DECIDING FULL BRANCHING TIME LOGIC

E. Allen Emerson & A. Prasad Sistla[1]

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-85-28 November 1985

Reprinted from *Information & Control*, Vol. 61, No. 3, pp. 175-201.

---

[1]Electrical & Computer Engineering Department, University of Massachusetts, Amherst, MA 01003.

Reprinted from INFORMATION AND CONTROL
Vol. 61, No. 3, June 1984
Printed in Belgium

Deciding Full Branching Time Logic*

E. ALLEN EMERSON†

Computer Sciences Department, University of Texas, Austin, Texas 78712

AND

A. PRASAD SISTLA‡

Electrical and Computer Engineering Department, University of Massachusetts, Amherst, Massachusetts 01003

In this paper the full branching time logic (CTL*) is studied. It has basic modalities consisting of a path quantifier, either A ("for all paths") of E ("for some path"), followed by an arbitrary linear time assertion composed of unrestricted combinations of the usual linear temporal operators F ("sometime"), G ("always"), X ("nexttime"), and U ("until"). It is shown that the problem of determining if a CTL* formula is satisfiable in a structure generated by a binary relation is decidable in triple exponential time. The decision procedure exploits the special structure of the finite state ω-automata for linear temporal formulae which allows them to be determinized with only a single exponential blowup in size. Also the expressive power of tree automata is compared with that of CTL* augmented by quantified auxillary propositions.

1. INTRODUCTION

A number of systems of branching time temporal logic have been proposed for reasoning about *existential* properties of concurrent programs (e.g., potential for deadlock along *some* future) in addition to *universal* properties (e.g., inevitability of service along *all* futures). The modalities of these logics are of the general form: either A (for *all* paths) or E (for *some* path) followed by a combination of the usual linear time operators F (sometime), G (always), X (nexttime), and U (until). In many such logics restrictions are placed on how the linear time operators can combine with the path quantifiers. For example, in the logic *UB* of (Ben-Ari, Manna, and Pnuelli, 1981), A or E is always paired with a single occurrence of F, G, or

* Preliminary versions of some of these results were presented at the 1983 CMU Workshop on Logics of Programs and the 1984 ACM Symposium on Theory of Computing.

† The first author was partially supported by NSF Grant MCS-8302878.

‡ The second author was partially supported by NSF Grant MCS-8105553.

0019-9958/84 $3.00

X. While these restrictions can reduce the complexity of reasoning in a logic, they can also significantly limit the logic's expressive power. For instance, a property associated with *fairness* such as "along some future an event P occurs infinitely often" can be formulated as $EGFP$; however, this formula involves a nesting of F inside G violating the restrictions of UB's syntax and is provably (cf. Emerson and Halpern, 1983) not equivalent to any UB formula.

In this paper, we study the full branching time temporal logic CTL* of (Emerson and Halpern, 1983) in which a path quantifier A or E can prefix an assertion composed of *unrestricted* combinations (i.e., involving arbitrary nestings and boolean connectives) of the linear time operators F, G, X, and U; CTL* subsumes a number of logics from the literature including the systems of (Manna and Pnueli, 1979; Lamport, 1980; Gabbay *et al.*, 1980; Ben-Ari, Manna, and Pnueli, 1981; Emerson and Halpern, 1982; and Clarke, Emerson, and Sistla, 1983), as well as the *Computation Tree Logic* of (Clarke and Emerson, 1981). (It is also closely related to the logic MPL of (Abrahamson, 1980); see below.) We interpret CTL* formulae over *R-generable* models (cf. Emerson, 1983)—i.e., structures generated by a binary relation like those used in (Fischer and Ladner, 1979; and Ben-Ari, Manna, and Pnueli, 1981). We show that satisfiability for CTL* with this semantics is decidable in triple exponential time.

Somewhat surprisingly, for some time it was not known if there was a decision procedure of elementary complexity for full branching time logic interpreted over this very natural class of structures. In (Abrahamson, 1980) a logic, MPL is defined which has a very similar syntax to CTL* but somewhat different semantics. While a double exponential decision procedure is given for MPL interpreted over structures which violate the R-generability condition, for semantics (corresponding to) R-generable structures, (Abrahamson, 1980) gives only a nonelementary decision procedure [1] and states that the existence of an elementary procedure is open. Recently, other researchers (Pnueli and Sherman, 1983; Vardi and Wolper, 1983) have, independently, announced four exponential decision procedures for the R-generable case. Our procedure is thus exponentially faster. We can give a faster decision procedure, in part, because we uncover some structural properties of branching time and linear time logics which had gone heretofore unnoticed.

To get our decision procedure, we first show that given any CTL* formula f_0 we can derive an "equivalent" formula f_1 of length $O(|f_0|)$ in which the depth of nesting of path quantifiers is at most *two*. This establishes a *normal form* for CTL* which is essentially conjunctions and disjunctions

[1] The decision procedure is obtained by translation into SnS, the second order monadic theory of n successors; by the results of (Meyer, 1974), SnS is not elementary recursive.

of subformulae of the form Ap_0, $AGEp_0$, and Ep_0, where p_0 is a pure linear time formulae (i.e., p_0 contains no path quantifiers). We then argue that f_1 is satisfiable iff it has an infinite tree-like model, where the branching at each node is bounded by $|f_1|$. This enables us to reduce the satisfiability problem to the emptiness problem for finite automata on infinite trees (Rabin, 1969): For each subformula Ap_0, $AGEp_0$, or Ep_0, we build a *complemented pairs tree automaton* of size at most double exponential in $|p_0|$. These tree automata are then combined using a cross product construction to get a complemented pairs tree automaton for f_1 of size at most double exponential in $|f_1|$ which accepts infinite trees that define models of f_1. By the results of (Streett, 1981) the emptiness problem of this tree automaton is decidable in time exponential in its size, i.e., in time triple exponential in $|f_0|$. As a corollary, we also obtain a small model theorem since an automaton accepts an infinite tree iff it accepts a finitely generable tree obtained by "unwinding" a finite tree (Rabin, 1969; Hossley and Rackoff, 1972).

Building the tree automata for $AGEp_0$ or Ep_0 is straightforward. However, design of the tree automaton for Ap_0 is much more subtle. A *tableau* construction can be applied to p_0 to get a nondeterministic (Buchi) automaton $\mathscr{A}_1$ on infinite strings (where acceptance is defined by repeating a designated set of states infinitely often) recognizing $\{x: x \vDash p_0\}$ with $N = \exp(|p_0|)$ states. A seemingly natural next step would be to program the tree automaton to simply run $\mathscr{A}_1$ down every path from the root of the input tree to check that p_0 indeed holds along every path. In fact, for this tree automaton to work correctly, the string automaton must be *deterministic*. It is well known that the subset construction (Rabin and Scott, 1959) cannot *in general* be used to determinize finite automata on infinite strings; instead, the "classical" method for determinizing such an automaton involves application of McNaughton's (1966) construction and yields an equivalent deterministic string automaton with a number of states, that is, *double exponential* in N. However, we show that $\mathscr{A}_1$ has a special structure derived from the tableau which allows us to obtain, by means of a rather delicate construction, an equivalent deterministic automaton with a number of states only *single exponential* in N. This in turn enables us to construct the tree automaton for Ap_0 of the desired size.

Last, we compare the expressive power of branching time logic with tree automata. We show that while CTL* itself is less expressive than tree automata, CTL* (resp., UB) with quantification over auxillary propositions is as expressive as pairs (resp. Buchi) tree automata.

The remainder of the paper is organized as follows: In Section 2 we give some preliminary definitions. Then in Section 3 we discuss the normal form and tree-like models. Section 4 shows how the tableau for a linear time formula defines a Buchi automaton and describes its special structure while Section 5 shows how to determinize it with only a single exponential blowup.

The design of the tree automata is given in Section 6, and Section 7 gives our expressiveness results. Section 8 presents some concluding remarks.

2. Preliminaries: Definitions and Terminilogy

2.1. *Syntax.* We will inductively define a class of state formulae (true or false of states) and a class of path formulae (true or false of paths). We use the large roman letters P, Q, R,..., to represent atomic propositions and small the roman letters p, q, r,..., to represent nonatomic (state or path) formulae:

(S1) Any 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 is a state formula.

(P1) Any state formula p is a path formula.

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

(P3) If p, q are path formulae then so are Xp, $(p\ U\ q)$.

The set of state formulae generated by all of the above rules forms the language CTL* of full branching time temporal logic (cf. Emerson and Halpern, 1983) while the set of path formulae generated by rules (S1), (P1), (P2), and (P3) forms the language $L(X, U)$ of ordinary linear time temporal logic (cf. Emerson and Clarke, 1982). (We call this latter type of path formula a *pure path formula* or a *pure linear time formula* to emphasize that it contains no nested A's or E's.) The other connectives can then be defined as abbreviations: $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, $p \equiv q$ abbreviates $(p \Rightarrow q) \wedge (q \Rightarrow p)$, Ap abbreviates $\neg E \neg p$, Fp abbreviates $true\ U p$, Gp abbreviates $\neg F \neg p$, and $(p\ W\ q)$ abbreviates $\neg(\neg p\ U \neg q)$. (Note: $|p|$ denotes the *length* of p viewed as a string in the obvious way.)

2.2. *Semantics.* We define the semantics of a CTL* formula with respect to a structure $M = (S, R, L)$, where

(1) S is a nonempty *set of states*,

(2) R is a nonempty total *binary relation* on S, and

(3) L is a *labelling* which assigns to each state a set of atomic propositions true in the state.

