DECIDING BRANCHING TIME LOGIC

E. Allen EMERSON[1] and A. Prasad SISTLA[2]

1. Computer Sciences Department, University of Texas, Austin, TX 78712
2. Electrical and Computer Engineering Dept., Univ. of Massachusetts, Amherst, MA 01003

Abstract: In this paper we study the full branching time logic (CTL*) in which a path quantifier, either A ("for all paths") or E ("for some path"), prefixes an assertion composed of arbitrary combinations of the usual linear time operators F ("sometime"), G ("always"), X ("nexttime"), and U ("until"). We show that the problem of determining if a CTL* formula is satisfiable in 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. We also compare the expressive power of tree automata with 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 [BMP81], A or E is always paired with a single occurrence of F,G, or 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. [EH83]) not equivalent to any UB formula.

In this paper, we study the full branching time temporal logic CTL* of [EH83] 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 [MP79], [LA80], [GPSS80], [BMP81], [EH82], [CES83], as well as the *Computation Tree Logic* of [CE81]. (It is also closely related to the logic MPL of [AB80]; see below.) We interpret CTL* formulae over *R-generable* models (cf. [EM83]) - i.e., structures generated by a binary relation like those used in [FL79] and [BMP81]. 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 [AB80] 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, [AB80] gives only a nonelementary decision procedure and states that the existence of an elementary procedure is open. Recently, other researchers ([PS83], [VW83]) 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 *nor-*

[1]The first author was partially supported by NSF Grant MCS-8302878.

[2]The second author was partially supported by NSF Grant MCS-8105553.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1984 ACM 0-89791-133-4/84/004/0014 $00.75

mal form for CTL* which is essentially conjunctions and disjunctions 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 nested 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 ([RA69]): 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 [ST81] 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 ([RA69], [HR72]).

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 $\mathcal{A}_1$ on infinite strings (where acceptance is defined by repeating a designated set of states infinitely often) recognizing $\{x: x \models p_0\}$ with $N = \exp(|p_0|)$ states. A seemingly natural next step would be to program the tree automaton to simply run $\mathcal{A}_1$ 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 ([RS59]) 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 construction [McN66] and yields an equivalent deterministic string automaton with a number of states that is double exponential in N. However, we show that $\mathcal{A}_1$ 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.

Lastly, we compare the expressive power of branching time logic with tree automata. We show that 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, the concluding Section 7 gives our expressiveness results.

2. Preliminaries: Definitions and Terminology

2.1 Syntax and Semantics. We inductively define a class of state formulae (true of false of states) and a class of path formulae (true or false of paths):

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 the above rules forms the language CTL*. Other connectives are introduced as abbreviations in the usual way: $p \vee q$ abbreviates $\neg(\neg p \wedge \neg q)$, $p \Rightarrow q$ abbreviates $\neg p \vee q$, Ap abbreviates $\neg E \neg p$, Fp abbreviates *true* U p, Gp abbreviates $\neg F \neg p$, etc. (Note: |p| denote the *length* of p.)

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

S is a nonempty *set of states*,
R is a nonempty, total *binary relation* on S, and
L is a *labelling* which assigns to each state a set of atomic propositions true in the state

A *fullpath* $(a_1,a_2,a_3,...)$ 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 path x, respectively). When M is understood, we write simply $a \models p$ ($x \models p$). We define $\models$ inductively using the convention that $x = (a_1,a_2,a_3,...)$ denotes a path and x^i denotes the suffix path $(a_i,a_{i+1},a_{i+2},...)$:

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 $x \models p$
    $x \models \neg p$ iff not $(x \models p)$
P3. $x \models Xp$ iff $x^2 \models p$
    $x \models (p\ U\ q)$ iff for some $i \geq 1$, $x^i \models q$ and for all $j \geq 1$ [$j \leq i$ 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 (i.e., linear time) formulae.

Note that in determining whether $x \models p_0$ only the truth values of the atomic propositions actually appearing in p_0 matter. We can thus view a fullpath $x = a_1a_2a_3...$

as an infinite string of sets of atomic propositions of p_0 (so each $a_i \in$ PowerSet(AtomicPropositions(p_0))).

