There are distinct advantages to thinking automata-theoretically. First, for obtaining decision procedures, automata-theoretic techniques provide the only known methods of obtaining elementary time decision procedures for some very expressive logics such as CTL* and the Mu-calculus. The techniques are general and uniform. For example, the techniques above can be combined to yield a single exponential decision procedure for the Mu-calculus [EJ88]. This was a problem which again was not obviously in elementary, much less exponential time (cf. [KP83], [SE84]). Secondly, automata can provide a general, uniform framework encompassing essentially all aspects temporal reasoning about reactive systems (cf. [VW84]], [VW86], [Va87], [AKS83], [Ku94]). Automata themselves have been proposed as a potentially useful specification language. Automata, moreover, bear an obvious relation to temporal structures, the state transition graphs of concurrent programs, etc. This makes it possible to account for various types of temporal reasoning applications such as program synthesis [PR89] and model checking in a conceptually uniform fashion [VW86]. Verification systems based on automata have also been developed (cf. [Ku86]).

Thus, temporal reasoning has benefited from automata. But the converse holds as well, with much work on automata inspired by and/or using ideas from temporal and modal logics. The more improved nonemptiness algorithm for pairs tree automata discussed above uses the notion of transition diagram for a tree automaton and exploits the relationship to a CTL tableau. New types of automata on infinite objects have also been proposed to facilitate reasoning in temporal logic (cf. [St81], [VS85], [MP87a], [EL87]). Exponential improvements (down to single exponential) in the complexity of determinization of ω-string automata appeared in a special case in [ES83] and in the general case in [Sa88]. Exponential improvements (again down to single exponential) in the cost of complementation of such automata have appeared in [SVW87] (cf. [EJ89]). Not only

[10] Some intuition for this phenomenon can be gained by considering the case of CTL. In a naive formulation, a tree automaton for a CTL formula would use one complemented pair of lights for each eventuality to check its fulfillment. The number of lights is thus linear in the formula length while the tableau/transition diagram size is exponential in the formula length.

do determinization and complementation of string automata have fundamental applications to decision procedures for temporal and modal logics, but they are basic problems to the theory of automata on infinite objects in their own right.

6 Expressiveness versus Complexity

Two of the most important characteristics of a temporal or modal logic intended for reasoning about programs are (i) its *expressiveness*, referring to which correctness properties it can and cannot express; and (ii) its *complexity*, referring to the computational cost of performing various types of mechanical reasoning operations such as model checking and satisfiability/validity testing in the logic.

Expressiveness can be characterized in several ways. Theoretical results can be established to show that, e.g., logic L_1 is strictly subsumed in expressive power by logic L_2. Accordingly, let us write $L_1 < L_2$ to mean that every property expressible by a formula of L_1 is also expressible by a formula of L_2 but that there is at least one property expressible by a formula of L_2 that is not expressible by *any* formula of L_1. We analogously write $L_1 \equiv L_2$ and $L_1 \leq L_2$. It is then possible to establish various hierarchies and other relations of expressive power among logics. Thus, it is possible to characterize expressiveness of one logic relative to another. Another possibility is to take a particular logic or formalism as a reference standard providing, in an sense, an absolute yardstick against which other logics are compared. We will see some examples of both relative and absolute expressiveness subsequently.

In practice, we may be more concerned with what specific correctness properties can and cannot be expressed within a formalism. While in general the more expressive the better, it is often sufficient to just be able to express invariance AGp, leads-to/temporal implication $AG(p \Rightarrow AFq)$, and a few other simple properties. A significant practical issue is how *conveniently* can we capture a desired property.

A related theoretical issue concerns *succinctness* of specifications. This refers to the economy of descriptive power of a formalism: how long a formula is required to a capture a property? Two formalisms can be of equivalent raw expressive power, yet differ radically in succinctness. For example, PLTL and FOLLO (the First Order Language of Linear Order)[11] are equivalent in expressive power, but FOLLO can be *nonelementarily* more succinct, meaning that the translation of FOLLO into PLTL may cause a blowup in length not bounded by any fixed composition of exponential functions. We may ask if this means that FOLLO is better suited than PLTL for specification and reasoning about reactive programs? Probably not. Even though FOLLO can be astronomically more succinct,

[11] FOLLO is essentially a formalization of the right hand side of the definition of the temporal operators; e.g., FP corresponds to $\exists t(t > 0 \wedge P(t))$.

it is quite possible that it is *too* succinct. Certainly, it is known that the complexity of mechanical reasoning in it would be nonelementary. PLTL, on the other hand, seems to provide a good combination of expressive power, succinctness, and complexity of mechanical reasoning (as does CTL in the branching time framework). [12]

6.1 Tradeoffs

In general, the goals of work in this area are (a) to formulate the most expressive logic possible with the lowest complexity decision problem relevant to the application at hand; and (b) to understand the tradeoffs between complexity and expressiveness. In connection with point (b), it is worth noting that there is some relationship between the syntactic complexity of temporal logic formula and the computational complexity of their decision procedures. This appears related to the size and structure of the automaton (that would be) constructed for the formula. However, the relationship is somewhat intricate.

6.2 Expressiveness Hierarchy

The hierarchy shown below illustrates some of the key points regarding expressiveness that we would like to emphasize.

$$\text{CTL} \equiv \text{CTL}^+ < \text{CTF} < \text{CTL}^* < \text{PDL-}\Delta < \text{L}\mu \equiv \text{Tree Automata}$$

The first thing to note is that (finite state, pairs) tree automata coincide in expressive power with the Mu-calculus. Since virtually all mechanical reasoning operations can be performed in terms of tree automata and all branching time logics can, it turns out, be translated into tree automata, it is reasonable to take tree automata as a reference standard for branching time expressibility. [13]

[12] Actually, rather little research effort has gone into work on succinctness. Particularly valuable topics might include: identification of tractable and useful fragments of FOLLO (or equivalently S1S), use of ω-regular expressions as a specification language, and general efforts to gain a deeper understanding of the relation between syntactic complexity of a formula and the cost of mechanical reasoning w.r.t. the formula.

[13] In this connection, there is one minor technical caveat about comparing apples and oranges in the context of expressiveness: tree automata as originally defined run on infinite binary trees and can distinguish "left" from "right". In contrast, logics such as CTL* (or the the Mu-calculus with AX, EX as opposed to $<0>$, $[0]$, $<1>$, $[1]$) are interpreted over models of arbitrary arity but cannot distinguish "left" from "right". There are a variety of ways to formulate a uniform, compatible framework permitting

We next note that the logic PDL-Δ is strictly subsumed in expressive power by the Mu-calculus. PDL-Δ is the Propositional Dynamic Logic with infinite repetition operator; in essence, it permits assertions whose basic modalities are of the form $E\alpha$ where α is an ω-regular expression (cf. [St81]). PDL-Δ can be translated into the Mu-calculus essentially because ω-regular expressions can be translated into the "linear time" Mu-calculus (cf. [EL86]). For example, $P^*Q \equiv \mu Z.Q \vee (P \wedge XZ)$ and $P^\omega \equiv \nu Y.P \wedge XZ$. Similarly, $EP^*Q \equiv \mu Z.Q \vee (P \wedge EXZ)$ and $EP^\omega \equiv \nu Y.P \wedge EXZ$. The general translation can be conducted along these lines. It can be shown, however, that $\nu Y. <1> Y \wedge <2> Y$ is not expressible in PDL-Δ, over ternary structures with directions/arc labels 0, 1, 2 (cf. [Niw84]).

We also have that CTL* is strictly subsumed in expressive power by PDL–Δ. CTL* can be translated into PDL–Δ through use of the following main idea: each linear time formula h defines, using the tableau construction, an equivalent Buchi automaton (cf. [ES83]) which can be translated into an equivalent ω–regular expression α. Thus, the basic modality Eh of CTL* maps to $E\alpha$ in PDL–Δ. Because ω–regular expressions are strictly more expressive than PLTL (cf. [MP71], [Wo83]), there are properties expressible in PDL-Δ that cannot be captured by any CTL* formula. $E(P; \mathit{true})^\omega$ is perhaps the classic example of such a property.

