On the Relation of Programs and Computations to Models of Temporal Logic

Pierre Wolper

Université de Liège

Institut Montéfiore, B28, Sart-Tilman, 4000 Liège, Belgium

Abstract

In recent years there has been a proliferation of program verification methods that use temporal logic. These methods differ by the version of temporal logic they are based upon and by the way they use this logic. In this paper we attempt to give a simple, unified and coherent view of the field. For this, we first characterize the models and model generators of different versions of temporal logic using automata theory. From this characterization, we build a classification of verification and synthesis methods that use temporal logic.

1 Introduction

Among the nonclassical logics used in computer science, *temporal logic* has probably been the most successful. Since it was first suggested by Pnueli as a tool for the verification of concurrent programs [Pn77], a large body of work on its theory and applications has developed. Not only has the original use advocated by Pnueli been studied further, but other variants of temporal logic and other ways of using it have been investigated.

The variant of temporal logic initially presented by Pnueli is the linear version [Pn77], [Pn81], [MP81], [MP83c], [Wo83]. In this version, time is viewed as linear which means that each time instant has a unique successor. In other words, the structures over which linear time temporal logic is interpreted are linear sequences. A second common version of temporal logic is branching time temporal logic [BMP81], [EH85a]. In this version, each time instant may have several immediate successors which correspond to different futures, for instance those resulting from nondeterminism in a program. The structures over which branching time temporal logic is interpreted can be viewed as infinite trees. Still further versions of temporal logic have been developed to deal with structures that are partial orders [PW84], [KP87]. The motivation there is to reason about computations that are usefully viewed as partial orders, for instance those of distributed systems.

Interestingly, the two main versions of temporal logic (linear time and branching time) have been used in a variety of ways. The oldest method uses either branching or linear time temporal logic for program verification as follows: the program is described by a set of temporal formulas; the desired property of the program is stated in temporal logic; and an axiomatic system is used to prove that the program formulas imply the desired property [Pn81], [GPSS80], [La80], [OL82], [MP81], [MP83c]. In most cases, this method is built around a first-order version of temporal logic.

The other uses of temporal logic are based on propositional temporal logic. The first of these is synthesis. There, one starts with a formula describing the desired behavior of the program and one synthesizes a program by building a structure that satisfies the formula. This structure which is a set of states and an accessibility relation[1] can then be interpreted as a program. This can be done either with linear time [Wo82], [MW84] or with branching time temporal logic [EC82]. Moreover, the method for building the structure is completely algorithmic and is based on the decision procedure for the propositional temporal logic being considered.

A second use of propositional temporal logic is model checking. Here, instead of building a program from a formula, one checks that a given finite-state program, viewed as a structure over which the formula is interpreted, actually satisfies it. This gives a fully algorithmic verification procedure for finite-state programs. It was first proposed in [CE81] and further developed in [CES86], [EL85b] and [LP85]. It has also been extended to the verification of probabilistic finite-state programs in [Va85a], [PZ86], [VW86b].

The number of variants of temporal logic and the number of ways of using these variants can make the field seem rather confusing to the uninitiated or even to the well-read nonexpert. A thorough understanding of all the issues involved and of the applicability of the various methods and variants of the logic can be difficult to grasp.

In this paper, our (ambitious?) goal is to give a simple, unified and coherent view of the field. Our approach is twofold. First, we study the models and model generators for each variant of temporal logic. The models are the structures satisfying the formulas of the logic. For instance they are linear sequences for linear time temporal logic and trees for branching time temporal logic. Model generators are mathematical structures that represent the set of models of a given formula of the logic. For linear time they are automata on infinite words [Bu62] and for branching time they are automata on infinite trees [Ra69].

Once we have this characterization of the logics, we consider several possible views of programs and their computations. For instance a finite-state program can be viewed as a nondeterministic automaton and its computations as linear sequences. Another possibility is to view the computations of a distributed system as partially ordered sets of states.