A *fullpath* $(a_1, a_2, a_3, \ldots)$ is an infinite sequence of states such that $(a_i, a_{i+1}) \in R$ for all i. We write $M, a \models p$ $(M, x \models p)$ to mean that state formula p (path formula p) is true in structure M at state a (of pah x, respectively). When M is understood, we write simply $a \models p$ $(x \models p)$. We define $\models$ inductively using the convention that $x = (a_1, a_2, a_3, \ldots)$ denotes a path and x^i denotes the suffix path $(a_i, a_{i+1}, a_{i+2}, \ldots)$:

(S1) $a \models P$ iff $P \in L(s)$, for any atomic proposition P

(S2) $a \models p \wedge q$ iff $a \models p$ and $a \models q$

$a \models \neg p$ iff not $(a \models p)$

(S3) $a \models Ep$ iff for some fullpath x starting at a, $x \models p$

(P1) $x \models p$ iff $a_1 \models p$, for any state formula p

(P2) $x \models p \wedge q$ iff $x \models p$ and $x \models q$

$x \models \neg p$ iff not $(x \models p)$

(P3) $x \models Xp$ iff $x^2 \models p$

$x \models (p\ U\ q)$ iff for some $i \geqslant 1$, $x^i \models q$ and for all $j \geqslant 1$ $[j \leqslant i$ implies $x^j \models p]$

We say that state formula p is *valid*, and write $\models p$, if for every structure M and every state a in M, $M, a \models p$. We say that state formula p is *satisfiable* if for some structure M and some state s in M, $M, a \models p$. In this case we also say that M defines a *model* of p. We define validity and satisfiability similarly for path formulae.

Note that, given a pure path formula p_0 and a path x, only the truth values of the atomic propositions actually appearing in p_0 matter in determining whether $x \models p_0$. We can thus view a path $x = a_1 a_2 a_3 \cdots$, as an infinite string of sets of atomic propositions of p_0 (so each $a_i \in$ PowerSet (AtomicPropositions(p_0)) where AtomicPropositions(p_0) denotes the set of atomic propositions appearing in p_0).

2.3. DEFINITION. Given a pure path formula p_0, *the Fischer–Ladner closure of* p_0 is the least set $\mathrm{FL}(p_0)$ of subformulae of p_0 such that

(1) $p_0 \in \mathrm{FL}(p_0)$

(2) if $p \wedge q \in \mathrm{FL}(p_0)$ then $p, q \in \mathrm{FL}(p_0)$

(3) if $\neg p \in \mathrm{FL}(p_0)$ then $p \in \mathrm{FL}(p_0)$

(4) if $(p\ U\ q) \in \mathrm{FL}(p_0)$ then $p, q, X(p\ U\ q) \in \mathrm{FL}(p_0)$

(5) if $Xp \in \mathrm{FL}(p_0)$ then $p \in \mathrm{FL}(p_0)$.

Note. $|\mathrm{FL}(p_0)| = O(|p_0|)$.

The *extended Fischer–Ladner closure of* p_0, $EFL(p_0)$, is the set $\mathrm{FL}(p_0) \cup \{\neg p : p \in \mathrm{FL}(p_0)\}$.

2.4. DEFINITION. A set $s \subseteq \mathrm{EFL}(p_0)$ is *maximal* provided that $\forall p = \neg q \in \mathrm{EFL}(p_0)$, at least one of q, $\neg q \in s$. A set $s \subseteq \mathrm{EFL}(p_0)$ is *consistent* provided that

(1) $\forall p = \neg q \in s$ at most one of $q, \neg q \in s$

(2) $(p \wedge q) \in s$ iff $p \in s$ and $q \in s$
$\neg(p \wedge q) \in s$ iff $\neg p \in s$ or $\neg q \in s$

(3) $(p\, U\, q) \in s$ iff $q \in s$ or $p, X(p\, U\, q) \in s$
$\neg(p\, U\, q) \in s$ iff $\neg q, \neg p \in s$ or $\neg q, \neg X(p\, U\, q) \in s$.

2.5. DEFINITION. The *tableau* for p_0 is a labelled, directed graph $\mathscr{T} = (V, R)$, where the set of nodes $V = \{s \subseteq \mathrm{EFL}(p_0)$: s is maximal and consistent$\}$ and $R = \{$arcs $s \to t$: $s, t \in V$ and for each formula $Xp \in \mathrm{EFL}(p_0)$ $[Xp \in s$ iff $p \in t]\}$.

2.6. *Terminology.* The symbols $\overset{\infty}{\exists}$ and $\overset{\infty}{\forall}$ are read "there exist infinitely many" and "for all but a finite number," respectively. We write *i.o.* to abbreviate "infinitely often," *f.o.* to abbreviate "only finitely often," and *a.e.* to abbreviate "almost everywhere" (meaning "at all but a finite number of instances"). We extend the *AtomicPropositions* (c) notation to indicate the set of all atomic propositions appearing in formula c or elements of node c or input symbol c. We also write $\exp(n)$ to indicate c^n for some $c > 1$. We further use $\exp^2(n)$ to abbreviate $\exp(\exp(n))$ and $\exp^3(n)$ for $\exp(\exp(\exp(n)))$.

2.7. *Finite Automata on Infinite Strings and Infinite Trees.* There is an extensive literature for finite automata on infinite strings and on infinite trees, and the reader is referred to (McNaughton, 1966; Rabin, 1969, 1970; Hossley and Rackoff, 1972) as well as (Street, 1981). For now, we briefly review the following definitions:

A *finite automaton* $\mathscr{A}$ *on infinite strings* consists of a tuple (Σ, S, δ, s_0), where Σ is the finite *input alphabet*, S is the finite set of *states*, $\delta: S \times \Sigma \to \mathrm{PowerSet}(S)$ is the *transition function*, and $s_0 \in S$ is the *start state*—plus an *acceptance condition* as described subsequently. A *run* r of $\mathscr{A}$ on infinite input string $x = a_1 a_2 a_3 \cdots$, is an infinite sequence $r = s_0 s_1 s_2 s_3 \cdots$, of states such that $\forall i \geqslant 0$, $\delta(s_i, a_{i+1}) \supseteq \{s_{i+1}\}$. For a *Buchi* automaton acceptance is defined in terms of a distinguished set of states, GREEN, (think of a green light flashing upon entering any state of GREEN): x is *accepted* iff there exists a run r on x such that $\overset{\infty}{\exists}$ GREEN flashes along r. For a *pairs* automaton we have a finite list $((\mathrm{RED}_1, \mathrm{GREEN}_1), \ldots, (\mathrm{RED}_k, \mathrm{GREEN}_k))$ of pairs of sets of states (think of them as pairs of colored lights, where $\mathscr{A}$ flashes the red light of the 1st pair upon entering any state of set RED_1, etc.): x is *accepted* iff there exists a run r on x such that for some pair $i \in [1{:}k]$ ($\neg \overset{\infty}{\exists}$ RED_i flashes and $\overset{\infty}{\exists}$ GREEN_i flashes) along r. Finally, a *complemented pairs* automaton *accepts* x iff there exists a run r on x such that the above pairs condition is false, i.e., iff for all pairs $i \in [1{:}k]$ ($\overset{\infty}{\exists}$ GREEN_i flashes implies $\overset{\infty}{\exists}$ RED_i flashes) along r.

Let $\Gamma_n = \{b_0, b_1, ..., b_{n-1}\}$ be an alphabet over n distinct symbols $b_0, ..., b_{n-1}$. Then Γ_n^* may be viewed as an *infinite n-ary tree* T_n, where the empty string λ is the *root* node and each node t has as its *successors* the nodes $tb_0, ..., tb_n$. A finite (infinite) *path* through T_n is a finite (resp., infinite) sequence $x = t_0, t_1, t_2, ...$, of nodes such that for all i, t_{i+1} is a successor of t_i. An *infinite n-ary Σ-tree* is a labelling ϕ which maps $T_n \to \Sigma$.

A *finite automaton* $\mathscr{A}$ *on infinite n-ary Σ-trees* consists of a tuple (Σ, S, δ, s_0) plus an acceptance condition similar to a string automaton except that $\delta: S \times \Sigma \to \text{PowerSet}(S^n)$. A *run* of $\mathscr{A}$ on Σ-tree ϕ is a function $\rho: T_n \to S$ such that for all $s \in T_n(\rho(sb_0), ..., \rho(sb_{n-1})) \in \delta(\rho(s), \phi(s))$. We say that $\mathscr{A}$ *accepts* input Σ-tree ϕ iff $\exists$ a run ρ of $\mathscr{A}$ on ϕ such that $\forall$ path x starting at the root of T_n, if $r = \rho \mid x$, the sequence of states $\mathscr{A}$ goes through along path x, then the string acceptance condition (as above) holds along r.

3. Normal Form and Tree Models

3.1. Theorem. *Given any* CTL* *formula f_0 we can construct a corresponding formula f_1 in a normal form composed of conjunctions and disjunctions of subformulae of the form Ap_0, Ep_0, or $AGEp_0$, where p_0 is a pure linear time formula such that* (1) *f_1 is satisfiable iff f_0 is satisfiable and* (2) *$|f_1| = O(|f_0|)$. Moreover, any model of f_1 can be used to define a model of f_0 and conversely.*

Proof. We will initially obtain a preliminary normal form f_2 composed of conjunctions and disjunctions of subformulae of the form Ap_0, Ep_0, or $AG(P \equiv A/Ep_0)$, where P denotes an atomic proposition or its negation and, for brevity, we write A/Ep to indicate a formula of either the form Ap or Ep. We will then apply the validities $AG(Q \equiv Ep) \equiv AG(\neg Q \equiv A\neg p)$ and $AG(Q \equiv Ap) \equiv (A[G(Q \Rightarrow p)] \wedge AGE(\neg Q \Rightarrow \neg p))$ to transform f_2 into f_1 in the final normal form.[2]

To get the preliminary form, we first drive negations inward using DeMorgan's laws and dualities such as $\neg Fp \equiv G\neg p$, $\neg Ap \equiv E\neg p$, etc. so that only atomic propositions appear negated. The resulting formula f_3 consists of conjunctions and disjunctions of the form $g = A/Ep$, where each p is a path formula possibly containing nested A's or E's. We then reduce each such g appearing in f_3 to the form $g^0 = A/Ep^0 \wedge \bigwedge_{i=1}^{n} AG(Q_i \equiv A/Eq_i)$, where p^0 and the q_i are all pure path formulae and where $n \leqslant |f|$. We do this by introducing "fresh" atomic propositions for each "deeply"

[2] The verification of these identities is straightforward and involves applying valid equivalences such as $|AG(Q \Rightarrow Ep)| \equiv |AGE(Q \Rightarrow p)|$ where Q is an arbitrary atomic proposition and p is an arbitrary state formula. The reader may note that the generalized equivalence $|AG(q \Rightarrow Ep)| \equiv |AGE(q \Rightarrow p)|$ where p and q are both arbitrary state formulae is also valid. However, the formula $|AG(r \Rightarrow Ep)| \equiv |AGE(r \Rightarrow p)|$ where p is an arbitrary state formula but r is an arbitrary *path* formula is *not* a valid equivalence.

nested A/Ep subformula. For example, $E(GEFAFP \wedge FAGR)$ becomes $E(GEFQ_1 \wedge FAGR) \wedge AG(Q_1 \equiv AFP)$ which becomes $E(GQ_2 \wedge FAGR) \wedge AG(Q_1 \equiv AFP) \wedge AG(Q_2 \equiv EFQ_1)$ which finally becomes $E(GQ_2 \wedge FQ_3) \wedge AG(Q_1 \equiv AFP) \wedge AG(Q_2 \equiv EFQ_1) \wedge AG(Q_3 \equiv AGR)$.