It is worth noting that CTL* syntax can be described in a sort of shorthand: $\mathrm{B}(F, G, X, U, \wedge, \neg, \circ)$. This means that the basic modalities of CTL* are of the form A or E (for a Branching time logic) followed by a pure linear time formula built up from the linear time operators F, G, X, U, the boolean connectives $\wedge, \neg$, with nestings/compositions allowed as indicated by $\circ$. Then we have the expressiveness results below (cf. [EH86]).

We next compare CTL* with CTF, which is the precursor to CTL and CTL*, going back to [EC80]. CTF may be described as the logic $\mathrm{B}(F, G, X, U, \overset{\infty}{F}, \wedge, \neg)$. Plainly, any CTF formula is a CTL* formula. The difference, syntactically, is that CTF does not permit arbitrary nesting of linear time formulas in its basic modalities, although it does permit the special infinitary operator(s) $\overset{\infty}{F}$ (and in effect its dual $\overset{\infty}{G}$) to support reasoning about fairness. However, the CTL* basic modality $A(F(P \wedge XP))$ is not a CTF formula and, moreover, can be shown to be inequivalent to any CTF formula. Thus, CTL* is strictly more expressive than CTF.

The logic CTL$^+$ is given as $\mathrm{B}(F, G, X, U, \wedge, \neg)$, permitting basic modalities with linear time components that are boolean combinations of the linear time operators F, G, X, U. Thus, CTL$^+$ is a sublanguage of CTF, omitting the infini-

meaningful comparisons. One way is to compare "symmetric" tree automata, which do not distinguish left from right, interpreted over binary trees with branching time logics. See also the amorphous tree automata of [BG93]. The Mu-calculus would be equivalent to tree automata in any such reasonable common framework.

tary operator $\overset{\infty}{F}$ and its dual $\overset{\infty}{G}$. It can be shown that, for example, $E\overset{\infty}{F}P$ is not expressible in CTL$^+$. Thus, CTL$^+$ is strictly subsumed in expressive power by CTF.

The last logic shown, CTL, is B(F, G, X, U), whose basic modalities permit just a single linear temporal operator F, G, X, U following A or E. Plainly CTL is a sublanguage of CTL$^+$. Conversely, every formula of CTL$^+$ can be translated into an equivalent CTL formula. The essence of the translation is that the CTL$^+$ formula $E(FP_1 \wedge FP_2)$ is equivalent to the CTL formula $EF(P_1 \wedge EFP_2) \vee EF(P_2 \wedge EFP_1)$.

Finally, we comment again that comparisons of raw expressive power do not necessarily hold in like kind for succinctness. For example, even though CTL is equivalent in raw expressive power to CTL$^+$, the translation of CTL$^+$ into CTL can be effected with an exponential blowup. The CTL$^+$ formula $E(FP_1 \wedge FP_2 \wedge \ldots \wedge FP_n)$ is translated into the following equivalent but long CTL formula:

$$\bigvee_{i_1, i_2, \ldots, i_n \text{ is a permutation of } [1:n]} EF(P_{i_1} \wedge EF(P_{i_2} \wedge \ldots \wedge EFP_{i_n}))$$

Furthermore, while CTL* is less expressive than PDL–Δ it can be exponentially more succinct.

6.3 Complexity Summary

The table below summarizes key complexity results for automated temporal reasoning. The left column indicates the logic under consideration, the associated entry in the middle column characterizes the complexity of the logic's model checking problem, while the associated right column entry describes the complexity of the satisfiability problem. Each row describes a particular logic: PLTL, CTL, CTL*, and the Mu-calculus.

Logic	Model Checking	Satisfiability Testing
PLTL	PSPACE-complete $O(\lvert M\rvert \cdot exp(\lvert p_0\rvert))$ time	PSPACE-complete
CTL	P-complete $O(\lvert M\rvert \cdot \lvert p_0\rvert)$ time	EXPTIME-complete
CTL*	PSPACE-complete $O(\lvert M\rvert \cdot exp(\lvert p_0\rvert))$ time	2EXPTIME-complete
L$\mu_{(k)}$	NP $\cap$ co-NP $(\lvert M\rvert \cdot \lvert p_0\rvert)^{O(k)}$ time	EXPTIME-complete

The first row deals with PLTL, whose complexity was first analyzed in [SC85]. (cf. [Va9?]). PLTL model checking can be polynomially transformed to PLTL satisfiability testing. The essential point is that the "structure" of a structure M can be described by a PLTL formula where the nexttime operator and extra propositions are used to characterize what states are present what are the successors of each state.

The satisfiability problem of PLTL is PSPACE-complete. In practice, this bound amounts to a decision procedure of complexity $exp(n)$ for an input formula h of length n. The decision procedure is a specialization of that for CTL: build the exponential sized tableau for the formula, which may be viewed as a Buchi nfa on infinite strings and tested for nonemptiness in time polynomial in the size of of the automaton. It is possible,in fact, to build the automaton on-the-fly, keeping track of only an individual node and a successor node at any given time, guessing an accepting path in nondeterministic polynomial space.[14] This serves to show membership in PSPACE for satisfiability testing of PLTL and for model checking of PLTL by virtue of the above-mentioned reduction. By a generic reduction from PSPACE-bounded Turing machines, PLTL model checking can be shown to be PSPACE-hard; it then follows that PLTL satisfiability testing is also PSPACE-hard.

An important multi-parameter analysis of PLTL model checking was performed by Lichtenstein and Pnueli [LP85], yielding a bound of $O(|M| \cdot exp(|h|))$ for an input structure M and input formula h. The associated algorithm is simple and elegant. We wish to check whether there is a path starting at a given state s_0 in M satisfying the PLTL formula h. (We clarify why we have formulated the PLTL model checking problem in just this way below.) To do that, first build the tableau $\mathcal{T}$ for h. Then form essentially the product graph $M \times \mathcal{T}$, view it as a tableau, and test it for satisfiability. This amounts to looking for a path through the product graph whose projection onto the second coordinate defines a model of h that, by virtue of the projection onto its first coordinate, must also be a path in M. Vardi and Wolper [VW86] made the important recognition that this construction could be described still more cleanly and uniformly in purely automata-theoretic terms. Use $\mathcal{T}$ to define an associated Buchi nfa $\mathcal{A}$. Then define a Buchi automaton $\mathcal{B}$ that is the product of M and $\mathcal{A}$, and simply test $\mathcal{B}$ for nonemptiness.

Along with the above algorithm and its complexity analysis, the following "Lichtenstein–Pnueli thesis" was formulated: despite the potentially daunting exponential growth of the complexity of PLTL model checking in the size of the specification h formula, it is the linear complexity in the size of the input structure M which matters most for applications, since specifications tend to

[14] See the very interesting work by Barringer et. al [BFGGO89] on executable temporal logics extending this idea.

be quite short while structures tend to be very large. Thus, the argument goes, the exponential growth is tolerable for small specifications, and we are fortunate that the cost grows linearly in the structure size. Our main point of concern thus should be simply the structure size.

There appears to be a good deal of empirical, anecdotal evidence that the Lichtenstein-Pnueli thesis is often valid in actual applications. As further noted in a forthcoming section, very simple assertions expressible in a fragment of CTL are often useful. On the other hand, it is also possible to find instances where the Lichtenstein-Pnueli thesis is less applicable.

We remark that we have formulated the PLTL model checking problem to test, in effect, $M, s_0 \models Eh$. However, in applications using the linear time framework, we want to know whether *all* computations of a program satisfy a specification h'. This amounts to checking $M, s_0 \models Ah'$. It is, of course, enough to check $M, s_0 \not\models E(\neg h')$ which the Lichtenstein–Pnueli formulation handles. Since PLTL is trivially closed under complementation we thus have a workable, efficient solution to the "all paths" problem in terms of of the Lichtenstein–Pnueli formulation. (cf. [EL87]).