[1]a relation which for each state gives the next states

Our description of the applications of temporal logic to program verification is based on the fact that to verify a program one needs two things: a description of the program and a description of the property to be verified. Depending on the type of program considered (e.g. deterministic or nondeterministic), the program and its computations can be different types of structures (e.g. branching or linear). Once this is fixed, we can choose to describe the program either by a formula, a model or a model generator of the adequate logic. A similar choice is made for the property. It is these choices of description formalisms that determine the verification method. For instance, model checking with linear time temporal logic is obtained by describing the program as a model generator of that logic and the desired property of its computations by a formula of the logic. In the branching-time approach to model checking, the program is considered as a model of the logic and the properties of the set of computations of the program are described by a formula of the logic.

With this view of things, each application of temporal logic to verification can be analyzed by answering a few questions: what is a program? what is a computation? how is the program described? how is the desired property described? Once this is known, it is very easy to understand the method at hand. Another advantage is that most algorithms used in this area can be derived from the correspondence between the various types of descriptions of structures: formulas, models and model generators. This approach also extends to synthesis. Of course, in this application, one only needs a description of the desired program in a form that is not directly executable: a formula. The synthesis algorithm then transforms this description in one that is executable: a model or a model generator. Finally, let us note that our approach extends to recent verification work where the properties to be verified are described by an automaton [AS85], [AS87], [MP87].

Even though our coverage of temporal logic and its applications is quite extensive, it is not comprehensive. For instance, we do not deal with interval temporal logic [Mo83], [HMM83], [SMV83], continuous time temporal logic and compositional approaches to temporal verification [BKP84], [BKP85], [BKP86], [NGO85]. Nevertheless, our systematic way of looking at applications of temporal logic could also be useful in these areas.

2 Temporal Logic

In this section, we will describe three versions of temporal logic: linear time temporal logic, branching time temporal logic and the partial order temporal logic of [PW84]. For each of these logics, we will give syntax and semantics and discuss their models and model generators. We limit ourselves to the propositional versions of the logic. For completeness and to show the techniques used to establish them, we give proofs for the results stated in this section. However, in a first reading of the paper these proofs can be skipped.

2.1 Linear Time

We have chosen one of the most common versions of propositional temporal logic appearing in the computer science literature. We will refer to it as *Linear Temporal Logic* (LTL).

2.1.1 Syntax

LTL formulas are built from

- A set P of atomic propositions: p_1, p_2, p_3, ...
- Boolean connectives: $\wedge$, $\neg$.
- Temporal operators: $\bigcirc$ ("next"), U ("until").

The formation rules are:

- An atomic proposition $p \in P$ is a formula.
- If f_1 and f_2 are formulas, so are $f_1 \wedge f_2$, $\neg f_1$, $\bigcirc f_1$, $f_1\, U\, f_2$.

We use $\Diamond f$ ("eventually") as an abbreviation for $(\mathit{true}\, U\, f)$ and $\Box f$ ("always") as an abbreviation for $\neg \Diamond \neg f$. We also use $\vee$ and $\supset$ as the usual abbreviations, and parentheses to resolve ambiguities.

2.1.2 Semantics

A *structure* for an LTL formula (with set P of atomic propositions) is a triple $M = (W, R, \pi)$ where

- W is a finite or enumerable set of states (also called worlds).
- R: $W \rightarrow W$ is a total successor function that for each state gives a unique next state.
- π: $W \rightarrow 2^P$ assigns truth values to the atomic propositions of the language in each state.

For a structure M and a state $w \in W$ we have

- $\langle M, w\rangle \models p$ iff $p \in \pi(w)$, for $p \in P$
- $\langle M, w\rangle \models f_1 \wedge f_2$ iff $\langle M, w\rangle \models f_1$ and $\langle M, w\rangle \models f_2$

- $\langle M, w \rangle \models \neg f$ iff not $\langle M, w \rangle \models f$
- $\langle M, w \rangle \models \circ f$ iff $\langle M, R(w) \rangle \models f$