To describe the reduction formally, let $g_0 = g$. Inductively, assume we have $g_k = A/Ep_k \wedge \bigwedge_{i=1}^{k} AG(Q_i \equiv A/Eq_i)$, where the q_i are pure path formulae but p_k may not be. If p_k is a pure path formula, we are done. Otherwise, let A/Eq_{k+1} be a subformula of p_k such that q_{k+1} is a pure path formula. Then let p_{k+1} be the result of substituting a unique, previously unused atomic proposition Q_{k+1} for A/Eq_{k+1} in p_k and define $g_{k+1} = A/Ep_{k+1} \wedge \bigwedge_{i=1}^{k+1} AG(Q_i \equiv A/Eq_i)$. Note that g_{k+1} is satisfiable iff g_k is satisfiable. In particular, a model of g_k defines a model of g_{k+1} by extending the labelling so that that Q_{k+1} is true exactly at the states where A/Eq_{k+1} holds. Conversely, a model of g_{k+1} must be a model of g_k.

This reduction process must terminate within $n \leqslant |p|$ steps because $|p_{k+1}| < |p_k|$. When it does terminate, let $p^0 = p_n$ so that $p^0, q_1, \ldots, q_n$ are all pure path formulae. Moreover, $|g^0| = O(|g|)$ since $|g^{k+1}| = |g^k| +$ some constant C, as can be seen by transforming g_{k+1} into $g_k \wedge AG(Q_{k+1} \equiv Q_{k+1})$ by textually swapping the occurrence of Q_{k+1} in p_{k+1} with A/Eq_{k+1} in $AG(Q_{k+1} \equiv A/Eq_{k+1})$.

The reduced formula f_2 is of length $O(|f_0|)$ and is in the preliminary normal form. Since f_1 is of length about $2 \cdot |f_2| = O(|f_0|)$, we are done. ∎

It is well known that any R-generable model may be unwound into an equivalent infinite tree-like model. Using an approach similar to that of (Street, 1981) we can ensure that the resulting tree-like model has some additional structure wich simplifies programming the tree automata:

Suppose $M = (S, R, L)$ is a model of f_1 so that $M, s_0 \models f_1$. We will construct another model $M' = (S', R', L')$ with $S' = \Gamma_{n+1}^*$ where Γ_{n+1}^* is the alphabet $\{b_0, b_1, \ldots, b_n\}$ and $Ep_1, \ldots, Ep_n$ are all the Ep subformulae of f_1. Intuitively, M' is obtained by unravelling M so that each Ep_i subformula is satisfied along a *designated* path of M' which is a copy of a correponding path in M. We define a function $g: S' \to S$. Let $g(\lambda) = s_0$, where λ is the empty string. Inductively, assume $g(z)$ is defined $= t_0$. For each subformula Ep_k, if $M, t_0 \models Ep_k$ then let $x = t_0, t_1, t_2, \ldots$, be a path in M such that p_k holds along it. Then let $zb_k b_0^\omega$ be a "copy" of x, i.e., let $g(z) = t_0$, $g(zb_k) = t_1$, $g(zb_k b_0) = t_2$, $g(zb_k b_0 b_0) = t_3$, etc. Now define R' by the rule $(z_1, z_2) \in R'$ iff (i) $z_2 = z_1 b_i$ for some $i \in [0{:}\, n]$ and (ii) $(g(z_1), g(z_2)) \in R$. Finally, let $L'(z) = L(g(z))$. (Note: let zb_0^ω be a copy of a path starting at $g(z)$.)

By construction of M', every path starting at λ of M' is a copy of a path starting at s_0 of M. Hence, if $M, s_0 \models Ap$ then $M', \lambda \models Ap$. In addition, for every state z of M' and for each Ep_i subformula, if $M, g(z) \models Ep_i$ then $M', z \models Ep_i$. Thus, if $M, s_0 \models Ep$ ($M, s_0 \models AGEp$) then $M', \lambda \models Ep$ (resp. $M', \lambda \models AGEp$). It follows that $M', \lambda \models f_1$. We have thus shown

3.2. THEOREM. *For any formula f_1 of* CTL* *in the above normal form, if f_1 has a model M, then it has an infinite tree-like model M' where each node is of outdegree $\leqslant |f_1|$. Moreover, each Ep subformula of f_1 that holds in M is satisfied along a designated path of the tree-like model M'.*

4. THE TABLEAU AS A NONDETERMINISTIC FINITE AUTOMATON

We may view the tableau for a linear time formula p_0 as defining the transition diagram of a nondeterministic finite automaton $\mathscr{A}$ on infinite strings which accepts $\{x: x \models p_0\}$ by letting the arc $u \to v$ be labelled with AtomicPropositions(v). A run r of $\mathscr{A}$ on input $x = a_1 a_2 a_3 \cdots$, is an infinite sequence $r = s_0 s_1 s_2 s_3 \cdots$, of tableau nodes such that $\forall i \geqslant 0$, $\delta(s_i, a_{i+1}) \supseteq \{s_{i+1}\}$, where δ is the transition function of $\mathscr{A}$. (Actually, s_0 is not a tableau node but the unique *start state* defined so that $\delta(s_0, a) = \{$tableau nodes $u: p_0 \in u$ and AtomicPropositions(u) = AtomicPropositions(a)$\}$.) Note that $\forall i \geqslant 1$ AtomicPropositions(s_i) = AtomicPropositions(a_i). Any run of $\mathscr{A}$ would correspond to a model of p_0 (in that $\forall i \geqslant 1$, $x^i \models \{$formulas $p: p \in s_i\}$) except that eventualities might not be fulfilled. To check fulfillment, we can define acceptance via complemented pairs: if $EFL(p_0)$ has m eventualities, we let $\mathscr{A}$ have m pairs $(\text{RED}_i, \text{GREEN}_i)$ of lights. Each time a state containing $(p_i U q_i)$ is entered, flash GREEN_i; each time a state containing q_i is entered, flash RED_i. A run r *accepted* iff $\forall i \in [1:m]$ $[[\overset{\infty}{\exists}\ \text{GREEN}_i$ flashes $\Rightarrow \overset{\infty}{\exists}\ \text{RED}_i$ flashes$]$, iff every eventuality is fulfilled, iff $x \models p_0$.

However, we find it more convenient to convert $\mathscr{A}$ into an elquivalent nondeterministic Buchi automaton, $\mathscr{A}_1$: We say that the eventuality $(p U q)$ is *pending* at state s of run r provided that $(p U q) \in s$ and $q \notin s$. Observe that run r of $\mathscr{A}$ on input x corresponds to a model of p_0 iff not ($\exists$ eventuality $(p U q)$, $(p U q)$ is pending a.e. along r) iff ($\forall$ eventuality $(p U q)$, $(p U q)$ is *not* pending i.o. along r). The Buchi automaton $\mathscr{A}_1$ is then obtained from $\mathscr{A}$ by augmenting the state with an $m+1$ valued counter so that a state of $\mathscr{A}_1$ is of the form (tableau component, counter component). (The start state of $\mathscr{A}_1$ is (start state of $\mathscr{A}$, 1).) The counter is incremented from i to $i+1$ (mod$(m+1)$) when the ith eventuality $(p_i U q_i)$ is next seen to be not pending along the tableau component of the run. 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 $\overset{\infty}{\exists}$ GREEN flashes iff $\forall i \in [1:m]$ $((p_i U q_i)$ is not pending i.o.), iff every pending eventuality is sometime fulfilled, iff $x \models p_0$. Moreover, $\mathscr{A}_1$ still has $N = \exp(|p_0|) \cdot O(|p_0|) = \exp(|p_0|)$ states.

The tableau has the following special structure:

4.1. LEMMA. *If s_1, s_2, t are nodes of $\mathscr{E}$ such that s_1, s_2 are both*

immediate predecessors of t, *and* $AtomicPropositions(s_1) = AtomicPropositions(s_2)$, *then* $s_1 = s_2$.

Proof. We argue by induction on the structure of formulas in s_1, s_2 that $p' \in s_1$ iff $p' \in s_2$, for all $p' \in EFL(p_0)$. The basis case of atomic propositions follows directly by assumption. Suppose $p' \in s_1$. If $p' = \neg p$ then $p \notin s_1$. By induction hypothesis, $p \notin s_2$. So $\neg p \in s_2$ by maximality. If $p' = p \wedge q \in s_1$ then consistency of s_1 implies $p, q \in s_1$. By induction hypothesis, $p, q \in s_2$, so, again, by consistency $p \wedge q \in s_2$. If $p' = Xp \in s_1$ then, by definition of the tableau, $p \in t$ and so $Xp \in s_2$.

Finally suppose $p' = (p\,U\,q) \in s_1$. By consistency, either $q \in s_1$ or $p, X(p\,U\,q) \in s_1$. If $q \in s_1$ then, by induction hypothesis, $q \in s_2$, so consistency implies $p\,U\,q \in s_2$ also. If $p, X(p\,U\,q) \in s_1$ then by induction hypothesis, $p \in s_2$. By definition of the tableau, $(p\,U\,q) \in t$ and also $X(p\,U\,q) \in s_2$. By consistency then, $(p\,U\,q) \in s_2$.