The next row concerns CTL. CTL model checking is P-complete. Membership in P was established in [CE81] by a simple algorithm based on the Tarski-Knaster theorem. This was improved to the bound $O(|M||f|)$ for input structure M and CTL formula f in [CES86]. Satisfiability testing for CTL is complete for deterministic exponential time [EH85]. The upper bound established using the tableau method was discussed previously. The lower bound follows by a generic reduction from alternating polynomial space Tm's (cf. [FL79]).

We next consider CTL*. Its model checking problem is of the same complexity as for PLTL. It is PSPACE-complete with a multi-parameter bound of $O(|M| \cdot exp(|f|))$. The lower bound follows because the PLTL model checking problem is a special case of the CTL* model checking problem. The upper bound follows because, as noted above, $Ah \equiv \neg E \neg h$, and by using recursive descent to handle boolean connectives and nested path quantifiers. In particular, to check the formula $E(FAGP \wedge GAFQ)$, first check AGP and label all states where it holds with auxiliary proposition P' ; next check AFQ and label all states where it holds with auxiliary proposition Q' ; finally, check $E(FP' \wedge GQ')$. Of course, in practice it is not really necessary to introduce the auxiliary propositions. It is simply enough to observe that subformulas AGP and AFQ are state formulas that can be used to first label the states where they are found to hold before evaluating the top level formula. CTL* satisfiability can be tested in deterministic double exponential time by building a tree automaton of essentially double exponential size and then testing it for nonemptiness as discussed previously. The lower bound follows by a generic reduction from alternating exponential space Tm's [VS85].

Finally, we come to the complexity of the Mu-calculus, Lμ. There are a number of interesting open questions concerning model checking in the Mu-calculus. First, a "tight" bound is not known. We do know that it is in NP $\cap$ co-NP [EJS93]. There are not too many such problems known. It suggests that we ought to be able to show that it is in P, but it has so far resisted all efforts to do so by a number of researchers. On the other hand, were it, say, shown to be NP-complete then we would get that NP = co-NP, which seems highly unlikely.

The argument establishing membership in NP follows from observing that, given a structure M and a Mu-calculus formula f, a nondeterministic Tm can guess a "threaded" annotation of the states and edges in M with "ranked" subformulas. The ranks indicate how many times a μ-formula can be unwound and the threads indicate whether subformula f at state s generates subformula g at a successor state t. It is then only necessary to check that this annotation is propositionally consistent at each state and well-founded. Membership in co-NP follows because the Mu-calculus is trivially closed under negation.

However, we can say more about useful fragments of the Mu-calculus. Recall the notion of *alternation depth* of a Mu-calculus formula referring to the depth of "significant" nestings of alternating least and greatest fixpoints. We use Lμ_k to indicate the Mu-calculus fragment where formulas are restricted to alternation depth k. Using the Tarski-Knaster theorem and basic monotonicity considerations, a time bound of $O((|M| \cdot |f|)^{k+1})$ can be shown (cf. [EL86]). Subsequent work has addressed improving the degree of the polynomial to simply k (cf. [An93], [CS93]). In fact, this can be improved to about $O((|M| \cdot |f|)^{k/2})$ (cf. [Lo+94]) by a technique that trades time for space and stores, roughly, all intermediate results. However, this method also uses space about $O((|M| \cdot |f|)^{k/2})$. In contrast, the straightforward algorithm only uses about $O(|M| \cdot |f|)$ space. Of course, all known "practical" properties including all those expressible in PDL-Δ are in Lμ_2 and can be model checked in low order polynomial time. Finally, satisfiability testing for the Mu-calculus is in deterministic exponential time [EJ88], as shown by building a tree automaton essentially equivalent to the Mu-calculus formula and testing it for nonemptiness. The lower bound of being deterministic exponential time hard again follows by simulating alternating polynomial space Tm's.[15]

6.4 Automaton Ineffable Properties

While finite state automata on infinite trees seem a good reference standard for logics of the sort we have been considering, it is worth noting that there are some types of reasonably properties correctness properties which are not expressible by any finite state tree automaton. One such property we refer to as "uniform inevitability".

[15] or by succinctly encoding CTL.

The property of (ordinary) inevitability of P is expressed by the CTL formula AFP. This is of the general form $\forall$ computation $\exists$ time P; note that the time i a which P occurs along a given computation path depends on the specific path.

The property of *uniform inevitability* is of the general form $\exists$ time $\forall$ computation P. That is, there is a single, uniform time i such that along all computation paths P holds at exactly time i.

Interestingly, it can be shown that uniform inevitability is not definable by any finite state automaton on infinite trees [Em87]. The argument is akin to that used to establish the pumping lemma for ordinary context free languages. However, uniform inevitability is definable by a type of pushdown tree automata.

6.5 Mu-calculus is Equivalent to Tree Automata

The Mu-calculus is equivalent in expressive power to tree automata. This result was first given in [EJ91]. We will discuss that argument and the information that can be extracted from it.

But first we note that the result can be rather easily obtained by using translation through SnS concatenated with other known results. In [KP83] it was established that the Mu-calculus can be translated into SnS: $\mathrm{L}\mu \leq \mathrm{S}n\mathrm{S}$. Earlier, [Ra69] established the basic result that tree automata are equivalent to SnS: $\mathrm{ta} \equiv \mathrm{S}n\mathrm{S}$. Later, [Niw88] showed that for a restricted Mu-calculus, call it $\mathrm{R}\mu$, we have $\mathrm{ta} \equiv \mathrm{R}\mu$. By definition, $\mathrm{R}\mu \leq \mathrm{L}\mu$. Putting it all together,

$$\mathrm{L}\mu \leq \mathrm{S}n\mathrm{S} \equiv \mathrm{ta} \equiv \mathrm{R}\mu \leq \mathrm{L}\mu$$

We conclude that $\mathrm{L}\mu \equiv \mathrm{ta}$.

The limitation of the above argument is that many appeals to the *Complementation Lemma* are made in the proof of the step $\mathrm{S}n\mathrm{S} \equiv \mathrm{ta}$ in [Ra69]. The Complementation Lemma asserts that if tree language L is accepted by a finite state pairs automaton on infinite trees, the complement language $\overline{L}$ is also accepted by some finite state pairs tree automaton on infinite trees. The original proof of Rabin [Ra69] was extraordinarily difficult and intricate, and the Lemma remains one of the technically most challenging results in the field. Because of its technically formidable nature as well as its importance to applications involving decidability of logical theories[16], a number of authors have attempted to give simpler proofs of the Complementation Lemma (cf. [Ra71], [GH82], [Mu84]).

Arguments were given in [EJ91] (a) to prove directly that $\mathrm{ta} \equiv \mathrm{L}\mu$; and showing how to use that to (b) give a simplified proof of the Complementation Lemma by translation through the Mu-calculus. The equivalence (a) is established by showing

$$\mathrm{L}\mu \leq \mathrm{ata} \leq \mathrm{ta} \equiv \mathrm{R}\mu \leq \mathrm{L}\mu$$

[16] We emphasize that CTL* and $\mathrm{L}\mu$ satisfiability seem to require string automaton determinization or tree automaton complementation.

where ata denotes the class of alternating tree automata (which are defined technically below). All of these translations from left-to-right are direct. None involves the Complementation Lemma. We can then accomplish the simplified proof (b) of the Complementation Lemma based on the following simple idea: given a a tree automaton $\mathcal{A}$ there is an equivalent Lμ formula f. The negation of f, $\neg f$, is certainly also a formula of Lμ since Lμ is trivially closed under syntactic negation. Therefore, $\neg f$ can then be translated into an equivalent tree automaton $\overline{\mathcal{A}}$ which recognizes precisely the complement of the set of trees recognized by $\mathcal{A}$.