In the following definitions, we denote by $R^i(w)$ the i^{th} successor of w, that is the i^{th} element in the sequence

$$w,\ R(w),\ R(R(w)),\ R(R(R(w))),\ \ldots$$

- $\langle M, w \rangle \models f_1 \, U \, f_2$ iff for some $i \geq 0$, $\langle M, R^i(w) \rangle \models f_2$ and for all $0 \leq j < i$, $\langle M, R^j(w) \rangle \models f_1$

2.1.3 Models and Model Generators

A structure $M = (W, R, \pi)$ together with a state $w \in W$ constitutes an *interpretation* for LTL formulas. If this interpretation satisfies a formula f ($\langle M, w \rangle \models f$), it is said to be a *model* of f. We want to give a characterization of the models of LTL formulas. To do this, we first need to define a notion of *canonical* model. Indeed, we want to identify models that are not different in any essential way (for instance isomorphic models) and represent them in a standard form. For this, we define a notion of equivalence of interpretations (and hence models) based on bisimulation [Mi80].

Definition 2.1 *Two interpretations $\langle M_1, w_{01} \rangle$ and $\langle M_2, w_{02} \rangle$ where $M_1 = (W_1, R_1, \pi_1)$ and $M_2 = (W_2, R_2, \pi_2)$ are equivalent iff there exists a relation $\phi \subseteq W_1 \times W_2$ such that:*

- *$(w_{01}, w_{02}) \in \phi$;*
- *if $(w_1, w_2) \in \phi$, then $\pi_1(w_1) = \pi_2(w_2)$;*
- *if $(w_1, w_2) \in \phi$, then $(R(w_1), R(w_2)) \in \phi$.*

Intuitively, Definition 2.1 states that two interpretations are equivalent if one can find a relation between their states such that the initial states are in correspondence, the labels of related states are identical and the relation is preserved by the transitions. The following lemma shows the rather obvious fact that equivalent interpretations satisfy the same LTL formulas.

Lemma 2.1 *If two interpretations $\langle M_1, w_{01} \rangle$ and $\langle M_2, w_{02} \rangle$ are equivalent, then for any LTL formula f, $\langle M_1, w_{01} \rangle \models f$ iff $\langle M_2, w_{02} \rangle \models f$.*

Proof: Assume that $M_1 = (W_1, R_1, \pi_1)$ and $M_2 = (W_2, R_2, \pi_2)$. If $\langle M_1, w_{01} \rangle$ and $\langle M_2, w_{02} \rangle$ are equivalent, there is a relation $\phi \subseteq W_1 \times W_2$ satisfying the conditions of Definition 2.1. We show that for any LTL formula f and for every pair of states $(w_1, w_2) \in \phi$, $\langle M_1, w_1\rangle \models f$ iff $\langle M_2, w_2\rangle \models f$. The proof is by induction on the structure of f. The case where f is an atomic proposition follows directly from the fact that $\pi_1(w_1) = \pi_2(w_2)$. The inductive cases are also straightforward. In the case of the operator U, notice that if $(w_1, w_2) \in \phi$, then for all i, $(R_1^i(w_1), R_2^i(w_2)) \in \phi$. ∎

We can now establish the existence of canonical interpretations for LTL formulas. Canonical interpretations for LTL formulas are of the form $\langle M, 0\rangle$, where $M = (N, succ, \pi)$, N being the set of natural numbers and *succ* the successor function on the natural numbers (i.e., $succ(i) = i + 1$ for all $i \in N$). As in a canonical interpretation the set of states, the successor function and the initial state are all fixed, such a model is characterized by a function $\pi : N \rightarrow 2^P$. We now establish that every interpretation for an LTL formula is equivalent to a canonical interpretation.

Lemma 2.2 *Every interpretation $\langle M, w_0\rangle$ where $M = (W, R, \pi)$ is equivalent to a canonical interpretation $\pi' : N \rightarrow 2^P$.*

Proof: The canonical interpretation π' is defined by $\pi'(i) = \pi(R^i(w_0))$, for all $i \geq 0$. To check that the canonical interpretation π' is equivalent to $\langle M, w_0\rangle$, consider the relation ϕ defined by $\phi = \{(i, R^i(w_0)) | i \geq 0\}$. ∎

Basically, to obtain a canonical interpretation from an arbitrary interpretation, one simply *unwinds* the interpretation into an infinite sequence. Now that we have established the existence of canonical interpretations which are mappings from the natural numbers to the set 2^P, we will only consider these. Doing this is not restrictive as every other interpretation is equivalent in the strong sense of Definition 2.1 to a canonical interpretation.