We just showed that $p' \in s_1$ implies $p' \in s_2$. By symmetry, $p' \in s_1$ iff $p' \in s_2$. ∎

The automaton $\mathscr{A}_1$ inherits from the tableau a similar special structure so that, essentially, different runs on the same input cannot merge:

4.2. THEOREM. *If* $r_1 = (s_0, s_1, s_2, \ldots)$ *and* $r_2 = (t_0, t_1, t_2, \ldots)$ *are two runs of* $\mathscr{A}_1$ *on input* x, *and* r_1, r_2 *"intersect" after having read the same finite prefix of* x (*technically*, $\exists k$, $s_k = t_k$), *then* r_1, r_2 *coincide up to the point of intersection* (*technically*, $\forall j \leqslant k$, $s_j = t_j$).

Proof. Let s_i' (t_i') denote the tableau component of s_i (resp., t_i). By hypothesis, $s_k = t_k$ and hence $s_k' = t_k'$. Since the two runs r_1 and r_2 are on the same input, for all $i \geqslant 1$, AtomicPropositions$(s_i') =$ AtomicPropositions(t_i'). Thus by repeatedly applying Lemma 4.1, we see that for all $j \leqslant k$, $s_j' = t_j'$ (i.e., the tableau components of the two runs coincide out to position k). Note that the counter component of the ith state along a run of $\mathscr{A}_1$ depends only on (i) the initial value of the counter and (ii) the tableau components of the preceeding states along the run. Since the start state of $\mathscr{A}_1$ is unique and since the two runs coincide in their tableau components out to position k, it follows that they also coincide in their counter components out to position k. Thus the two runs coincide entirely out to position k as claimed. ∎

Given a Buchi automaton $\mathscr{A}_1$ for linear time formula $p_0' = \neg p_0$ with $N = \exp(|p_0'|) = \exp(|p_0|)$ states, we will show in the next section how to construct an equivalent deterministic pairs automaton $\mathscr{A}^*$ of size $(\exp(N^2)$ states, N^2 pairs). Since $\mathscr{A}^*$ is deterministic and $\mathscr{A}^*$ accepts x iff $x \models \neg p_0$, we may view $\mathscr{A}^*$ as a deterministic *complemented* pairs automaton which accepts x iff $x \models p_0$. This will allow us to construct the desired tree automaton for Ap_0.

5. How to Determinize the Buchi Automaton

5.1. *The Run Tree.* The set of all runs of the nondeterministic Buchi automaton $\mathcal{A}_1$ on input x may be viewed as an infinite directed acyclic graph (DAG) of width $\leqslant N = \exp(|p_0|)$, where each node on level i of the DAG represents one of the possible states $\mathcal{A}_1$ could be in after having read the first i symbols of x. Since by Theorem 4.2 no two runs on x can merge, it is actually a tree. However, a run can *dead end*, (e.g., if $\neg Fp \in$ a node on level i and p appears in $i+1$th input symbol). Observe that, while there may be an infinite number of runs in this tree, *there are at most N distinct runs of infinite length*; the rest are finite. (In the sequel, we will say that a *P-node* of the run tree is one corresponding to a state of $\mathcal{A}_1$ where $\mathcal{A}_1$'s GREEN light flashes.)

5.2. *Intuition.* The dfa $\mathcal{A}^*$ is based on the *subset construction*—it builds the tree of all runs on input x, a level at a time—plus some machinery to do, roughly, a *depth-first search* of the run tree looking for an infinite run along which there are infinitely many P-nodes. The problem is complicated by the possibility that there may be infinitely many P-nodes in the run tree but only a finite number of them on any one path. Up to N markers are used in order to follow each active run. Associated with each marker i are N pairs of lights: $\langle i, 0\rangle, \ldots, \langle i, N-1\rangle$. There are thus a total of N^2 pairs of lights. The need for multiple pairs of light per marker is explained subsequently.

Intuitively, $\mathcal{A}^*$ operates as follows. As each symbol of x is read, the next level of the run tree is built from the current level, which will shortly become the new current level. (Only two levels are kept in memory at one time.) Each state of the current level is the tip of an active run which is associated with some marker i. Note that some runs split apart and others die out. Whenever (the) run (associated with marker) i splits, one alternative is followed by marker i and the other alternatives are assigned "free" (i.e., currently unused) markers $j_1 \cdots j_k$. We then say that the runs just started up, $j_1, \ldots, j_k$, spawn off run i. When and if run i dies, its marker becomes free for use with another run that may later start up. Since there are at most N active runs at any level, the N markers can be recycled indefinitely so that each active run is always assigned a marker.

We want each marker i to follow an infinite run if possible. However, run i may split apart many (even infinitely many) times. Some branches may be infinite and others finite. How does $\mathcal{A}^*$ know which of the alternatives is infinite and should be followed? If there were a way for $\mathcal{A}^*$ to know this, one pair of lights per run would suffice. For we could then simply have, for each run i, the pair of lights $\langle i, 0\rangle$ flash GREEN whenever marker i encountered a P-node and flash RED whenever run i encountered a dead end (see Fig. 1a, b). (The RED flashes are needed to ensure that an infinite number of "noncollinear" P-nodes do not cause erroneous acceptance.)

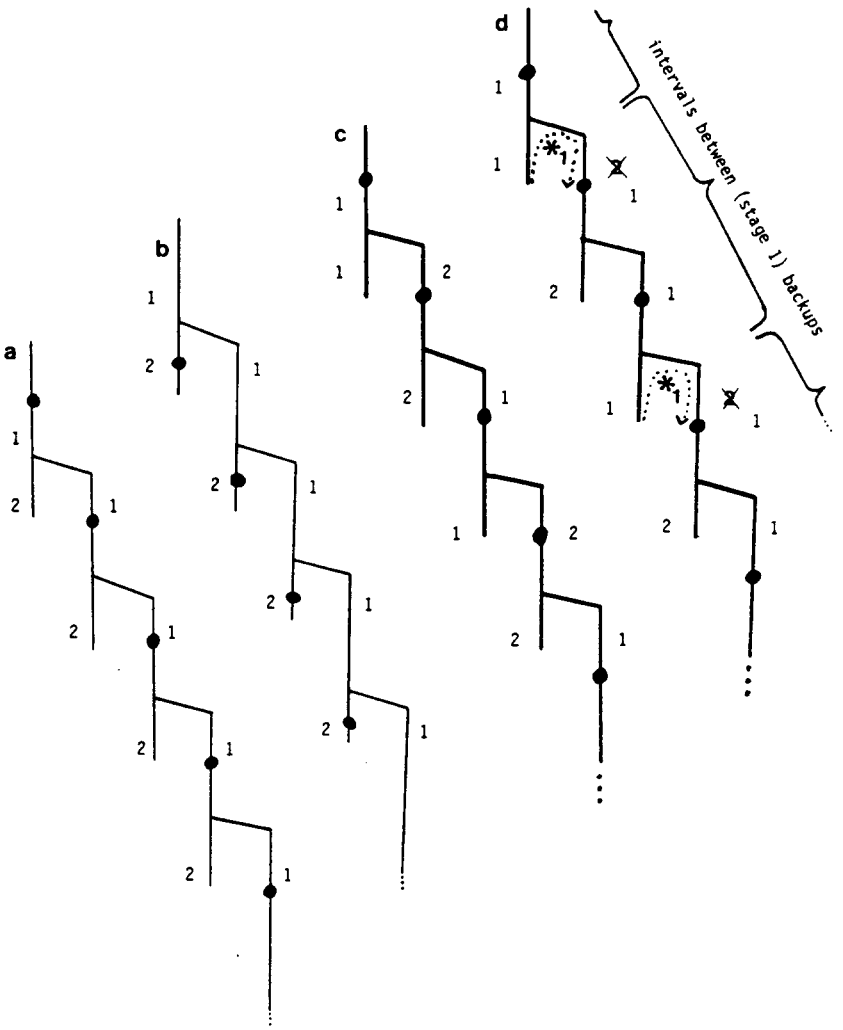

FIG. 1. (a) Correctly accepts because $\langle 1, 0\rangle$ flashes GREEN i.o.; (b) Correctly rejects because $\langle 2, 0\rangle$ flashes RED i.o.; (c) Erroneously rejects because $\langle 2, 0\rangle$, $\langle 1, 0\rangle$ both flash RED i.o.; (d) Accepts because backups allow run 1 to follow the infinite path. *Note.* In these figures, ● denotes a *P*-node.

However, there is in general no way for $\mathscr{A}^*$ to know which alternatives to follow because this depends on the suffix of the input yet to be read: one suffix might make alternative A infinite and alternative B finite while another suffix might do the opposite. Since $\mathscr{A}^*$ is deterministic, on some inputs it may repeatedly make poor decisions in which case the above rules can lead to false results. For example, in Fig. 1c, $\mathscr{A}^*$ erroneously rejects because both $\langle 1, 0\rangle$ and $\langle 2, 0\rangle$ flash RED as well as GREEN i.o.

The problem is that the single infinite path in the run tree has been parsed into infinitely many finite pieces rather than a single infinite piece. The solution is to have any run i which dead-ends *back up*—but as little as possible—by taking over the "youngest" surviving run j which previously spawned off i. For example, in Fig. 1d because "father" run 1 is older than its "son" run 2 (it was "born" earlier), when run 1 dead-ends it takes over its youngest son, run 2. The rules for the backup require that $\mathscr{A}^*$ flash RED on pairs $\langle 2, 0\rangle$, $\langle 2, 1\rangle$ since run 2 is totally obliterated when run 1 takes it over. $\mathscr{A}^*$ also flashes RED on the pair $\langle 1, 0\rangle$. This ensures that $\mathscr{A}^*$ will not falsely accept due to GREEN flashes on $\langle 1, 0\rangle$ caused by noncollinear P-nodes detected by run 1 prior to backups. Then, $\mathscr{A}^*$ flashes GREEN on the pair $\langle 1, 1\rangle$ iff a P-node has been seen on the finite path from the site of the previous backup of run 1 to the site of the current backup (indicated by *'s).

Consider the simple case where the width N of the run tree is at most 2. Then for any input x, one of two situations obtains:

(1) After a certain depth, $\mathscr{A}^*$ always makes "good" decisions and run 1 never again has to backup. Then pair $\langle 1, 0\rangle$ will never again flash RED. It will flash GREEN i.o. iff $\overset{\infty}{\exists}$ P-nodes along the run 1.