Remark: The restricted Mu-calculus, Rμ, of [Niw88] consists of formulas $f, g, \ldots$ built up from constructs of these forms: atomic proposition constants and their negations $P, Q, \neg P, \neg Q \ldots$, atomic proposition variables $Y, Z, \ldots$, restricted conjunctions of the form $P \wedge EX_0 Y \wedge EX_1 Z$, disjunctions $f \vee g$, and least and greatest fixpoint operators $\mu Y.f(Y)$, $\nu Y.f(Y)$. Since it is not syntactically closed under complementation, nor obviously semantically closed (as the general $\wedge$ is missing)[17], we cannot use it directly to establish the Complementation Lemma.

Lμ into ata

The idea underlying the translation of Lμ into ata is simple: a Mu-calculus formula *is* an alternating tree automaton. In more detail, the syntax diagram of a Mu-calculus formula may be viewed as (a particular formulation of) the transition diagram of an alternating tree automaton that checks the local structure of the input tree to ensure that it has the organization required by the formula. As the alternating automaton runs down the input tree, "threads" from the syntax diagram are unwound down each path; in general, there may be multiple threads going down the same path of the tree due to conjunctions in the formula. We remark that it is these conjunctions and associated multiple threads which make the automaton an alternating one.

For example, the syntax diagram of $\mu Y.P \vee (<0> Y \wedge <1> Y)$ is shown in Figure 3. As indicated, there is a node in the syntax diagram for each subformula and an edge from each formula to its immediate subformulae. In addition, there is an edge from each occurrence of Y back to μY.

The transition diagram consists of AND-nodes and OR-nodes. All nodes are OR-nodes except those corresponding to connective $\wedge$.[18] Each node has an input symbol, usually an implicit ε indicating, as usual, that no input is consumed. The automaton starts in state μY, from which it does an ε move into state $\vee$. In state $\vee$ it makes a nondeterministic choice. The automaton may enter state

[17] It is semantically closed but the proof requires an appeal to the Complementation Lemma.

[18] Matters are simplified by noting that over a binary tree 0,1 are functions, not just relations.

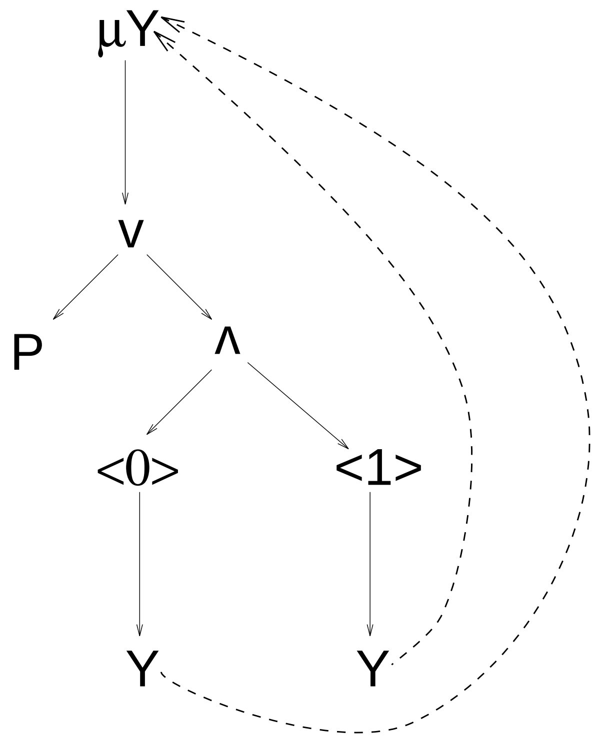

Fig. 3. Syntax diagram of $\mu Y.P \vee (<0>\ Y \wedge <1>\ Y)$

P. In this case it checks the input symbol labeling the current node to see if it matches P, in which case it accepts; otherwise, it rejects. Alternatively, the automaton may enter state $\wedge$, which is an AND-node and from which it will exercise universal nondeterminism. From $\wedge$ the automaton launches two new threads of control: $<0>$ down the left branch and $<1>$ down the right branch. Then state $<0>$ does an ε move to ensure that at the left successor node the automaton is in state Y from which does an ε move into μY. Similarly, state $<1>$ does an ε move to ensure that at the right successor node the automaton is in state Y and then μY. Etc.

Acceptance is handled by colored lights placed to ensure that μ–regeneration[19] is well-founded. One way to do this is to associate a pair of lights ($\textsc{red}_i$, $\textsc{green}_i$) with each eventuality $\mu Y_i.f$. Whenever the eventuality $\mu Y_i.f$ is regenerated along along a thread, indicated in the syntax diagram by traversing the edge re-entering the node for μY_i from within the scope of μY_i, flash $\textsc{green}_i$. Whenever the scope of μY_i is exited, flash $\textsc{red}_i$. Thus, $\mu Y_i.f$ is regenerated infinitely often along a

[19] Informally, this refers to recursive unfolding of μ- formulae.

thread iff $\overset{\infty}{F}$ GREEN$_i$ $\wedge \neg\overset{\infty}{F}RED_i$ holds along it. Call a thread meeting the associated pairs condition *bad*. Then the Mu-calculus formula is true of the input tree iff the associated ata accepts the tree by virtue of the existence of a run on the tree such that all threads of all paths of the run are *good* (i.e., not bad). This amounts to acceptance by the complemented pairs condition $\wedge_i(\overset{\infty}{F}\text{GREEN}_i \Rightarrow \overset{\infty}{F}\text{RED}_i)$

Nondeterminization

Next, we must discuss the "nondeterminization" of an alternating tree automaton needed to show ata $\leq$ ta. First, however, we make precise the notion of such an alternating tree automaton. An ata $\mathcal{A}$ is just like an ordinary, nondeterministic ta except that it has both existential and universal nondeterminism. This is reflected in its transition function which permits transitions of the form $\delta(q,a)$ $= \{(\{r_1^1,\ldots,r_1^{\ell_1}\},\{s_1^1,\ldots,s_1^{m_1}\}),\ldots,(\{r_k^1,\ldots,r_k^{\ell_k}\},\{s_k^1,\ldots,s_k^{m_k}\})\}$. The meaning is that when automaton $\mathcal{A}$ is in state q at current node v of the input tree which is labeled with symbol a, then $\mathcal{A}$ first uses existential nondeterminism to choose among k alternative next moves. Each possible existential next move is specified by a pair of the form $(\{r_i^1,\ldots,r_i^{\ell_i}\},\{s_i^1,\ldots,s_i^{m_i}\})$, with $1 \leq i \leq k$. Then the automaton exercises universal nondeterminism to go into *all* of the states $r_i^1,\ldots,r_i^{\ell_i}$ on the left successor node $v0$ and into *all* of the states $s_i^1,\ldots,s_i^{m_i}$ on the right successor node $v1$.

As usual, the automaton $\mathcal{A}$ starts out in a designated start state q_0 on the root node λ of the input tree. By applying the above transitions a *run* of $\mathcal{A}$ on the input tree is constructed. This run is not just a labeling of the nodes of the input tree with automaton states, but a superposition of a tree of threads (sequences of automaton states) on the input tree. If q is the tip of a thread z a node v, which is labeled by symbol a, and $\delta(q,a)$ is of the form above, then z branches into extending threads of the form $zr_i^1,\ldots,zr_i^{\ell_i}$ corresponding to the left successor $v0$, and $zs_i^1,\ldots,zs_i^{m_i}$, corresponding to the right successor $v1$. Any infinite tree comprised of threads in this way constitutes a run of $\mathcal{A}$ on the input tree. The automaton $\mathcal{A}$ accepts the input tree provided there exists a run on the input tree such that along every path every thread meets the acceptance condition of the automaton (Buchi, pairs, etc.).

Note that a run ρ of $\mathcal{A}$ on an input tree may be viewed as a tree superimposed on the input tree. Thee may be two or more copies of the same automaton state q as the tip of two different finite threads corresponding to a particular tree node. These copies of q may make different existential moves. One therefore cannot merge such different copies of q as the "threads" that would result would in general be different from those in the original run.