Our goal is to give a description of the set of canonical models of an LTL formula. It is this description that we will call the *model generator* for an LTL formula. The model generators for LTL formulas are automata on infinite words. Before defining these automata, let us note that a canonical interpretation π can be viewed as an infinite word $w = a_1 a_2 a_3 \ldots$ over the alphabet 2^P, where $a_i = \pi(i - 1)$, $i \geq 1$

The type of finite automata on infinite words we consider is the one defined by Büchi [Bu62]. A *Büchi sequential automaton* is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- Σ is an alphabet,
- S is a set of states,
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function,
- $S_0 \subseteq S$ is a set of starting states, and
- $F \subseteq S$ is a set of designated states.

A *run* of A over an infinite word $w = a_1a_2\dots$, is an infinite sequence $s_0, s_1, \dots$, where $s_0 \in S_0$ and $s_i \in \rho(s_{i-1}, a_i)$, for all $i \geq 1$. A run $s_0, s_1, \dots$ is *accepting* if there is some designated state that repeats infinitely often, i.e., for some $s \in F$ there are infinitely many i's such that $s_i = s$. The infinite word w is *accepted* by A if there is an accepting run of A over w. The set of denumerable words accepted by A is denoted $L(A)$.

Example 2.1 Consider the automaton of Figure 1 where $F = \{s_1\}$ and $S_0 = \{s_0\}$. The language

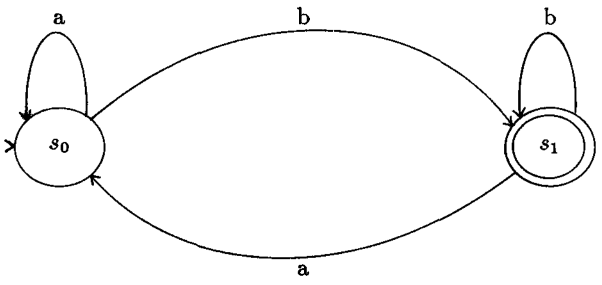

Figure 1: A Büchi sequential automaton

it accepts consists of all words of the form $(a^*bb^*)^\omega$, where $*$ denotes finite repetition and ω denotes infinite repetition. Note that the words of the form $(a^*bb^*)^*a^\omega$ are not accepted by this automaton. Intuitively, a word is accepted if it contains b infinitely often. ∎

We can now give the result relating LTL and Büchi automata. We show that given an LTL formula, we can build a Büchi sequential automaton that accepts exactly the canonical models of that formula. In the statement of the following theorem, we use $|f|$ to denote the length of a formula f, i.e., the number of symbols (propositions or connectives) it contains.

Theorem 2.3 ([WVS83]) *Given an LTL formula f, one can build a Büchi sequential automaton $A_f = (\Sigma, S, \rho, S_0, F)$, where $\Sigma = 2^P$ and $|S| \leq 2^{O(|f|)}$, such that $L(A_f)$ is exactly the set of sequences satisfying the formula f.*

Proof: The construction uses the notion of the *closure* of an LTL formula f, denoted $cl(f)$. For a formula f, $cl(f)$ consists of all the subformulas of f and their negation, where we identify $\neg\neg f_1$ with f_1. Precisely, $cl(f)$ is defined as follows:

- $f \in cl(f)$

- $f_1 \wedge f_2 \in cl(f) \rightarrow f_1, f_2 \in cl(f)$
- $\neg f_2 \in cl(f) \rightarrow f_2 \in cl(f)$
- $f_2 \in cl(f) \rightarrow \neg f_2 \in cl(f)$
- $\bigcirc f_2 \in cl(f) \rightarrow f_2 \in cl(f)$
- $f_1 \, U \, f_2 \in cl(f) \rightarrow f_1, f_2 \in cl(f)$.