(2) $\mathscr{A}^*$ makes infinitely many "poor" decisions so that run 1 backs up i.o. in which case $\langle 1, 0\rangle$ flashes RED i.o. Then $\overset{\infty}{\exists}$ P-nodes along run 1 iff $\overset{\infty}{\exists}$ GREEN flashes of $\langle 1, 1\rangle$.

In general, when the width $N \geqslant 2$, we have N pairs of lights and associated *stages* of backups for each marker i. (By convention, when marker i is pushed from a node to a successor node without any actual backup we have a stage 0 backup of run i. P-nodes detected in this way are "recorded" via GREEN flashes of $\langle i, 0\rangle$.) Roughly, ancestor run i takes over descendent run j in a backup of stage m when the highest stage of previous backups of run i which must be "undone" is $m - 1$ (See Fig. 2). P-nodes detected by run i on the path between consecutive stage m backup points are recorded via GREEN flashes of $\langle i, m\rangle$.

5.3. *The Spawning Tree.* To perform these backups, $\mathscr{A}^*$ does not have to reread portions of the input. Instead, $\mathscr{A}^*$ is able to remember enough information in various "flag bits" to simulate rereading of inputs as needed. The "data structure" used in implementing $\mathscr{A}^*$ is the *spawning tree* which is defined:

(1) There is one node, labelled i, for each active run i. Thus, there are at most N nodes.

(2) If run i has spawned, in order, runs $j_1,\ldots,j_k$ then node i has sons, in order from left to right, $j_k,\ldots,j_1$. (Note: if two or more sons are spawned simultaneously, order them using some fixed convention.)

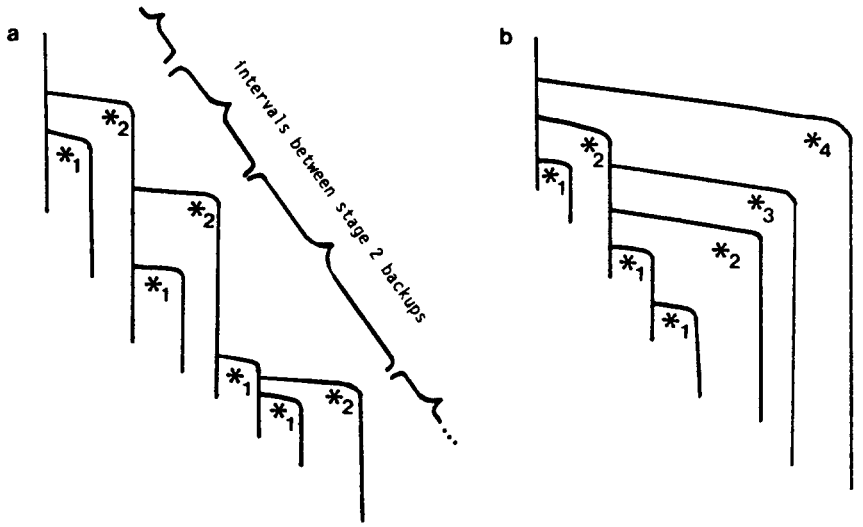

FIG. 2. (a) A path parsed by stage 2 backups; (b) A stage 4 backup. Note nested stage 1, 2, 3 backups.

(3) Each node i is labelled with its *name* as well as

(a) birth$[i]$—a *single bit* $= 1$ iff a P-node has ever been seen along i since its birth.

(b) bstage$[i]$—a $O(\log N\ bit)$ counter $= m$, the maximum of the stage numbers of the backups of h, the father run of i, which have occurred at descendents of the point where i spawned off from h.

(c) backup$[i]$—an *array of N bits*: backup$[i]$ $[k] = 1$ iff a P-node had been seen along i since its last stage k backup.

(d) fbirth$[i]$—a *single bit* $= 1$ iff, at the time i spawns off from its father h, h has seen a P-node since its birth.

(e) fbackup$[i]$—an *array of N bits*: fbackup$[i]$ $[k] = 1$ iff, at the time i spawns off from its father h, h had seen a P-node since its last stage k backup.

(f) state$[i]$—a $O(\log N\ bit)$ counter $= k$ iff the current state associated with run i is state k.

See Fig. 3 for an example of the spawning tree and how it represents active runs. The spawning tree provides all needed information for performing backups, controlling the lights, and associated bookkeeping operations. Moreover, it can be represented using $O(N^2)$ bits.

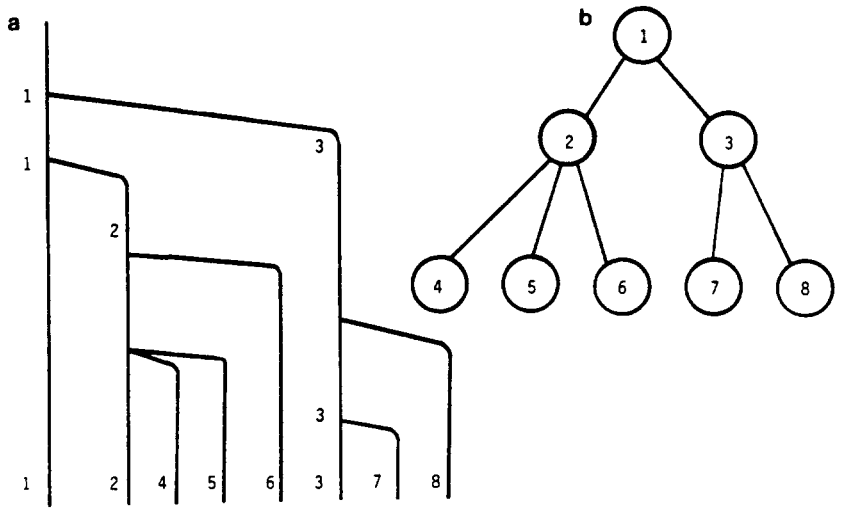

FIG. 3. (a) A run tree and (b) its corresponding spawning tree.

5.4. *Implementation.* The following "pseudo-code" describes the implementation in greater detail:

Flash GREEN on $\langle -, 0\rangle$ pairs with P-nodes:

for each active marker i
  if state$[i]$ is a P-node then flash GREEN on $\langle i, 0\rangle$
    birth$[i] := 1$
    backup$[i] := (1,\ldots, 1)$
end

Read input symbol
Pre-compute successor states of each current state associated with a node of the spawning tree.
In the spawning tree, cross-out all nodes corresponding to markers with no successor.

Backup as needed:

  Repeat the following until all crossed-out nodes are deleted.

    Find a topmost crossed-out node: i

    Pre-order walk the subtree rooted at i to try to find the first non-crossed-out node: j

    if j exist then
      Run j is the "youngest" surviving descendant run of i
      Let i backup and take over run j as described below

```
    if j does not exist then
      delete the entire subtree rooted at i from the spawning tree
      flash RED on ⟨k, 0⟩,..., ⟨k, N − 1⟩ for all k in the subtree
      return all such k to the pool of available markers
End of repeat
(At this point, all remaining runs have ⩾1 successors)
for each active run i
    if i has a single descendant, advance marker i to it
    if i has several descendants s_1,..., s_k then
      assign i to s_1
      assign "free" markers i_2,..., i_k to s_2,..., s_k, respectively
      for each i' ∈ {i_2,..., i_k}
        add i' as a leftmost son of i in the spawning tree
        let bstage[i'] := 0
        let fbackup[i'] := backup[i]
        let fbirth[i'] := birth[i]
      end
end
```

We now describe how to do a backup of run i. Refer to Fig. 4 as needed. Suppose the current node A associated with marker i has no successors, there is a descendant run of i which survives beyond depth(A), and i is not taken over at this depth by a backup of an ancestor run. Let run j be the

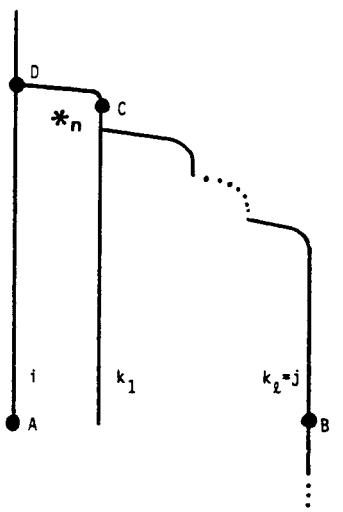

FIG. 4. A stage n backup of run i.

"youngest" (as determined above) descendant run of run i which survives beyond depth(A). Let the sequence of descendant runs of i that are ancestors of j be $i = k_0, k_1, ..., k_l = j$. (Possibly, $l = 1$ so that $k_1 = j$; if $l > 1$ then runs $k_1, ..., k_{l-1}$ dead end at depth(A) just as does run i.) Run i will *take over* run j (as well as runs $k_1, ..., k_{l-1}$) in a *backup of stage* $bs = 1 + \text{bstage}[k_1]$ by performing the actions numbered below.

Note that node B is the current node of run j, node C is the first node of run k_1, and node D is the deepest node of run i which has a descendent node (namely, some immediate successor of B) at a depth greater than depth(A). We say that, for this backup of run i, node A is the *dead point*, node B is the *advance point*, node C is the *backup point*, and node D is the *branch point*. We also say that the backup *occurs* at location node C at time depth(A).

(1) Flash RED on $\langle i, bs-1\rangle$, $\langle i, bs-2\rangle, ..., \langle i, 0\rangle$ since for each $m < bs$, the most recent previous stage m backup of run i has failed in that its backup point does not live on any infinite path.

(2) Flash RED on $\langle k, N-1\rangle, ..., \langle k, 0\rangle$ for each run k whose node is encountered in performing the preorder *walk* from (but not including) i to (and including) j the spawning tree because each such run dies at depth(A). (Each of $k_1, ..., k_l = j$ is such a k but there may be more.)

(3) Flash GREEN on $\langle i, bs\rangle$ iff fbackup$[k_1][bs]$ (iff between the time of the previous stage bs backup of i and this new stage bs backup point, run i has seen a P-node; note that the new stage bs backup point is the first node of run k_1).