However, for a *history free* alternating tree automaton, if it has an accepting run, it has an accepting run in which the existential choices depend only on the current state and position in the tree (and not the thread of which it is a tip). Thus, such a *history free run* will not necessarily be a tree superimposed on the input tree, but a dag, with intuitively threads intertwined down each tree branch, such that there is only a single copy of each automaton state at each tree node. Along a path through the input tree, the coSafra construction (cf. [EJ89], [Sa92]) can be used to bundle together the collection of infinitely many threads along the path, which are choked through a finite number of states at each node.

Finally, it turns out that the alternating tree automaton corresponding to a Mu-calculus formula is history free. The intuitive justification is that at each existential choice node, one can take the choice of least rank, ensuring that μ's are fulfilled as soon as possible.

Remark: An alternative method of nondeterminization is to essentially construct the tree automaton for testing satisfiability of a Mu-calculus formula as in [SE84]. A sharpening of this construction builds a tree automaton equivalent to the Mu-calculus formula. Basically, perform a tableau construction to get a local automaton checking the invariance properties inherent in the formula. Conjoin it with a global automaton that checks well-foundedness of μ–regenerations. The global automaton is obtained as follows: Build an ω–string automaton that guesses a bad thread through the tableau along which some μ–formula is regenerated infinitely often, in violation of the requirement that it should be well-founded. Then use the coSafra construction to simultaneously determinize and complement that string automaton. The tree automaton that runs the resulting string automaton down all paths of the tree is the desired global automaton that checks the liveness properties associated with μ–formulas.

6.6 Restricted Temporal Logics

If one considers longstanding previous trends in the work on decision procedures for (propositional) temporal and modal logics of programs, one observes the following characteristics:

- There has been a continual emphasis toward logics of increasingly greater expressive power.
- Most work has sought the most expressive temporal logic decidable in single exponential time deterministically; this is a consequence of the fact that the temporal logic subsumes ordinary propositional logic and the reasonable presumption that P $\neq$ NP, strongly suggesting that we can not do better in general.
- A quest for increasing generality and richness of of expressive power.

We now discuss a possibly promising counter-trend (cf. [ESS89]) toward more limited logics with these characteristics:

- The limited logics exhibit greater specificity; they are tailored for particular applications.
- The limited logics are of restricted expressive power. The restrictions may limit both raw expressive power and economy of descriptive power.
- They are intended to support efficient, polynomial time decision procedures.

We will focus on the restricted logics from [ESS89] and [EES90]. It can be quite delicate to obtain a logic that is restricted, efficiently decidable, and at the same time useful. Some of the implications of these requirements are:

- We must give up propositional logic in its full generality, since obviously any logic subsuming propositional logic must be at least NP-hard.
- The atomic propositions should be disjoint and exhaustive. Otherwise, if we allow overlapping propositions, there can be as many as 2^n subsets of n propositions, yielding an immediate combinatorial explosion. (In practice, this restriction may not be that onerous in our applications. For example, we may wish to describe a program which may be in any of n locations. This may be described using propositions $at\text{-}loc_1,...,at\text{-}loc_n$.
- The overall syntax should be simplified. One simplification is to restrict formulas to be of the form: $\bigwedge$ *assertion*; that is, to be a conjunction of simpler assertions. Note that for purposes of program specification $\bigwedge$ is more fundamental than $\bigvee$. One typically wants a program that meets a conjunction of criteria. Another simplification is to limit the depth of nesting of temporal operators. Deeply nested temporal modalities are rarely used in practice anyway.

Simplified CTL

We first consider Simplified CTL (SCTL). It turns out that SCTL corresponds precisely to the fragment of CTL actually used in program synthesis in [EC82].

The formulae of SCTL are conjunctions of assertions of the following forms:

- $P \vee \ldots \vee P'$ –initial assertions
- $AG(Q \vee \ldots Q')$ – invariance assertions
- $AG(P \Rightarrow AF(R \vee \ldots \vee R))$ – leads-to assertions
- $AG(P \Rightarrow A((Q \vee \ldots \vee Q')U_s(R \vee \ldots R'))$ – assurance assertions
- $AG(P \Rightarrow AX(Q \vee \ldots \vee Q') \wedge$ – successor assertions
  $EX(R \vee \ldots \vee R') \wedge \ldots \wedge EX(S \vee \ldots \vee S'))$

over an alphabet of disjoint, exhaustive propositions $P, P', Q, Q', R, R', \ldots$, etc., subject to the following *Euclidean Syntactic Constraint (ESC)*:

> If an SCTL formula has conjuncts of the form $AG(P \Rightarrow AFQ)$ and $AG(P \Rightarrow AX(\ldots R \vee \ldots) \ldots)$ then $AG(R \Rightarrow AFQ)$ must also be a conjunct of the formula.

The ESC ensures that eventualities are recoverable from propositions alone.

The significance in practice of the ESC is that, while it is a restriction, it permits the specification of "history-free" processes. This means that the eventualities pending at each state S of a process are the same irrespective of the path taken by the process from its initial state to S.

The restricted syntax of SCTL permits the decision procedure of CTL to be simplified yielding a polynomial time decision procedure for formulas of SCTL. To understand in broad terms why this is possible in it helpful to think automata-theoretically [VW84]. The automaton/tableau for a CTL formula may be thought of as the product of the local automaton (the nexttime tableau) with the global automaton, which is itself the product of an eventuality automaton for each eventuality such as AFP. The size of the whole automaton is thus the size of the the nexttime tableau times the product of the sizes of each eventuality automaton. An eventuality automaton for, say, AFP has 2 states, one for AFP being pending, one for it it being fulfilled (or not required to be satisfied). Thus, the size of the whole automaton is exponential in the number of eventualities. However, if the set of pending eventualities can be determined from set of atomic propositions, as is the case with SCTL owing to the ESC, then the local automaton can serve as the entire automaton.

The SCTL decision procedure then amounts to:

- Construct the nexttime tableau for the input formula f_0 using the nexttime assertions. Each atomic proposition P is an AND-node of the tableau. Each such AND-node P gets sets of successor AND-nodes, intermediated by OR-nodes, based on the nexttime assertion[20] associated with P. For example, $AG(P \Rightarrow AX(Q_1 \vee Q_2 \vee R_1 \vee R_2) \wedge EXQ_1 \wedge EX(R_1 \vee R_2))$ would have two OR-node successors, one with AND-node Q_1 as a successor, the other with AND-nodes R_1, R_2 as successors. This is the local automaton $\mathcal{A}_{f_0}$. By virtue of the ESC it is also the entire automaton. The initial assertions determine the "start state".
- Check $\mathcal{A}_{f_0}$ for "nonemptiness". Repeatedly delete "bad" nodes from its transition diagrams. There is associated with every node a set eventualities e that must be fulfillable. The key step is to ensure that each such e is fulfillable by finding the appropriate DAG[Q,e]'s as for ordinary CTL.

Example: Consider the SCTL formula comprised of the conjunction of the following assertions:

[20] Without loss of generality we may assume there is only one.

$$(P \vee R)$$

$$AG(P \vee R \vee S \vee T)$$

$$AG(P \Rightarrow AXS \wedge EXS)$$
$$AG(R \Rightarrow AX(R \vee T) \wedge EXT)$$
$$AG(S \Rightarrow AX(P \vee R) \wedge EXP \wedge EXR)$$
$$AG(T \Rightarrow AX(P \vee T) \wedge EX(P \vee T)$$

$$AG(P \Rightarrow AFT)$$
$$AG(S \Rightarrow AFT)$$

We get the initial tableau shown in Figure 4 (i). Since AFT is not fulfillable at node P, delete P. Propagate the deletion to any incident edges as well as OR-nodes whose only successors were P.

The resulting tableau is shown in Figure 4 (ii). Node S violates its successor assertion because it no longer has a P successor. Thus, S is deleted and its now spurious successor OR-node.

In Figure 4 (iii) the final pruned tableau is shown. It induces the model shown in Figure 4 (iv).