2.2 Definitions. The *Fischer-Ladner Closure of p_0, FL(p_0)*, is the least set of formulae such that

(1) $p_0 \in FL(p_0)$
(2) if $p \wedge q \in FL(p_0)$ then $p,q \in FL(p_0)$
(3) if $\neg p \in FL(p_0)$ then $p \in FL(p_0)$
(4) if $(p\ U\ q) \in FL(p_0)$ then $p,q, X(p\ U\ q) \in FL(p_0)$
(5) if $Xp \in FL(p_0)$ then $p \in FL(p_0)$

Note: $|FL(p_0)| = O(|p_0|)$

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

A set $s \subseteq EFL(p_0)$ is *maximal* provided that $\forall p = \neg q \in EFL(p_0)$, at least one of $q,\neg q \in s$. A set $s \subseteq EFL(p_0)$ 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 $\neg q,\neg X(p\ U\ q) \in s$

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

Remark: We use the symbols $\overset{\infty}{\exists}$ and $\overset{\infty}{\forall}$ which are read "there exist infinitely many" and "for all but a finite number", respectively. We also 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 also write *exp(n)* to mean c^n *for some* $c > 1$ and use $exp^2(n)$ to abbreviate *exp(exp(n))*, etc.

2.3 Finite Automata on Infinite Strings and Trees. There is an extensive literature for finite automata on infinite strings and on infinite trees, and the reader is referred to [McN66], [RA69], [RA70], [HR74] as well as [ST81]. For now, we briefly review the following definitions:

A *finite automaton $\mathcal{A}$ 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*, δ: $S \times \Sigma \rightarrow$ 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 $\mathcal{A}$ on infinite input string $x = a_1a_2a_3...$ is an infinite sequence $r = s_0s_1s_2s_3...$ of states such that $\forall i \geq 0$ $\delta(s_i,a_{i+1}) \supseteq \{s_{i+1}\}$. 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 $((RED_1,GREEN_1)$,..., $(RED_k,GREEN_k))$ of pairs of sets of states: 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 $T_n \rightarrow \Sigma$.

A *finite automaton $\mathcal{A}$ 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 $\delta : S \times \Sigma \rightarrow$ PowerSet(S^n). A *run* of $\mathcal{A}$ on Σ-tree ϕ is a function $\rho : T_n \rightarrow S$ such that for all $s \in T_n$ $(\rho(sb_0),...,\rho(sb_{n-1})) \in \delta(\rho(s),\phi(s))$. We say that $\mathcal{A}$ *accepts* input Σ-tree ϕ iff $\exists$ a run ρ of $\mathcal{A}$ on ϕ such that $\forall$ path x starting at the root of T_n, if $r = \rho|x$, the sequence of states $\mathcal{A}$ 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 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 sketch. We first drive negations inward using DeMorgan's laws and dualities such as $\neg Fp \equiv G\neg p$. We then introduce "fresh" atomic propostions for each "deeply" nested subformula of the form Ap or Ep. For example, $f_0 = E(GEFAFP \wedge FAGR)$ 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 $E(GQ_2 \wedge FQ_3) \wedge AG(Q_1 \equiv AFP) \wedge AG(Q_2 \equiv EFQ_1) \wedge AG(Q_3 \equiv AGR) = f_2$. It is easy to see that a model of f_0 defines a model of f_2 by extending the labelling so that Q_1 is true exactly at the states where AFP holds, etc. Conversely, a model of f_2 must be a model of f_0. Note that that $|f_2| = O(|f_0|)$. Finally, we use 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 get f_1, of length about $2 \cdot |f_2| = O(|f_0|)$. □

By carefully unwinding an R-generable model, we can also show

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

4. The Tableau as a Nondeterministic Finite Automaton

The tableau $\mathcal{T}$ for a linear time formula p_0 defines the transition diagram of a nondeterministic finite automaton $\mathcal{A}$ on infinite strings which accepts $\{x: x \models p_0\}$ by letting the arc $u \rightarrow v$ be labelled with AtomicPropositions(v). A run r of $\mathcal{A}$ on input $x = a_1a_2a_3...$ is an infinite sequence $r = s_0s_1s_2s_3...$ of tableau nodes such that $\forall i \geq 0$ $\delta(s_i,a_{i+1}) \supseteq \{s_{i+1}\}$ where δ is the transition function of $\mathcal{A}$. (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 $\forall i \geq 1$ AtomicPropositions(s_i) = AtomicPropositions(a_i). Any run of $\mathcal{A}$ would correspond to a model of p_0 (in that, $\forall i \geq 1$, $x^i \models \{$formulas p: $p \in s_i\}$) except that eventualities might not be fulfilled.