(4) For each $m \in [1:l]$, let $t_m := \bigvee_{n \in [2:m]} \text{fbirth}[k_n]$ so that for each such m, $t_m = 1$ iff on the path from where k_m is born back to run i, a P-node occurs. (Note that $t_1 = 0$; for $m > 1$, this path includes exactly the following segments [first node of k_1: last node of k_1 before k_2 splits off] [first node of k_2: last node of k_2 before k_3 splits off ... [first node of k_{m-1}: last node of k_{m-1} before k_m splits off]).

(5) Let run i resume at the current node of the run $j = \text{run } k_l$ which has just been taken over: Flash GREEN on $\langle i, 0\rangle$ iff $t_l \vee \text{birth}[j]$.

(6) We must now adjust birth$[i]$, backup$[i]$ for where run i resumes (the "old" current node of j, node B): birth$[i]$:= fbirth$[k_1] \vee t_l \vee$ birth$[j]$ corresponding to the path, reading backwards, [the current node of $j = k_l$: the first node of $j = k_l$ [the last node of k_{l-1} before k_l splits off: the first node of k_1] [the last node of i before k_1 splits off: the first node of i]

For $n \neq bs$, backup$[i][n]$:= fbackup$[k_1][n] \vee t_l \vee$ birth$[j]$

For $n = bs$, backup$[i][bs] := t_l \vee$ birth$[j]$.

(7) Now i may get some new sons k which were sons of the $k_1, ..., k_l = j$. We must collapse the spawning tree properly to install these

new sons, and for each new son k of i, update fbirth$[k]$, fbackup$[k]$:

for $n := 1$ to l
    add the oldest surviving son of k_n as a son of i
    ⋮
    add the youngest surviving son of k_n as a son of i
end

(When the above loop is done, the oldest group of suns of i will be those that were there originally, still present in their original order. The next oldest group of sons will be those of k_1, with the oldest having been added first, the youngest last. So the youngest son if i will be the youngest surviving son of k_l, provided it exists.)

Delete all the nodes on the *walk* from (but not including) i to (and including) j from the spawning tree. This has collapsed the tree and installed i's new sons k.

To adjust fbirth$[k]$, fbackup$[k]$, where k is a surviving son of k_m, $1 \leqslant m \leqslant l$: fbirth$[k] :=$ fbirth$[k_1] \vee t_m \vee$ fbirth$[k]$ corresponding to the path, reading backwards [the last node of k_m before k is born: the first node of k_m] [the last node of k_{m-1} before k_m is born: the first node of k_1] [the last node of i before k_1 is born: the first node of i].

For $n \neq bs$, fbackup$[k]$ $[n] :=$ fbackup$[k_1]$ $[n] \vee t_m \vee$ fbirth$[k]$

For $n = bs$, fbackup$[k]$ $[bs] := t_m \vee$ fbirth$[k]$

(8) We must ensure that for each son k of i, bstage$[k]$ = the maximum stage of backup of run i, which has occurred at a descendent of the point where k split off from i. If k is an older sibling of k_1 (so k was a son of i present before this backup), let bstage$[k] = \max\{bs,$ bstage$[k]\}$ to reflect the fact that i took over k_1 at a descendant of k via a stage bs backup. If k is son just added to i, let bstage$[k] = 0$ to reflect that no backups of i have yet occurred below where k splits off from the "new, backed up" i.

Remark. The above description provides a template for $\mathscr{A}^*$ to be implemented by a program with $O(N)$ instructions on a RAM (random access machine) of wordlength $O(\log N)$ bits. Since the spawning tree can be represented in $O(N^2)$ bits, $\mathscr{A}^*$ can be realized as a deterministic complemented pairs finite state automaton of size ($\exp(N^2)$ states, N^2 pairs).

5.5. *Correctness*

5.5.1. PROPOSITION. *If a stage n backup of run i occurs then (using the notation of Fig. 4) we have*

(a) *For each $m < n$, a stage m backup of i has previously occurred whose branch point is a descendant node of D.*

(b) *Each backup of run i that has previously occurred whose branch point is a descendant node of D is of stage $m < n$.*

(c) *Moreover, each such branch point lies on no infinite path.*

(d) *For some d, $depth(C) \leqslant d \leqslant depth(A)$, the width of the run tree at depth d is at least $n + 1$.*

Proof. We can argue by induction on n. Recall that, by convention, a stage 0 backup means no actual backup at all. So for $n = 0$, parts (a)–(c) hold vacuously and (d) holds trivially.

Now, suppose a stage $n > 0$ backup occurs. This means run i takes over $k_1, \ldots, k_l = j$ and that $\text{bstage}[k_1] = n - 1$. By the way the algorithm maintains bstage[–], there has been a stage $n - 1$ backup of i whose branch point is a descendant of D. By induction hypothesis, we see that for each $m < n$, there is a stage m backup's branch point at a descendant of D. This establishes (a). The truth of (b) also follows from the way bstage[–] is maintained: if there were previously a stage n or higher backup of i at a descendant of D, then $\text{bstage}[k_1] > n - 1$, a contradiction. To see that (c) is true, note that the algorithm is designed so that, for any backup, its branch point D'' is the deepest ancestor node of its dead point A'' which has any descendant node at depth greater than depth(A''). Finally, to establish (d) note that, by part (a), there is a stage $n - 1$ backup whose branchpoint D' is a descendent of D. By induction hypothesis, there is a d' such that $\text{depth}(C') \leqslant d' \leqslant \text{depth}(A')$ (where C' is the backup point, A' is the dead point of this stage $n - 1$ backup) and the width at depth d' is at least n. Since the path from B up to C does not include any descendent nodes of D' accessed by the time of the stage $n - 1$ backup, $\text{depth}(C) = \text{depth}(D) + 1 \leqslant \text{depth}(D') + 1 = \text{depth}(C')$, and $\text{depth}(A') \leqslant \text{depth}(A)$, we have that $\text{depth}(C) \leqslant d' \leqslant \text{depth}(A)$ and the width at depth d' is at least $n + 1$ (see Fig. 5). ∎

5.5.2. PROPOSITION. *Every infinite run r is eventually assigned a marker i that follows it (allowing for backups) forever. This marker never has to make more than a stage $N - 1$ backup to follow r.*

Proof. Suppose r is an infinite run. After a certain depth, every node on r of greater depth lies on only 1 infinite run, namely r. (If this were not true, the width of the run tree would increase without bound.) Let v_1 be such a node. Now v_1 is assigned a marker i_1. Since v_1 has a unique infinite path (the suffix of r starting at v_1) coming out of it, the only way i_1 will not follow r forever (allowing for backups) is if i_1 is taken over by an ancestor marker i_2. So either i_1 follows r forever after v_1, or i_1 is taken over by an ancestor i_2 at some node v_2. In the latter case, either i_2 follows r forever, or i_2 is taken over by an ancestor i_3 at some node v_3, etc. This process must stop with some ancestor run i_j for $j \leqslant N$ because, otherwise, the width of the

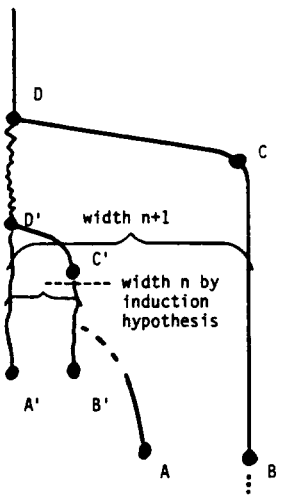

FIGURE 5

tree would exceed N. To see this, note that run i_2 started prior to run i_1 and continued down to depth(v_2) (>depth(v_1)), where it takes over i_1. Similarly, i_3 started prior to i_2 and continues down to depth(v_3) (>depth(v_2)), where it takes over i_2, etc. When i_j takes over i_{j-1} the width at depth(v_1) must be at least j. So the process must stop by i_N (see Fig. 6).

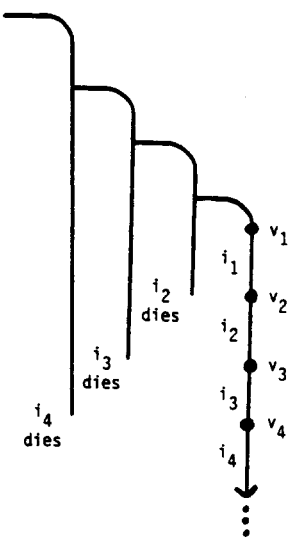

FIGURE 6

To see that a backup of stage $>N-1$ is not required, observe that by part (a) of Proposition 5.5.1, a backup of stage $>N-1$ would imply that the width of the run tree was $>N$. ∎

5.5.3. PROPOSITION. *Suppose that, for run i,*

(1) *at time t there is a stage n backup with backup point C,*

(2) *at time $t' > t$ there is a stage m backup with backup point C',*

(3) *for every backup occurring at time $t'' \in (t: t')$, the backup point C'' is a descendent of C, and*

(4) $m \leqslant n$.

Then C' is a descendent of C.

Proof. Suppose (1), (2), (3) hold. Immediately prior to the time t' backup, run i is a line segment of the form (first node of i): $C: A'$ where A' is the dead point of the Υ' backup. Thus branch point D' is either an ancestor of C or a descendent of C. If D' is an ancestor of C then $m > n$ by the way backup stages are computed. Now if (4) holds so that $m \leqslant n$, then D' (and C') must be a descendent of C. ∎

5.5.4. THEOREM. *For any input x, $\exists$ a run r of $\mathscr{A}_1$ along which $\overset{\infty}{\exists}$ P-nodes iff $\exists$ a pair $\langle i,j\rangle$ of $\mathscr{A}^*$ which flashes* GREEN *i.o. and* RED *f.o..*

Proof. ($\Rightarrow$) By Proposition 5.5.2, any infinite run r in the run tree of $\mathscr{A}_1$ on an input x, will eventually be assigned, by $\mathscr{A}^*$, a marker i, which it keeps forever allowing for backups of i. After that point, we consider run r parsed by the backups of marker i. We have the following cases:

(1) $\overset{\infty}{\exists}$ stage $N-1$ backups of i along r, or

(2) $\neg\overset{\infty}{\exists}$ stage $N-1$ backups of i along r and $\overset{\infty}{\exists}$ stage $N-2$ backups of i along r, or ...