Note that we have $|cl(f)| \leq 2|f|$. To build a Büchi automaton accepting the models of a formula f, we first build an automaton over the alphabet $2^{cl(f)}$. This automaton will recognize the set of words that are obtained from models of f by extending the label of each state with the elements of $cl(f)$ true in that state.

The Büchi automaton we build is the combination of two automata: the *local automaton* and the *eventuality automaton.* The local automaton checks for "local inconsistencies" in the model. More precisely, it checks that there are no propositional inconsistencies and that the temporal operators are locally satisfied. For instance, it checks that if $f_1 \, U \, f_2$ is true in a state, then either f_2 is true in that state or f_1 is true there and also $f_1 \, U \, f_2$ is true in the next state.

The local checking is sufficient except for *eventuality formulas.* These are the formulas of the form $f_1 \, U \, f_2$. The problem with these formulas, is that the local conditions imposed do not guaranty that a point where f_2 is true is indeed eventually reached. Checking this will be the role of the eventuality automaton.

Constructing the Local Automaton

The local automaton is $L = (2^{cl(f)}, N_L, \rho_L, N_f, N_L)$. The state set N_L is the set of all sets $\mathbf{s}$ of formulas in $cl(f)$ that do not have any propositional inconsistency. Namely they must satisfy the following conditions:

- for all $f_1 \in cl(f)$, we have that $f_1 \in \mathbf{s}$ iff $\neg f_1 \notin \mathbf{s}$.
- for all $f_1 \wedge f_2 \in cl(f)$, we have that $f_1 \wedge f_2 \in \mathbf{s}$ iff $f_1 \in \mathbf{s}$ and $f_2 \in \mathbf{s}$.

For the transition relation ρ_L, we have that $\mathbf{t} \in \rho_L(\mathbf{s}, a)$ iff $a = \mathbf{s}$ (i.e., all transitions leaving a state have the same label as that state) and:

- for all $\bigcirc f_1 \in cl(f)$, we have that $\bigcirc f_1 \in \mathbf{s}$ iff $f_1 \in \mathbf{t}$, and
- for all $f_1 \, U \, f_2 \in cl(f)$, we have that $f_1 \, U \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or both $f_1 \in \mathbf{s}$ and $f_1 \, U \, f_2 \in \mathbf{t}$.

Finally, the set of starting states N_f consists of all sets $\mathbf{s}$ such that $f \in \mathbf{s}$. The local automaton does not impose any acceptance conditions and so its set of designated states is the whole set of states.

The Eventuality Automaton

Given an LTL formula f, we define the set $e(f)$ of its eventualities as the subset of $cl(f)$ that contains all formulas of the form $f_1\, U\, f_2$. The eventuality automaton is $E = (2^{cl(f)}, 2^{e(f)}, \rho_E, \{\emptyset\}, \{\emptyset\})$, where for the transition relation ρ_E, we have that $\mathbf{t} \in \rho_E(\mathbf{s}, a)$ iff:

- $\mathbf{s} = \emptyset$ and for all $f_1\, U\, f_2 \in a$, we have that $f_1\, U\, f_2 \in \mathbf{t}$ iff $f_2 \notin a$.
- $\mathbf{s} \neq \emptyset$ and for all $f_1\, U\, f_2 \in \mathbf{s}$, we have that $f_1\, U\, f_2 \in \mathbf{t}$ iff $f_2 \notin a$.