To check fulfillment, we convert $\mathcal{A}$ into an equivalent nondeterministic Buchi automaton, $\mathcal{A}_1$: First, 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. Note that run r of $\mathcal{A}$ 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 $\mathcal{A}_1$ is then obtained from $\mathcal{A}$ by 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 $\exists^\infty$ 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, $\mathcal{A}_1$ 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 $\mathcal{T}$
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 $\mathcal{A}_1$ inherits from the tableau a similar special structure so that, essentially, different runs on the same input cannot merge:

4.2 Theorem. If $r_1=(s_0,s_1,s_2,...)$ and $r_2=(t_0,t_1,t_2,...)$ are two runs of $\mathcal{A}_1$ 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, $\forall j \leq k$ $s_j = t_j$).

Given a Buchi automaton $\mathcal{A}_1$ 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 $\mathcal{A}^*$ of size $(\exp(N^2)$ states, N^2 pairs). Since $\mathcal{A}^*$ is deterministic and $\mathcal{A}^*$ accepts x iff $x \models \neg p_0$, we may view $\mathcal{A}^*$ 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 $\leq N = \exp(|p_0|)$ where the nodes on level i of the DAG represent 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 $i+1^{st}$ 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: $<i,0>,...<i,N-1>$. There are thus a total of N^2 pairs of lights. The need for multiple pairs of lights 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 $j_1 \ldots j_k$. 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 re-cycled 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 <i,0> flash GREEN whenever marker i encountered a P-node and flash RED whenever run i encountered a dead end. See Figure 1. (The RED flashes are needed to ensure that an infinite number of "non-collinear" P-nodes do not cause erroneous acceptance.)

However, there is in general no way for $\mathcal{A}^*$ 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 $\mathcal{A}^*$ 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 Figure 2, $\mathcal{A}^*$ erroneously rejects because both <1,0> and <2,0> 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 *backup* - but as little as possible - by taking over the "youngest" surviving run j which previously spawned off i. For example, in Figure 3 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 $\mathcal{A}^*$ flash RED on pair <2,0>, <2,1> since run 2 is totally obliterated when run 1 takes it over. $\mathcal{A}^*$ also flashes RED on the pair <1,0>. This ensures that $\mathcal{A}^*$ will not falsely accept due to GREEN flashes on <1,0> caused by non-collinear P-nodes detected by run 1 prior to backups. Then, $\mathcal{A}^*$ flashes GREEN on the pair <1,1> 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, $\mathcal{A}^*$ always makes "good" decisions and run 1 never again has to backup. Then pair <1,0> will never again flash RED. It will flash GREEN i.o. iff $\exists^\infty$ P-nodes along the run 1.
(2) $\mathcal{A}^*$ makes infinitely many "poor" decisions so that run 1 backs up i.o. in which case <1,0> flashes RED i.o. Then $\exists^\infty$ P-nodes along run 1 iff $\exists^\infty$ GREEN flashes of <1,1>.

In general, when the width $N \geq 2$, 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 <i,0>.) 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 Figure 4. P-nodes detected by run i on the path between consecutive stage m backup points are recorded via GREEN flashes of <i,m>.

5.3 Implementation. To perform these backups, $\mathcal{A}^*$ does not have to re-read portions of the input. Instead, $\mathcal{A}^*$ is able to remember enough information in various "flag bits" to simulate re-reading of input as needed. The "data structure" used in implementing $\mathcal{A}^*$ is the *spawning tree* which is defined as follows:

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.)
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 Figure 5 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.