Restricted Linear Temporal Logic

We now consider another logic that turns out to be efficiently decidable: Restricted Linear Temporal Logic (RLTL). It is presented, for the sake of uniformity, in CTL-like syntax. But the path quantifiers are all A's; hence, any satisfiable formula has a linear model, and it may be viewed as a linear time logic. Alternatively, the path quantifiers may all be elided to get formulae that are literally within the syntax of PLTL.

Formulae are now conjunctions of assertions of the following forms:

- $AG(Q \vee \ldots Q')$ – invariance assertions
- $AG(P \Rightarrow AF(R \vee \ldots \vee R'))$ – leads-to assertions
- $AG(P \Rightarrow AX(Q \vee \ldots \vee Q'))$ – successor assertions

where again propositions are disjoint and exhaustive. There is no Euclidean Syntactic Constraint or related restriction.

Nonetheless, we can establish the following

Theorem Satisfiability of RLTL can be tested in deterministic polynomial time.

Proof idea. The basic idea is simple: Build the nexttime tableau. The input formula is satisfiable iff the tableau has a total, self-fulfilling strongly connected

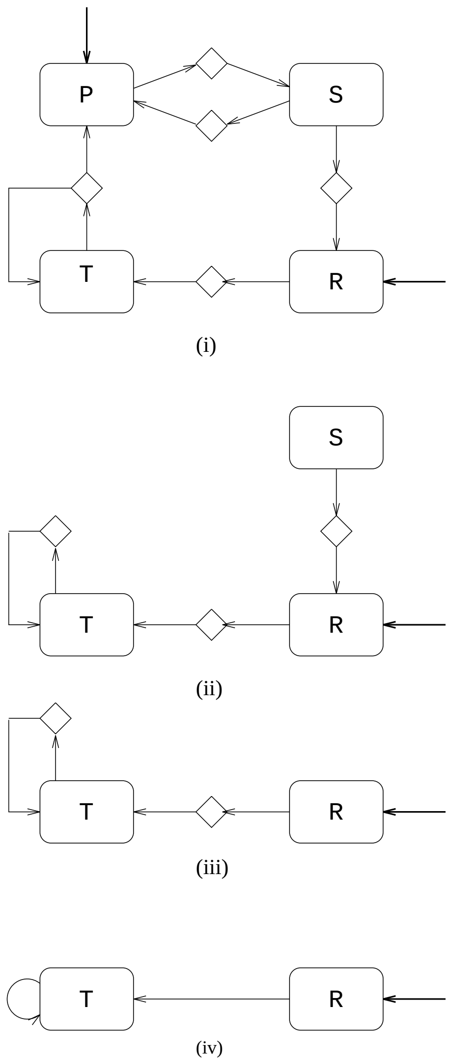

Fig. 4. SCTL decision procedure example

subgraph C. The latter means that if any $AF(Q \vee \ldots \vee Q')$ appears in C, then one of $Q, \ldots, Q'$ also appears in C. Thus it suffices to build the initial tableau, split into SCC's, delete non-self-fulfilling SCC's, until stabilization.

Example: Consider the RLTL formula comprised of the conjunction of the following assertions:

$$AG(P \vee Q \vee R \vee S \vee T)$$

$$AG(P \Rightarrow AXQ)$$
$$AG(R \Rightarrow AXP)$$
$$AG(T \Rightarrow AX(P \vee R))$$
$$AG(Q \Rightarrow AX(R \vee S))$$
$$AG(S \Rightarrow AX(R \vee S))$$

$$AG(Q \Rightarrow AFP)$$
$$AG(S \Rightarrow AFT)$$

The initial nexttime tableau is shown in Figure 5 (a). It is partitioned into SCC's: $\{\{T\}, \{P, Q, R, S\}\}$. The SCC $\{T\}$ is deleted because it is not total. The node S is deleted from SCC $\{P, Q, R, S\}$. because AFh is not fulfillable. See Figure 5 (b) Then the remainder of this SCC is split into SCC's yielding $\{\{P, Q, R\}\}$. The sole remaining SCC of this collection, $\{P, Q, R\}$, is total and self-fulfilling. Hence, any infinite path through it defines a model of the original specification as in Figure 5 (c).

Restricted Initialized Linear Temporal Logic.

We now illustrate that the boundary between efficient decidability and worst case intractability can be quite delicate. Certainly, RLTL is highly restricted. While useful specifications can be formulated in RLTL, many properties cannot be expressed. One very basic type of assertion that was omitted was an *initial assertion*. Let us define Restricted Initialized Linear temporal logic (RILTL) to be the logic permitting formulas which are conjunctions of assertions of the form:

- $P \vee \ldots \vee P'$ – initial assertions
- $AG(Q \vee \ldots Q')$ – invariance assertions
- $AG(P \Rightarrow AF(R \vee \ldots \vee R'))$ – leads-to assertions
- $AG(P \Rightarrow AX(Q \vee \ldots \vee Q'))$ – successor assertions

where one again propositions are disjoint and exhaustive, and there is no Euclidean Syntactic Constraint or related restriction. In other words, RILTL equals RLTL plus an initial assertion.

Most surprisingly, this small change increases the complexity:

Theorem The satisfiability problem for RILTL is NP-hard.

The idea behind the proof is as follows. Given a state graph M, such as that shown in Figure 6, we can capture its structure by a formula f_M that is a simple

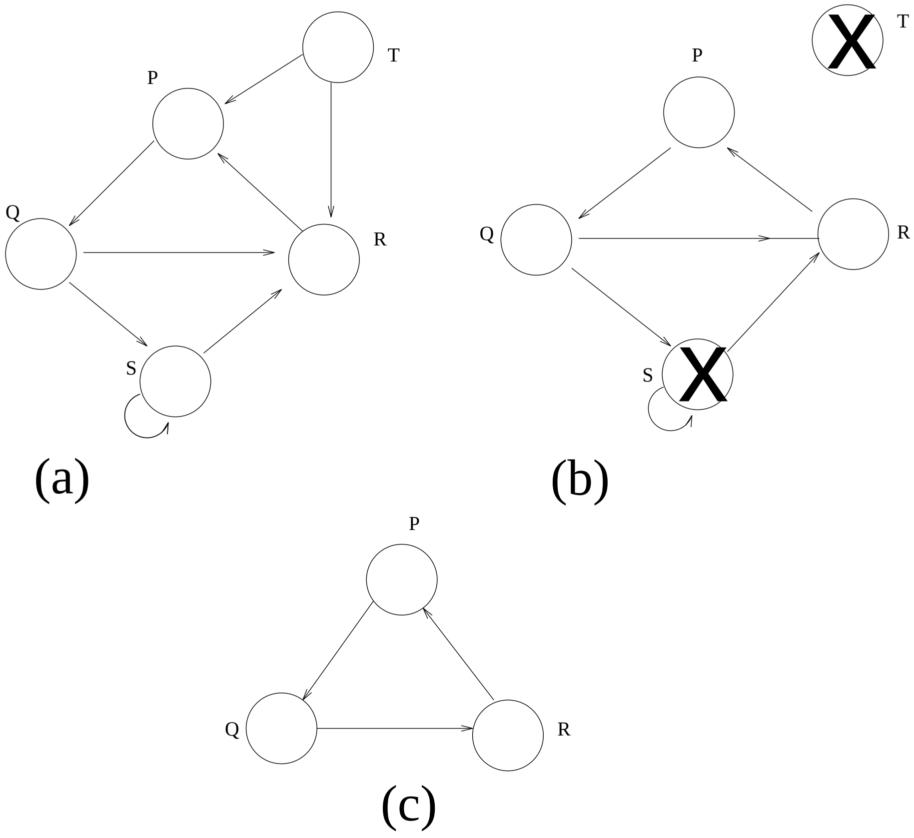

Fig. 5. Decision procedure for RLTL

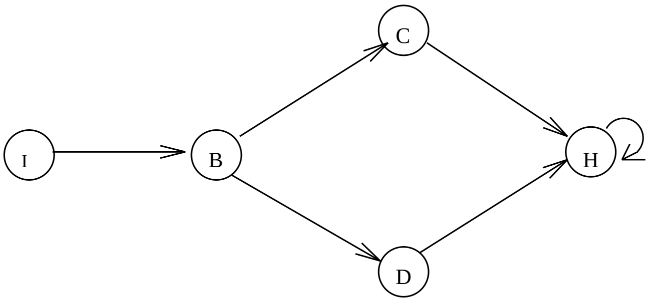

Fig. 6. Model checking can be reduced to RILTL satisfiability

conjunction of RILTL successor assertions, such as

$$\begin{gathered} AG(I \Rightarrow AXB) \wedge \\ AG(B \Rightarrow AX(C \vee D)) \wedge \\ AG(C \Rightarrow AXH) \wedge \\ AG(D \Rightarrow AXH) \wedge \\ AG(H \Rightarrow AXH) \end{gathered}$$