Intuitively, the eventuality automaton tries to satisfy the eventualities in the model. When the current state is $\emptyset$, it looks at the model to see which eventualities have to be satisfied. Thereafter, the current state says which eventualities have yet to be satisfied. Note that it is sufficient to only check periodically what eventualities have to be satisfied in the model. Indeed eventualities that are not satisfied in a given state of the model also appear in the next state due to the conditions imposed by the local automaton. We often describe this by saying that eventualities are *propagated.*

Combining the Automata

We now combine the local and eventuality automata to get the *model automaton.* The model automaton $M = (2^{cl(f)}, N_M, \rho_M, N_{M0}, F_M)$ is obtained by taking the cross product of L and E. Its sets of states is $N_M = N_L \times 2^{e(f)}$. The transition relation ρ_M is defined as follows: $(\mathbf{p}, \mathbf{q}) \in \rho_M((\mathbf{s}, \mathbf{t}), a)$ iff $\mathbf{p} \in \rho_L(\mathbf{s}, a)$ and $\mathbf{q} \in \rho_E(\mathbf{t}, a)$. The set of starting states is $N_{M0} = N_f \times \{\emptyset\}$, and the set of designated states is $F_M = N_L \times \{\emptyset\}$. Note that $|N_M| \leq 2^{|cl(f)|} \times 2^{|e(f)|} \leq 2^{3|f|}$

The automaton we have constructed, accepts words over $2^{cl(f)}$. However, the models of f are defined by words over 2^P. So, the last step of our construction is to take the projection of our automaton on 2^P. This is done by mapping each element $b \in 2^{cl(f)}$ into $b \cap P$. ∎

The construction we have given for the model generator of LTL formulas is one of the simplest to describe. However, it leads to an automaton that can contain numerous unnecessary states. It is possible to make the construction less wasteful essentially by constructing only the states that are reachable from the initial state. Note however that this does not influence the worst case complexity of the algorithm.

Example 2.2 The Büchi automaton of Figure 1 is a model generator for the formula $\Box \Diamond p$ when a is taken to be $\emptyset$ and b to be $\{p\}$. Note that the automaton obtained from the construction given in the proof of Theorem 2.3 is somewhat more complicated, though it is equivalent to (generates the same infinite words as) the automaton we have described. ∎

The construction we have given can be extended to most versions of linear time propositional temporal logic, such as the extended temporal logic defined in [Wo83], [WVS83], [VW88] and the temporal logic with past operators of [LPZ85]. The inverse construction (building a temporal formula from a Büchi automaton) is possible if one uses extended temporal logic [Wo83], [WVS83], [VW88].

Theorem 2.3 makes the theory of Büchi automata very relevant to temporal logic. The following theorem states important results on the *nonemptiness problem* for Büchi sequential automata, i.e., the problem of determining for a given Büchi sequential automaton A whether $L(A)$ is nonempty.

Theorem 2.4 *(1) [SVW87] The nonemptiness problem for Büchi sequential automata is solvable in nondeterministic logspace.*

(2) [EL85a,EL85b] The nonemptiness problem for Büchi sequential automata is solvable in linear time.

One can easily obtain a decision procedure for LTL by combining Theorems 2.3 and 2.4(1). Given a formula f, one builds the automaton accepting the models of this formula (using Theorem 2.3) and then checks whether this automaton is nonempty (using Theorem 2.4(1)). By combining the construction of the automaton with the algorithm checking for nonemptiness, it is possible to obtain a PSPACE upper bound for the decidability of LTL (see [SVW87], [VW88]). The PSPACE upper bound for LTL was originally proven in [SC82] and [SC85] where a matching lower bound is also established.

2.2 Branching Time

What distinguishes *Branching Temporal Logic* (BTL) from LTL is its ability to specify whether a property is true of some computations or of all computations in a set. Syntactically, it is essentially LTL with two *path quantifiers* ($\forall$ and $\exists$) that indicate to which computations (paths) the LTL formulas apply. Semantically, these paths will be extracted form a branching structure, i.e., a set of states and an accessibility relation such that each state may have several successors. There are several variants of branching temporal logic. The main distinction between these is the type of LTL formula that can appear in the scope of a path quantifier. We will consider two versions of branching temporal logic: the logics CTL and CTL* defined in [EH85a], [EH85b]. In CTL (*Computation Tree Logic*), the formulas appearing in the scope of path quantifiers are restricted to be a single temporal operator. In CTL*, they can be arbitrary LTL formulas. When we do not need to distinguish between versions of branching time temporal logic, we will use the acronym BTL.