The following "pseudo-code" describes the implementation in greater detail:

Flash GREEN on <-,0> pairs with P-nodes:

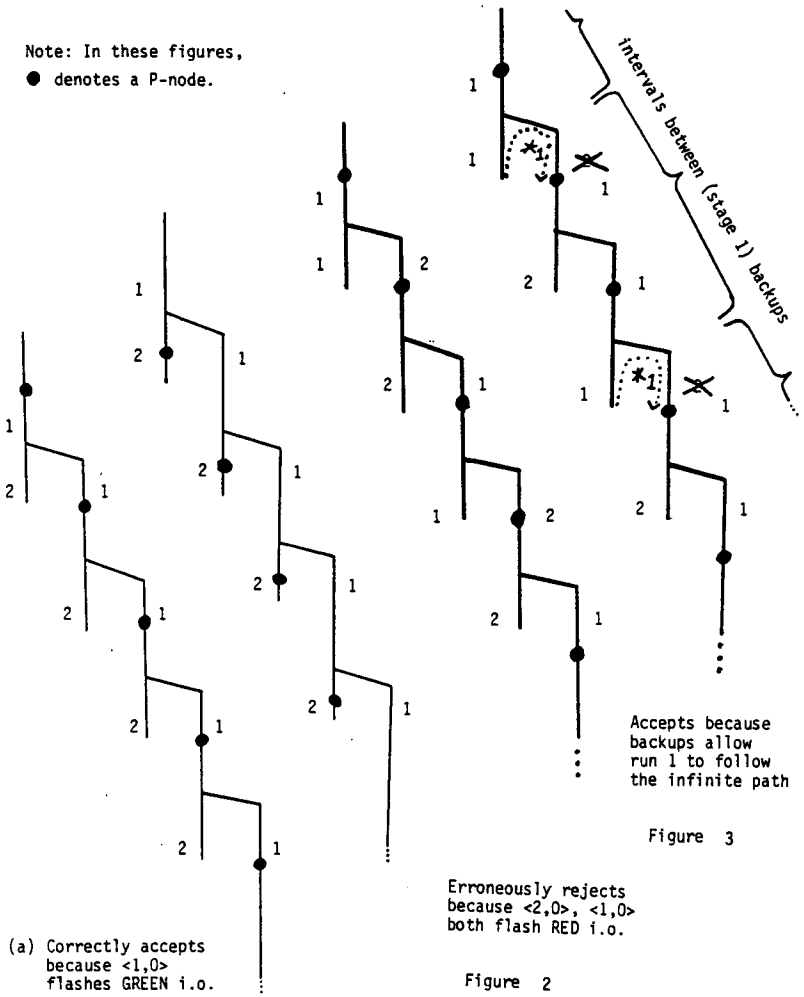
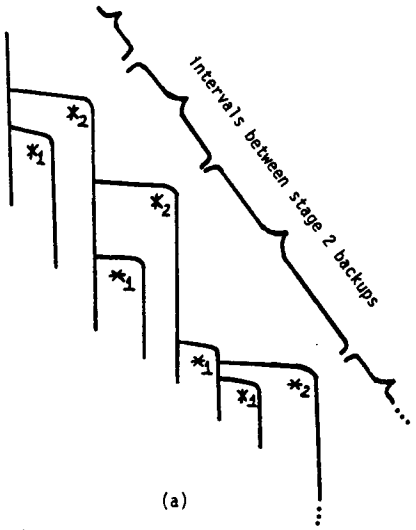
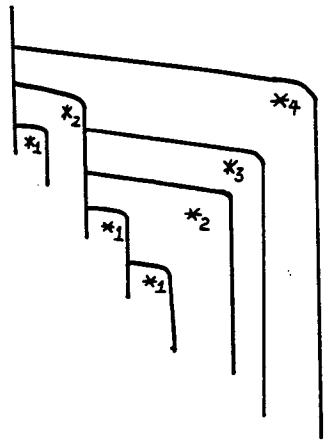


Figure 1

Figure 2

Figure 3





A stage 4 backup. Note nested stage 1, 2, 3 backups.