(N) $\neg\overset{\infty}{\exists}$ stage $N-1$ backups, $\neg\overset{\infty}{\exists}$ stage $N-2$ backups,..., and $\neg\overset{\infty}{\exists}$ stage 1 backups of i along r.

If the last case obtains, then there are only finitely many backups of any stage of marker i as it follows the path r. After the last backup, marker i is always pushed forward directly to the next node of r, and $\langle i, 0\rangle$ flashes GREEN every time a new P-node is encountered on r. If there are infinitely many such P-nodes, then plainly $\langle i, 0\rangle$ flashes GREEN i.o.; furthermore, after the last backup, $\langle i, 0\rangle$ will never again flash RED so it flashes RED f.o.

For the other cases, let j be the maximal j' such that $\overset{\infty}{\exists}$ stage j' backups of run i. Then for all $j'' \in (j: N)$, there are only finitely many stage j'' backups of run i. So after some time, there will never again be a RED flash

of the $\langle i,j\rangle$ pair. Consider the suffix of r after that time. It is parsed by the infinitely many stage j backups of i into infinitely many contiguous segments. Infinitely many of these segments will contain P-nodes iff $\overset{\infty}{\exists}$ P-nodes along r. Hence, at infinitely many of the stage j backups, a P-node will be detected in the segment from the previous to the current backup point. Accordingly, the pair $\langle i,j\rangle$ will flash GREEN each such time and hence i.o.

($\Leftarrow$) When $j=0$ we note that if $\langle i,0\rangle$ flashes GREEN i.o., RED f.o., then (by construction of $\mathscr{A}^*$) the marker i never backs up after the last RED flash. So at a certain node, say v, in the run tree, marker i is assigned and is thereafter always pushed forward without backing up. Since there are infinitely many GREEN flashes, (by construction of $\mathscr{A}^*$), there is an infinite path r' starting at v followed by marker i with no backups which has infinitely many P-nodes along it. Since there is a finite path r'' from the root to v, r'' concatenated with r' is the desired infinite run r with infinitely many P-nodes along it.

Otherwise assume $j>0$ and $\langle i,j\rangle$ flashes GREEN i.o., RED f.o.. That there is a last RED flash of $\langle i,j\rangle$ means that there are no more backups taken by marker i of stage $j'>j$. Consider the GREEN flashes occurring after the last RED flash of $\langle i,j\rangle$. For each n, at the nth such GREEN flash of $\langle i,j\rangle$, marker i backs up (via a stage j backup) with a backup point that is some node v_n. After being assigned to node v_n, marker i is never taken over by an ancestor marker i' (because if it were, $\langle i,j\rangle$ would again flash RED). For each n, v_{n+1} is a descendant of v_n (because it is reached from v_n without any backups of stage $j'>j$ and repeatedly applying Proposition 5.5.3) and there is a P-node on the finite path from v_n to v_{n+1}. Let r'' be the finite path from the root to v_1. Then r'' concatenated with $(v_1, v_2, v_3, \ldots)$ is the desired infinite run r along which there are infinitely many P-nodes. ∎

6. Programming the Tree Automata

In Section 3 we argued that a normal form CTL* formula f_1 is satisfiable iff it has an infinite tree-like model, where the branching at each node is bounded by $|f_1|$ and where each Ep_0 subformula is satisfied along a dsignated path. This enables us to reduce the satisfiability problem to the emptiness problem for finite automata on infinite trees: For each subformula Ap_0, $AGEp_0$, or Ep_0, we can build a complemented pairs tree automaton of size at most $(\exp^2(|p_0|)$ states, $\exp(|p_0|)$ pairs). These tree automata can then be combined using a cross product construction to get a complemented pairs tree automaton for f_1 of size $(\exp^2(|f_1|)$ states, $\exp(|f_1|)$ pairs) which accepts an infinite $|f_1|$-ary Σ-tree (where $\Sigma = \text{PowerSet}(\text{AtomicPropositions}(f_1))$) iff it defines a model of f_1 as described above. By the results of

(Streett, 1981) the emptiness problem for a complemented pairs tree automaton with m states and k pairs can be decided in time $\exp^2(k + \log m)$; hence, emptiness of the f_1 automaton is decidable in $\exp^3(|f_1|)$ time.

The tree automaton for an $AGEp_0$ subformula is designed so that it starts up at each node of the tree the nondeterministic Buchi string automaton for p_0 and runs it down the designated path for Ep_0 to ensure that p_0 actually holds along it. (Along the designated path acceptance is determined by the string automaton; along nondesignated paths acceptance occurs unconditionally). The tree automaton for an Ep_0 subformula operates similarly except that the string automaton only needs to be run down the designated path starting at the root of the tree. These tree automata can be implemented in size ($\exp(|p_0|)$ states, $|p_0|$ pairs).

To build the tree automaton for an Ap_0 subformula, we first construct the deterministic complemented pairs string automaton of size ($\exp^2(|p_0|)$ states, $\exp(|p_0|)$ pairs) as described in Section 5 for the linear time subformula p_0. The tree automaton for an Ap_0 subformula is then designed to simply run the deterministic string automaton for p_0 down every path from the root. Since the tree automaton is deterministic, it accepts iff for all paths x in the input tree the deterministic string automaton accepts iff for all paths x in the input tree p_0 holds along x iff Ap_0 holds at the root of the input tree. This tree automaton will be of size ($\exp^2(|p_0|)$ states, $\exp(|p_0|)$ pairs).

Remark. The string automaton for p_0 must be deterministic in order to get the tree automaton for Ap_0. To see this, consider two paths of the tree xy and xz which start off with a common prefix but eventually separate to follow two different infinite suffixes y and z. It is possible that p_0 holds along both paths, but in order for the nondeterministic string automaton to accept, it might have to "guess" while reading a particular symbol of x whether it will eventually read the suffix y or the suffix z. The state it guesses for y is in general different from the state it guesses for z. Consequently, no single run of a tree automaton based on a nondeterministic string automaton can lead to acceptance along all paths.

As a corollary, we have also obtained a small model theorem for CTL* since an automaton accepts an infinite tree iff it accepts a finitely generable tree obtained by "unwinding" a finite tree (Rabin, 1969; Hossley and Rackoff, 1972).

7. Expressiveness Results

We wish to relate the "expressive power" of tree automata with branching time logics. A precise comparison is difficult since (i) the logics can be interpreted over structures which are trees with nodes of infinite outdegree whereas the automata take input trees of fixed, finite outdegree, and (ii) the

tree automata can distinguish between, e.g., the leftmost and the rightmost successor node whereas the logics cannot. To facilitate a comparison, we therefore restrict our attention to (i) structures corresponding to infinite binary trees and (ii) *symmetric* binary tree automata with a transition function $\delta: S \times \Sigma \to \text{PowerSet}(S \times S)$ for which $(t, t') \in \delta(s, a)$ iff $(t', t) \in \delta(s, a)$. We can then show that CTL* augmented with existential quantification over atomic propositions (EQCTL*, for short) is exactly as expressive as symmetric pairs automata on infinite binary trees. Moreover, if we similarly augment UB of (Ben-Ari, Manna, and Pnueli, 1981) (recall that in UB, A or E is paired with a single F, G, or X), the resulting logic (call it EQUB) corresponds to symmetric Buchi automata on infinite binary trees.

An EQCTL* formula is of the form $\exists Q_1 \cdots \exists Q_m f$, where f is a CTL* formula and the Q_i are atomic propositions appearing in it. The semantics is that, given a structure $M = (S, R, L)$, $M, s \models \exists Q_1 \cdots \exists Q_m f$ iff there exists a structure $M' = (S, R, L')$ such that $M', s \models f$, where L' extends L by assigning a truth value to each Q_i in each state of S; EQUB is defined similarly.

7.1. THEOREM. EQCTL* *is exactly as expressive as symmetric pairs automata on infinite binary trees.*

Proof. Given any EQCTL* formula $f_1 = \exists Q_1 \cdots \exists Q_m f(P_1, \ldots, P_n)$ with free atomic propositions $P_1, \ldots, P_n$, we can construct an equivalent formula $g(P_1, \ldots, P_n)$ of $S2S$ with free set variables $P_1, \ldots, P_n$. For example, EFP_1 could be translated into a formula $\exists P(\text{PATH}(P) \wedge \exists x(x \in P \wedge x \in P_1))$, where PATH($P$) abbreviates $\lambda \in P \wedge \forall y \ (y \in P \Rightarrow (yb_0 \in P \vee yb_1 \in P \wedge \neg((yb_0 \in P \wedge yb_1 \in P)))$. By (Rabin, 1969) we can therefore construct a pairs automaton $\mathcal{A}$ which accepts an infinite binary Σ-tree with $\Sigma = \text{PowerSet}(P_1, \ldots, P_n)$ iff f_1 holds at the root of the corresponding structure. Since f_1 does not distinguish between left and right subtrees, we can assume without loss of generality that $\mathcal{A}$ is symmetric, i.e., if $\mathcal{A}$ itself is not symmetric we can obtain an equivalent automaton $\mathcal{A}'$ which is.

Let $\mathcal{A}'$ be the same as $\mathcal{A}$ but with transition function δ' such that $\delta'(s, a) = \{(t, u), (u, t): (t, u) \in \delta(s, a)\}$. Since any run of $\mathcal{A}$ is also a run of $\mathcal{A}'$, if $\mathcal{A}$ accepts an input tree, so does $\mathcal{A}'$. Conversely, suppose there is an accepting run of $\mathcal{A}'$ on an input tree M. M can be viewed as an infinite graph G which has the shape of a binary tree with nodes labelled from Σ and arcs labelled by either b_0 or b_1. By swapping the arc labels below appropriate nodes, we can get a graph G^* which is identical except for the arc labels and which corresponds to an input tree M^* accepted by $\mathcal{A}$. Thus M^* and G^* define a model of f_1. Since f_1 is oblivious to the labels on arcs, G and M also define a model of f_1. Because $\mathcal{A}$ accepts all trees defining models of f_1, $\mathcal{A}$ must also accept M. Thus, $\mathcal{A}$ and $\mathcal{A}'$ accept exactly the same set of trees as desired.