(Here we assume, for ease of exposition, that a unique proposition labels each node.) We thereby reduce model checking over M to RILTL satisfiability: $M, I \models E(FB \wedge FC \wedge FH)$ iff $f_M \wedge I \wedge AG(I \Rightarrow AFB) \wedge AG(I \Rightarrow AFC) \wedge AG(I \Rightarrow AFH)$ is satisfiable. We can generalize this argument to reduce arbitrary model checking problems of the form $M, s_0 \models E(FP_1 \wedge \ldots FP_n)$ to RILTL satisfiability. This restricted form of the PLTL model checking problem is known to be NP-hard as established by Sistla and Clarke [SC85]. (The intuition is that it is necessary to consider all possible permutations that the P_i can occur in.)

7 Conclusion

Due to the proliferation of computer microelectronics and associated safety critical software, there is an undeniable and growing need to find effective methods of constructing correct reactive systems. One factor these systems have in common beyond their nondeterministic, ongoing, reactive nature is that they are complex. While it is conceivable that it is easy to describe in general terms what such a system is supposed to do (e.g., provide an air traffic control system), it appears quite difficult to "get the details straight". Temporal logic appears to to provide a good handle on precisely stating just what behavior is to occur when at a variety of levels of detail. Automation of temporal reasoning appears to offer a good handle on actually keeping track of the myriad of associated points of fine detail.

Acknowledgements This work was supported in part by NSF grant CCR-9415496 and by SRC contract 95-DP-388.

We also thank Howard Barringer, Colin Stirling, and Moshe Vardi for most helpful comments on a preliminary version of this paper.

References

[An93] Anderson, H. R., Verification of Temporal Properties of Concurrent Systems, Ph. D. Dissertation, Computer Science Department, Aarhus Univ., Denmark, June 1993.

[AKS83] Aggarwal S., Kurshan R. P., Sabnani K. K., "A Calculus for Protocol Specification and Validation", in Protocol Specification, Testing and Verification III, H. Ruden, C. West (ed.'s), North-Holland 1983, 19-34.

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

[BFGGO89] Barringer, H., Fisher, M., Gabbay, D., Gough, G., and Owens, R., Metatem: A Framework for Programming in Temporal Logic. In Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness, Mook, The Netherlands, Springer LNCS, no. 430, June 1989.

[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.

[BPM83] Ben-Ari, M., Pnueli, A. and Manna, Z. The Temporal Logic of Branching Time. Acta Informatica vol. 20, pp. 207-226, 1983.

[BG93] Bernholtz, O., and Grumberg G., Branching Time Temporal Logic and Amorphous Automata, Proc. 4th Conf. on Concurrency Theory, Hildesheim, Springer LNCS no. 715, pp. 262–277, August 1993.

[BS92] Bradfield, J., and Stirling, C., "Local Model Checking for Infinite State Spaces", *Theor. Comp. Sci.*, vol. 96, pp. 157-174, 1992.

[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

[Br86] Bryant, R., Graph-based algorithms for boolean function manipulation, IEEE Trans. on Computers, C-35(8), 1986.

[Bu62] Buchi, J. R., On a Decision Method in restricted Second Order Arithmetic, Proc. 1960 Inter. Congress on Logic, Methodology, and Philosophy of Science, pp. 1–11.

[CE81] Clarke, E. M., and Emerson, E. A., Design and Verification of Synchronization Skeletons using Branching Time Temporal Logic, Logics of Programs Workshop, IBM Yorktown Heights, New York, Springer LNCS no. 131., pp. 52-71, May 1981.

[CES86] 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.

[CFJ93] Clarke, E. M., Filkorn, T., Jha, S. Exploiting Symmetry in Temporal Logic Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993.

[CGB88] Clarke, E. M., Grumberg, O., and Brown, M., Characterizing Kripke Structures in Temporal Logic, Theor. Comp. Sci., 1988

[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.

[CGB89] Clarke, E. M., Grumberg, O., and Brown, M., Reasoning about Many Identical Processes, Inform. and Comp., 1989

[CS93] Cleaveland, R. and Steffan, B., A Linear-Time Model-Checking Algorithm for the Alternation-Free Modal Mu-calculus, Formal Methods in System Design, vol. 2, no. 2, pp. 121-148, April 1993.

[Cl93] Cleaveland, R., Analyzing Concurrent Systems using the Concurrency Workbench, Functional Programming, Concurrency, Simulation, and Automated Reasoning Springer LNCS no. 693, pp. 129-144, 1993.

[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.

[CM90] Coudert, O., and Madre, J. C., Verifying Temporal Properties of Sequential Machines without building their State Diagrams, Computer Aided Verification '90, E. M. Clarke and R. P. Kurshan, eds., DIMACS, Series, pp. 75-84, June 1990.

[DGG93] Dams, D., Grumberg, O., and Gerth, R., Generation of Reduced Models for checking fragments of CTL, CAV93, Springer LNCS no. 697, 1993.

[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, IEEE Proc. 133, Pt. E 5, pp. 276-282, 1986.

[Em81] Emerson, E. A., Branching Time Temporal Logics and the Design of Correct Concurrent Programs, Ph. D. Dissertation, Division of Applied Sciences, Harvard University, August 1981.

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

[Em85] E.A. Emerson, "Automata, Tableaux, and Temporal Logics", *Proc. Workshop on Logics of Programs*, Brooklyn College, pp. 79–87, Springer LNCS no. 193, June 1985.

[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.

[EES90] Emerson, E. A., Evangelist, M., and Srinivasan, J., On the Limits of Efficient Temporal Satisfiability, Proc. of the 5th Annual IEEE Symp. on Logic in Computer Science, Philadelphia, pp. 464-477, June 1990.

[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.

[EJ88] Emerson, E. A. and Jutla, C. S., "Complexity of Tree Automata and Modal Logics of Programs", *Proc. 29th IEEE Foundations of Computer Sci., 1988*

[EJ89] Emerson, E. A., and Jutla, C. S., On Simultaneously Determinizing and Complementing ω-automata, Proceedings of the 4th IEEE Symp. on Logic in Computer Science (LICS), pp. 333–342, 1989.

[EJ91] Emerson, E. A., and Jutla, C. S. "Tree Automata, Mu-Calculus, and Determinacy", *Proc. 33rd IEEE Symp. on Found. of Comp Sci.*, 1991

[EJS93] Emerson, E. A., Jutla, C. S., and Sistla, A. P., On Model Checking for Fragments of the Mu-calculus, Proc. of 5th Inter. Conf. on Computer Aided Verification, Elounda, Greece, Springer LNCS no. 697, pp. 385-396, 1993.

[EL86] Emerson, E. A., and Lei, C.-L., Efficient Model Checking in Fragments of the Mu-calculus, IEEE Symp. on Logic in Computer Science (LICS), Cambridge, Mass., June, 1986.

[EL87] Emerson, E. A., and Lei, C.-L.m 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.

[ES93] Emerson, E. A., and Sistla, A. P., Symmetry and Model Checking, 5th International Conference on Computer Aided Verification, Crete, Greece, June 1993full version to appear in *Formal Methods in System Design*.