Figure 4

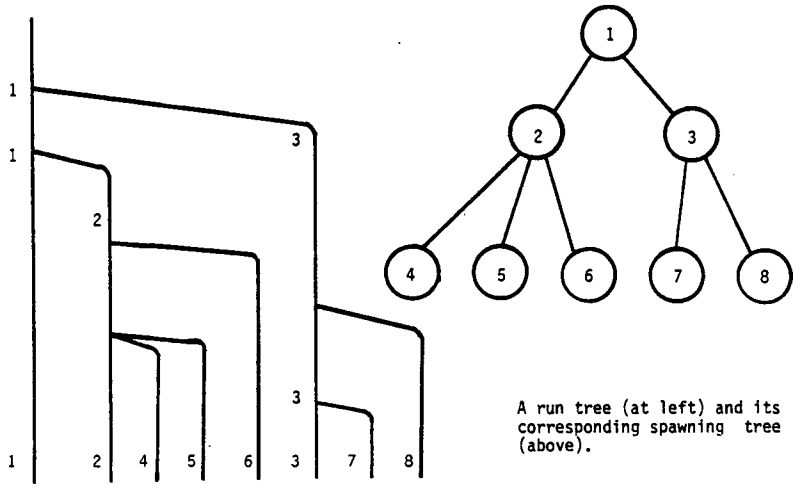


Figure 5

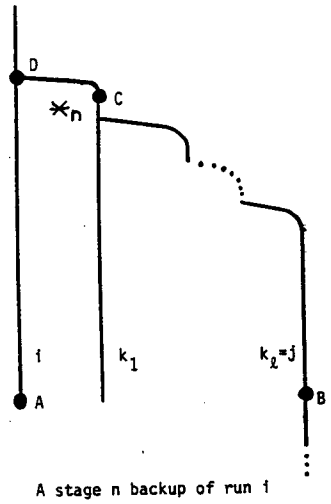


Figure 6

```
for each active marker i
  if state[i] is a  P-node then flash GREEN on <i,0>
                    birth[i] := 1
                    backup[i] := (1,...,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 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 exists then
     Run j is the "youngest" surviving descendant of i
     Let i backup to take over run j as described below
   if j does not exist then
     delete the subtree rooted at i
       from the spawning tree
     flash RED on <k,0>,...,<k,N-1>
       for all k in 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 Figure 6 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 "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* $= 1 +$ 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 <i,bs-1>, <i,bs-2>,...,<i,0> 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 <k,N-1>,...,<k,0> for each run k whose node is encountered in performing the preorder *walk* from (but not including) i to (and including) j in 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 <i,bs> 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 ∈ [1:l], let $t_m := \bigvee_{n\in[2:m]}$ 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 = run k_l which has just been taken over: Flash GREEN on <i,0> iff $t_l \vee$ 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 ≠ 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 sons 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 of 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, $1 \leq m \leq l$:
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 descendents 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 a 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.

5.4 Correctness.

5.4.1 Proposition. If a stage n backup of run i occurs then (using the notation of Figure 6) we have the following:

(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) $\leq$ d $\leq$ depth(A), the width of the run tree at depth d is at least n+1.

5.4.2 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.

5.4.3 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 $\leq$ n

Then C' is a descendent of C.

5.4.4 Theorem. For any input x, $\exists$ a run r of $\mathcal{A}_1$ along which $\exists^{\infty}$ P-nodes iff $\exists$ a pair <i,j> of $\mathcal{A}^*$ 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 $\mathcal{A}_1$ on an input x, will eventually be assigned, by $\mathcal{A}^*$, 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:

$\exists^{\infty}$ stage N-1 backups of i along r or

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

$\neg\exists^{\infty}$ stage N-1 backups, $\neg\exists^{\infty}$ stage N-2 backups,..., and
$\neg\exists^{\infty}$ 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 <i,0> flashes GREEN every time a new P-node is encountered on r. If there are infinitely many such P-nodes, then plainly <i,0> flashes GREEN i.o.; furthermore, after the last backup, <i,0> will never again flash RED so it flashes RED f.o. .

For the other cases, let j be the maximal j' such that $\exists^{\infty}$ 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 <i,j> 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 $\exists^{\infty}$ 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 <i,j> will flash GREEN each such time and hence i.o. .

($\Leftarrow$:) When j=0 we note that if <i,0> flashes GREEN i.o., RED f.o., then (by construction of $\mathcal{A}^*$) 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 the $\mathcal{A}^*$) 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 with infinitely many P-nodes along it.