For the converse, let $\mathscr{A}$ be a symmetric pairs automaton on infinite binary trees. For simplicity, we assume that the input alphabet is (or is coded as) $\Sigma = \text{PowerSet}(\{P_1,..., P_n\})$ for some list of atomic propositions $P_1,..., P_n$. We can design an EQCTL* formula which is true at the root of a binary Σ-tree (viewed as a structure in the obvious way) iff $\mathscr{A}$ accepts the tree: Let $\{q_1,..., q_n\}$ be the sate set of $\mathscr{A}$. Associate with each q_i an atomic proposition Q_i. Intuitively, Q_i holds at node s iff $\mathscr{A}$ is in state q_i at s. Any truth assignment to the Q_i defines a candidate run of $\mathscr{A}$ on the input tree. This is an actual run provided all transitions are consistent with the transition function δ of $\mathscr{A}$. We can easily write a formula $\text{run}(Q_1,..., Q_m)$ which ensures such consistency. For example, if $\delta(q_1, \{P_1, P_2\}) = \{(q_2, q_3), (q_3, q_2)\}$ then $AG((Q_1 \wedge P_1 \wedge P_2 \wedge \neg P_3 \wedge \cdots \wedge \neg P_n) \Rightarrow (AX(Q_2 \vee Q_3) \wedge EXQ_2 \wedge EXQ_3))$ is a conjunct of $\text{run}(Q_1,..., Q_m)$.

Now, let the acceptance condition of $\mathscr{A}$ be given by the list $((\text{RED}_1, \text{GREEN}_1),..., (\text{RED}_k, \text{GREEN}_k))$ of pairs of sets of states (i.e., lights). If, for example, $\text{RED}_i = \{q_1, q_2\}$ and $\text{GREEN}_i = \{q_3, q_4\}$ then the assertion that RED_i flashes f.o. and GREEN_i flashes i.o. along a path can be expressed by the path formula $\text{flash}_i = \neg GF(Q_1 \vee Q_2) \wedge GF(Q_3 \vee Q_4)$. Thus, the EQCTL* formula $\exists Q_1 \cdots \exists Q_m\ (\text{run}(Q_1 \cdots Q_m) \wedge A(\text{flash}_1 \vee \cdots \vee \text{flash}_k)$ is equivalent to $\mathscr{A}$. ∎

7.2. THEOREM. EQUB *is exactly as expressive as symmetric Buchi automata on infinite binary trees.*

Proof. Let $f_1 = \exists Q_1 \cdots \exists Q_m f(P_1,..., P_n, Q_1,..., Q_m)$ be an EQUB formula with free propositions $P_1,..., P_n$. Then $f(P_1,..., P_n, Q_1,..., Q_m)$ by itself is a UB formula with free propositions $P_1,..., P_n, Q_1,..., Q_m$. Let $S2S_{1.5}$ be the second order language of two successors with one class of set variables ranging over only finite sets, another class of set variables ranging over infinite sets, and explicit second order quantification allowed only for variables of the first class. We can construct from f an equivalent formula $g(P_1,..., P_n, Q_1,..., Q_m)$ in $S2S_{1.5}$ (where the free variables are of the second class) because quantification over finite sets suffices to express all the modalities of UB (e.g., AFP_1 can be expressed as "there exists a finite subtree all of whose frontier nodes satisfy P_1"). It is known (Rabin, 1983) that for every formula $g(P_1,..., P_n, Q_1,..., Q_m)$ of $S2S_{1.5}$, there is an equivalent Buchi automaton over binary Σ'-trees, where $\Sigma' = \text{PowerSet}(\{P_1,..., P_n, Q_1,..., Q_m\})$. By introducing additional nondeterminism to "guess" the truth assignments to the Q_i, we can obtain from $\mathscr{A}$ a Buchi automaton $\mathscr{B}$ on Σ-trees with $\Sigma = \text{PowerSet}(\{P_1,..., P_n\})$. The automaton $\mathscr{B}$ accepts exactly those trees corresponding to models of $\exists Q_1 \cdots \exists Q_m f(P_1,..., P_n, Q_1,..., Q_m)$. As before, we can assume without loss of generality that $\mathscr{B}$ is symmetric.

The proof of the converse parallels the corresponding part of the proof of

the previous theorem: Let $\mathcal{A}$ be a symmetric Buchi automaton. This formula $\text{run}(Q_1,\ldots,Q_m)$ is actually in UB syntax. To express the acceptance condition, that along every path, there are infinitely many occurences of states in GREEN we can write $AGAF(\vee\{Q_i: q_i \in \text{GREEN}\})$. ∎

8. Conclusions

We have given a triple exponential decision procedure for the full branching time logic CTL* interpreted over R-generable structures. We have also compared the expressive power of some branching time languages derived from CTL* with finite automata on infinite trees. We believe that our results serve to underscore the intimate relationship between systems of temporal logic and finite automata on infinite objects. This relationship was first exploited in (Streett, 1981) to give a decision procedure for PDL with repeat and was further developed in (Wolper, Vardi, and Sistla, 1983). An interesting aspect of our approach here is that by identifying some special structure of the automata derived from the temporal formalism, we could obtain better results than those obtained by relying solely on automata–theoretic techniques (Vardi and Wolper, 1983; Pnueli and Sherman, 1983; Wolper, 1982). Perhaps such special structure will allow similar improvements in decision procedures for other logics. Finally, we note one shortcoming of the automata–theoretic approach as opposed to tableau based methods (cf. Ben-Ari, Manna, and Pnueli, 1981; Emerson and Clarke, 1982; Emerson and Halpern, 1982): it provides little help in constructing an explicit, sound, and complete axiomitization. Indeed, the problem of giving an axiomitization for CTL* interpreted over R-generable structures is still open.

Received February 20, 1984; accepted August 1, 1984

Note added in proof. We also refer the reader to: Gurevich, Y., and Shelah, S. (1984), The Decision Problem for Branching Time Logic, manuscript, (reporting results obtained in principle during the Jerusalem Logic Year 1980–81), which shows decidability for another branching time logic, but does not consider complexity issues.

References

Abrahamson, K. (1980), "Decidability and Expressiveness of Logics of Processes," Ph.D. thesis, Univ. of Washington.

Ben-Ari, M., Manna, Z., and Pnueli, A. (1981), The temporal logic of branching time, *in* "Proc. 8th Ann. ACM Sympos. on Principles of Programming Languages."

Clarke, E. M., and Emerson, E. A. (1981), Design and synthesis of synchronization skeletons using branching time temporal logic, *in* "Proc. IBM Workshop on Logics of Programs," Lecture Notes in Computer Science No. 131, Springer-Verlag, Berlin/New York.

CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. (1982), Automatic verification of finite state concurrent programs: A practical approach, *in* "Proc. Ann. ACM Sympos. Principles of Programming Languages 1983."

EMERSON, E. A., AND CLARKE, E. M. (1982), Using branching time logic to synthesize synchronization skeletons, *Sci. Comput. Programming*, **2** 241–266.

EMERSON, E. A., AND HALPERN, J. Y. (1982), Decision Procedures and Expressiveness in the Temporal Logic of Branching Time, *in* "14th Ann. ACM Sympos. on Theory of Computing."

EMERSON, E. A., AND HALPERN, J. Y. (1983), "Sometimes" and "not never" revisited: On branching versus linear time, *in* "Proc. Ann. ACM Sympos. Principles of Programming Languages."

EMERSON, E. A. (1983), Alternative semantics for temporal logics, *Theoret. Comput. Sci.* **26**, 120–130.

FISCHER, M. J., AND LADNER, R. E. (1979), Propositional Dynamic Logic of Regular Programs, *J. Comput. System Sci.* **18**, 194–211.

GABBAY, D., *et al.* (1980), The temporal analysis of fairness, *in* "7th Ann. ACM Sympos. on Principles of Programming Languages."

HOSSLEY, R., AND RACKOFF, C. (1972), The Emptiness Problem For Automata on Infinite Trees, *in* "Proc. 13th IEEE Sympos. Switching and Automata Theory," 121–124.

LAMPORT, L. (1980), "Sometimes" is sometimes "not never," *in* "7th Ann. ACM Sympos. on Principles of Programming Languages.

MCNAUGHTON, R. (1966), Testing and generating infinite sequences by a finite automaton, *Inform. Contr.* **9**.

MANNA, Z., AND PNUELI, A. (1979), The modal logic of programs, *in* "Proc. 6th Int. Colloq. Automata, Lang. Programming, Lecture Notes in Computer Science No. 71, pp. 385–410, Springer-Verlag, Berlin/New York.

MEYER, A. R. (1974), Weak monadic second order theory of successor is not elementary recursive, "Boston Logic Colloquium," Lecture Notes in Mathematics No. 453, Springer-Verlag, Berlin, New York.

PNUELI, A. (1977), The temporal logic of programs, *in* "Prog. 19th Ann. Sympos. on Found. Comput. Sci."

PNUELI, A. (1981), The temporal logic of concurrent programs, *Theor. Comput. Sci.* **13**, 45–60.

PNUELI, A., AND SHERMAN, R., (1983), personal communication.

RABIN, M. (1969), Decidability of second-order theories and automata on infinite trees, *Trans. Amer. Math. Soc.* **141**, 1–35.

RABIN, M. (1970), "Automata on Infinite Trees and the Synthesis Problem," Hebrew Univ. Tech. Report No. 37.

RABIN, M. (1983), personal communication.

RABIN, M. AND SCOTT, D. (1959), Finite automata and their decision Problems, IBM J. Res. Develop. **3**, 114–125.

STREETT, R. (1981), "Propositional Dynamic Logic of Looping and Converse," Ph.D. thesis, MIT Lab for Computer Science No. TR-263, 1981; revised version, Inform. Contr. **54**, 1982, 121–141.

WOLPER, P. (1982), A translation from full branching time temporal logic to one letter propositional Dynamic logic with looping, unpublished manuscript.

VARDI, M., AND WOLPER, P. (1983), Yet another process logic, *in* "CMU Workshop on Logics of Programs," Springer-Verlag, Berlin/New York.

WOLPER, P., VARDI, M., AND SISTLA, A. (1983), *in* Reasoning about infinite computations, "Proc. 24th IEEE Ann. Sympos. Found. Comput. Sci."

Printed by the St. Catherine Press Ltd., Tempelhof 41, Bruges, Belgium