[ES83] Emerson, E. A., and Sistla, A. P., Deciding Full Branching Time Logic, Proc. of the Workshop on Logics of Programs, Carnegie-Mellon University, Springer LNCS no. 164, pp. 176–192, June 6–8, 1983; journal version appears in *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.

[Em87] Emerson, E. A., Uniform Inevitability is Finite Automaton Ineffable, Information Processing Letters, v. 24, pp. 77-79, 30 January 1987.

[Em90] Emerson, E. A., Temporal and Modal Logic, in Handbook of Theoretical Computer Science, vol. B, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[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, New York, 1986

[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.

[GS92] German, S. M. and Sistla, A. P. Reasoning about Systems with many Processes, Journal of the ACM, July 1992, Vol 39, No 3, pp 675-735.

[GH82] Gurevich, Y., and Harrington, L., "Trees, Automata, and Games", *14th ACM STOC*, 1982.

[HT87] Hafer, T., and Thomas, W., Computation Tree Logic CTL* and Path Quantifiers in the Monadic Theory of the Binary Tree, ICALP87.
Knowledge and Common Knowledge in a Distributed Environment, Proc. 3rd ACM Symp. PODC, pp. 50-61.

[HS86] 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., Probabilistic Temporal Logics for Finite and Bounded Models, 16th STOC, pp. 1-13, 1984.

[HS84] Hart, S. and Sharir, M. Probabilistic 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.

[HR72] 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.

[ID93] Ip, C-W. N., Dill, D. L., Better Verification through Symmetry, CHDL, April 1993.

[Je94] Jensen, K., Colored Petri Nets: Basic Concepts, Analysis Methods, and Practical Use, vol. 2: Analysis Methods, EATCS Monographs, Springer-Verlag, 1994.

[JR91] Jensen, K., and Rozenberg, G. (eds.), High-level Petri Nets: Theory and Application, Springer-Verlag, 1991.

[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

[KP83] Kozen, D., and Parikh, R. A Decision Procedure for the Propositional Mu-calculus, Proc. of the Workshop on Logics of Programs, Carnegie-Mellon University, Springer LNCS no. 164, pp. 176–192, June 6–8, 1983.

[KT87] Kozen, D. and Tiuryn, J., Logics of Programs, in Handbook of Theoretical Computer Science, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[Ku86] Kurshan, R. P., "Testing Containment of omega-regular Languages", Bell Labs Tech. Report 1121-861010-33 (1986); conference version in R. P. Kurshan, "Reducibility in Analysis of Coordination", LNCIS 103 (1987) Springer-Verlag 19-39.

[Ku94] Kurshan, R. P., *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach* Princeton University Press, Princeton, New Jersey 1994.

[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.

[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] Litchtenstein, O., and Pnueli, A., Checking That Finite State Concurrent Programs Satisfy Their Linear Specifications, 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.

[Lo+94] Long, D., Browne, A., Clarke, E., Jha, S., Marrero, W., An Improved Algorithm for the Evaluation of Fixpoint Expressions, Proc. of 6th Inter. Conf. on Computer Aided Verification, Stanford, Springer LNCS no. 818, June 1994.

[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.

[MP92] Manna, Z. and Pnueli, A., Temporal Logic of Reactive and Concurrent Systems: Specification, Springer-Verlag, 1992

[McM92] McMillan, K., Symbolic Model Checking: An Approach to the State Explosion Problem, Ph. D. Thesis, Carnegie-Mellon University, 1992.

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

[MP71] McNaughton, R. and Pappert, S. Counter-Free automata, MIT Press, 1971.

[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.

[Mo84] Mostowski, A. W., Regular Expressions for Infinite Trees and a Standard Form of Automata, Proc. 5th. Symp on Computation Theory, Zaborow, Poland, pp. 157–168, Springer LNCS no. 208, 1984.

[Mu84] Muchnik, A. A., Games on Infinite Trees and Automata with Dead-ends: a New Proof of the Decidability of the Monadic Theory of Two Successors, Semiotics and Information, 24, pp. 17–40, 1984 (in Russian).

[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

[Niw84] Niwinski, D., unpublished manuscript.

[Niw88] Niwinski, D., Fixed Points versus Infinite Generation, Proc. 3rd IEEE Symp. on Logic in Computer Science, pp. 402–409, 1988.

[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.

[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

[Pi90] Pixley, C., A Computational Theory and Implementation of Sequential Hardware Equivalence, CAV'90 DIMACS series, vol.3 (also DIMACS Tech. Report 90-3 1), eds. R. Kurshan and E. Clarke, June 1990.

[Pi92] Pixley, C., A Theory and Implementation of Sequential Hardware Equivalence, IEEE Transactions on Computer-Aided Design, pp. 1469–1478, vol. 11, no. 12, 1992.

[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.

[PR89] A. Pnueli and R. Rosner, On the Synthesis of a Reactive Module, 16th Annual ACM Symp. on Principles of Programing Languages, pp. 179-190, Jan. 1989.

[PR89b] A. Pnueli and R. Rosner, On the Synthesis of an Asynchronous Reactive Module, Proc. 16th Int'l Colloq. on Automata, Languages, and Programming, Stresa, Italy, July, 1989, pp. 652-671, Springer-Verlag LNCS no. 372.

[Pr81] Pratt, V., A Decidable Mu-Calculus, 22nd FOCS, pp. 421-427, 1981.

[Pr67] Prior, A., Past, Present, and Future, Oxford Press, 1967.

[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.

[Ra69] Rabin, M. O., "Decidability of Second Order Theories and Automata on Infinite Trees", *Trans. AMS*, 141(1969), pp. 1-35.

[Ra72] Rabin, M. O., Automata on Infinite Objects and Church's Problem, Conf. Board. of Math. Sciences, Regional Series in Math. no. 13, Amer. Math. Soc., Providence, Rhode Island, 1972.

[Ra71] Rackoff, C., The Emptiness and Complementation Problems for Automata on Infinite Trees, Master's Thesis, EECS Dept, MIT, 1971.

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

[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.

[Sa92] Safra, S., Exponential Determinization for ω-Automata with Strong Fairness Acceptance Condition, Proceedings of the 24th Annual ACM Symposium on the Theory of Computing, pp. 275-282, Victoria, May 1992.

[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 Axiomatized in Temporal Logic?, Information & Control, vol. 63., nos. 1/2, Oct./Nov. 84, pp. 88-112.

[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.

[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.

[St81] Streett, R., Propositional Dynamic Logic of Looping and Converse, PhD Thesis, MIT, 1981; journal version appears in Information and Control 54, 121-141, 1982.

[SE84] Streett, R., and Emerson, E. A., The Propositional Mu-Calculus is Elementary, ICALP84, pp 465 -472, July 84; journal version appears as An Automata Theoretic Decision Procedure for the Propositional Mu-calculus, Information and Computation v. 81, no. 3, June 1989.

[SW89] Stirling, C., and Walker, D., Local Model Checking in the Mu-calculus, pp. 369–383, Springer LNCS no. 351, 1989.

[St93] Stirling, C., Modal and Temporal Logics. in Handbook of Logic in Computer Science, (D. Gabbay, ed.) Oxford, 1993

[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

[Th91] Thomas, W., Automata on Infinite objects, in Handbook of Theoretical Computer Science, vol. B, (J. van Leeuwen, ed.), Elsevier/North-Holland, 1991.

[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

[Va88] Vardi, M. A Temporal Fixpoint Calculus, POPL, 1988.

[Va9?] Vardi, M, An Automata-theoretic Approach to Linear Temporal Temporal Logic, this volume.

[VS85] Vardi, M. and Stockmeyer, L., Improved Upper and Lower Bounds for Modal Logics of Programs, Proc. 17th ACM Symp. on Theory of Computing, pp. 240–251, 1985.

[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.

[Wo87] Wolper, P., On the Relation of Programs and Computations to Models of Temporal Logic, in Temporal Logic and Specification, Springer-Verlag LNCS no. 398, April 1987.

This article was processed using the $\LaTeX$ macro package with LLNCS style