Otherwise assume j > 0 and <i,j> flashes GREEN i.o., RED f.o. That there is a last RED flash of <i,j> 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 <i,j>. For each n, at the n^{th} such GREEN flash of <i,j>, 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, <i,j> 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 $(v_1,v_2,v_3,...)$ is the desired infinite run along which there are infinitely many P-nodes. □

6. Programming the Tree Automata

We can reduce the satisfiability problem for a normal form CTL* formula f_1 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 Σ = PowerSet(AtomicPropositions(f_1))) iff it defines a model of f_1 as described in section 3. By the results of [ST81] emptiness of the f_1 automaton is decidable in $\exp^3(|f_1|)$ time.

The tree automaton for an $AGEp_0$ subformula (resp., Ep_0 subformula) starts up at each node (resp., the root node) of the tree the 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 and 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 then simply runs 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 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.

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 δ: $S \times \Sigma \rightarrow$ 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 propositons (EQCTL*, for short) is exactly as expressive as symmetric pairs automata on infinite binary trees. Moreover, if we similarly augment UB of [BMP81] (recall that in UB, A or E is paired with a single F, G, X, or U), the resulting logic (call it EQUB) corresponds to symmetric Buchi automata on infinite binary trees.

An EQCTL* formula is of the form $\exists Q_1...\exists Q_m f$ 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), $M,s \models \exists Q_1...\exists Q_m f$ 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 $f_1 = \exists Q_1...\exists Q_m f(P_1,...P_n)$ with free atomic propositions $P_1,...,P_n$, we can construct an equivalent formula $g(P_1,...,P_n)$ of S2S with free set variables $P_1,...,P_n$. For example, EFP_1 could be translated into a formula $\exists P(PATH(P) \wedge \exists x(x \in P \wedge x \in P_1)$ 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 [RA69] we can therefore construct a pairs automaton $\mathcal{A}$ which accepts an infinite binary Σ-tree with Σ = PowerSet($P_1,...,P_n$) 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.

For the converse, let $\mathcal{A}$ be a symmetric pairs automaton on infinite binary trees. For simplicity, we assume that the input alphabet is (or is coded as) Σ = PowerSet($\{P_1,..., P_n\}$) for some list of atomic propostions $P_1,...,P_n$. We can design an EQCTL* formula which is true at the root of a binary Σ-tree iff $\mathcal{A}$ accepts the tree: Let $\{q_1,...,q_n\}$ be the state set of $\mathcal{A}$. Associate with each q_i an atomic proposition Q_i. Intuitively, Q_i holds at node s iff $\mathcal{A}$ is in state q_i at s. Any truth assignment to the Q_i defines a candidate run of $\mathcal{A}$ on the input tree. This is an actual run provided all transitions are consistent with the transition function δ of $\mathcal{A}$. We can easily write a formula

$run(Q_1,...,Q_m)$ which ensures such consistency. For example, if $\delta(q_1,\{P_1,P_2\}) = \{(q_2,q_3), (q_3,q_2)\}$ then AG($(Q_1 \wedge P_1 \wedge P_2 \wedge \neg P_3 \wedge ... \wedge \neg P_n) \Rightarrow (AX(Q_2 \vee Q_3) \wedge EXQ_2 \wedge EXQ_3)$) is a conjunct of $run(Q_1,...,Q_m)$. Now let the acceptance condition of $\mathcal{A}$ be given by the list $((RED_1,GREEN_1),...,(RED_k,GREEN_k))$ of pairs of sets of states (i.e., lights). If, for example, $RED_i = \{q_1,q_2\}$ and $GREEN_i = \{q_3, q_4\}$ 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 $flash_i = \neg GF(Q_1 \vee Q_2) \wedge GF(Q_3 \vee Q_4)$. Thus, the EQCTL* formula $\exists Q_1...\exists Q_m(run(Q_1...Q_m) \wedge A(flash_1 \vee ... \vee flash_k)$ is equivalent to $\mathcal{A}$. □

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

Proof. Let $f_1 = \exists Q_1...\exists Q_m f(P_1,...,P_n,Q_1,...,Q_m)$ be an EQUB formula with free propositions $P_1,...,P_n$. Then $f(P_1,...,P_n,Q_1,...,Q_m)$ by itself is UB formula with free propositons $P_1,...,P_n,Q_1,...,Q_m$. 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 ([RA83]) 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 S'-trees where $\Sigma' = PowerSet(P_1,...,P_n,Q_1,...,Q_m)$. By introducing additional nondeterminism to "guess" the truth assignments to the Q_i, we can obtain from $\mathcal{A}$ a Buchi automaton $\mathcal{B}$ on Σ-trees with $\Sigma = PowerSet(\{P_1,...,P_n\})$. The automaton $\mathcal{B}$ accepts exactly those trees corresponding to models of $\exists Q_1...\exists Q_m f(P_1,...,P_n,Q_1,...,Q_m)$. As before, we can assume without loss of generality that $\mathcal{B}$ 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 $run(Q_1,...,Q_m)$ 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 $AGAF(\vee\{Q_i : q_i \in GREEN\})$. □

8. Bibliography

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

[BMP81] Ben-Ari, M., Manna, Z., and Pnueli, A., The Temporal Logic of Branching Time. 8th Annual ACM Symp. on Principles of Programming Languages, 1981.

[CE81] Clarke, E. M., and Emerson, E. A., Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic, Proceedings of the IBM Workshop on Logics of Programs, Springer-Verlag Lecture Notes in Computer Science #131, 1981.

[CES83] Clarke, E. M., Emerson, E. A., and Sistla, A. P., Automatic Verification of Finite State Concurrent Programs: A Practical Approach, POPL83.

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

[EH82] Emerson, E. A., and Halpern, J. Y., Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. 14th Annual ACM Symp. on Theory of Computing, 1982.

[EH83] Emerson, E. A., and Halpern, J. Y., 'Sometimes' and 'Not Never' Revisited: On Branching versus Linear Time. POPL83.

[EM83] Emerson, E. A., Alternative Semantics for Temporal Logics, Theoretical Computer Science, vol. 26, pp. 121-130, 1983.

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

[GPSS80] Gabbay, D., Pnueli, A., *et al.*, The Temporal Analysis of Fairness. 7th Annual ACM Symp. on Principles of Programming Languages, 1980.

[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.

[LA80] Lamport, L., "Sometimes" is Sometimes "Not Never." 7th Annual ACM Symp. on Principles of Programming Languages, 1980.

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

[MP79] Manna, Z., and Pnueli, A., The modal logic of programs, Proc. 6th Int. Colloquium on Automata, Languages, and Programming, Springer-Verlag Lecture Notes in Computer Science #71, pp. 385-410, 1979.

[ME74] Meyer, A. R., Weak Monadic Second Order Theory of Successor is Not Elementary Recursive, Boston Logic Colloquium, Springer-Verlag Lecture Notes in Mathematics #453, 1974.

[PN77] Pnueli, A., The Temporal Logic of Programs, 19th Annual Symp. on Foundations of Computer Science, 1977.

[PN81] Pnueli, A., The Temporal Logic of Concurrent Programs, Theoretical Computer Science, V13, pp. 45-60, 1981.

[PS83] Pnueli, A. and Sherman, R., Personal Communication, 1983.

[RA69] Rabin, M., Decidability of Second order

Theories and Automata on Infinite Trees, Trans. Amer. Math. Society, vol. 141, pp. 1-35, 1969.

[RA70] Rabin, M., Automata on Infinite Trees and the Synthesis Problem, Hebrew Univ., Tech. Report no. 37, 1970.

[RA83] Rabin, M., personal communication.

[RS59] Rabin, M. and Scott, D., Finite Automata and their Decision Problems, IBM J. Research and Development, vol. 3, pp. 114-125, 1959.

[ST81] Streett, R., Propositional Dynamic Logic of Looping and Converse (PhD Thesis), MIT Lab for Computer Science, TR-263, 1981. (a short version appears in STOC81)

[Wo82] Wolper, P., A Translation from Full Branching Time Temporal Logic to One Letter Propositional Dynamic Logic with Looping, unpublished manuscript, 1982.

[VW83] Vardi, M., and Wolper, P., Yet Another Process Logic, CMU Workshop on Logics of Programs, Springer-Verlag, 1983.