2.2.1 Syntax

In defining the syntax of CTL and CTL*, one distinguishes between state and path formulas. Path formulas will be interpreted over linear sequences of states extracted from the branching structure. In contrast, state formulas will be interpreted in a state of the branching structure. They will be either formulas that can be interpreted by considering just one state (e.g., a boolean combination of atomic propositions) or build form path formulas to which a path quantifier ($\forall$ or $\exists$) is applied. Intuitively a state formula $\forall g$, where g is a path formula is true in a state if g is true on all paths leaving that state. The CTL and CTL* formulas built from a set of atomic propositions P are defined as follows:

State Formulas

- An atomic proposition $p \in P$ is a state formula.
- If f_1 and f_2 are state formulas, so are $f_1 \wedge f_2$ and $\neg f_1$.
- If g is a path formula, then $\exists g$ and $\forall g$ are state formulas.

Path Formulas (CTL)

- If g_1 and g_2 are state formulas, then $\bigcirc g_1$, and $g_1 \, U \, g_2$ are path formulas.

Path Formulas (CTL*)

- A state formula is a path formula.
- If g_1 and g_2 are path formulas, so are $g_1 \wedge g_2$ and $\neg g_1$.
- If g_1 and g_2 are path formulas, then $\bigcirc g_1$ and $g_1 \, U \, g_2$ are path formulas.

Finally, the formulas of CTL and CTL* are their respective state formulas. Intuitively, the difference between the two logics is that in CTL*, a path formula can be any boolean combination or nesting of temporal operators applied to state formulas. By contrats, in CTL, path formulas are a single temporal operator applied to a state formula. In CTL*, we will use $\Box$ and $\Diamond$ as the abbreviations defined in LTL. In CTL, we will use $\exists \Diamond f$ and $\forall \Diamond f$ to abbreviate $\exists(\mathit{true} \, U \, f)$ and $\forall(\mathit{true} \, U \, f)$ respectively. Also, $\exists \Box f$ will be an abbreviation for $\neg \forall \Diamond \neg f$ and $\forall \Box f$ for $\neg \exists \Diamond \neg f$.

Example 2.3 $\forall\Box\Diamond p$ and $\exists(\Box\Diamond p \supset \Box\Diamond q)$ are CTL* formulas, but not CTL formulas. Indeed, the path formula in $\forall\Box\Diamond p$ is $\Box\Diamond p$ and thus contains nested temporal operators. On the other hand, $\forall\Box\forall\Diamond p$ is a CTL formula. In this case, we have a single temporal operator ($\Box$) applied to a state formula ($\forall\Diamond p$). ∎

2.2.2 Semantics

A *structure* for a BTL (CTL or CTL*) formula (with set P of atomic propositions) is a triple $M = (W, Q, \pi)$ where

- W is a finite or enumerable set of states (also called worlds).
- $Q \subset W \times W$ is a total accessibility relation that for each state gives a nonempty set of successors.
- π: $W \rightarrow 2^P$ assigns truth values to the atomic propositions in each state.

Note that the only difference between a structure for LTL and a structure for BTL is that we have an accessibility relation Q rather than a successor function R. This means that each state can have several immediate successors rather than a unique one.

To give the semantics of a BTL formula for an interpretation $\langle M, w\rangle$, we need to define the notion of a path (or linear interpretation) extracted from a branching interpretation. Intuitively, one extracts a linear interpretation from a branching one by choosing one specific successor for each state. Formally, we have the following:

Definition 2.2 *A linear interpretation extracted from a branching interpretation $\langle M_B, w_0\rangle$ where $M_B = (W_B, Q_B, \pi_B)$ is a pair $\langle M_L, w_0\rangle$ where $M_L = (W_L, R_L, \pi_L)$ with $w_0 \in W_L \subseteq W_B$, $\pi_L(w) = \pi_B(w)$ and $(w, R_L(w)) \in Q_B$ for all $w \in W_L$.*

We can now give the semantics of BTL. For path formulas, the semantics are identical to those of LTL except that we interpret path formulas that are also state formulas according to the semantics of state formulas.

For state formulas, we define the truth of a formula g in an interpretation $\langle M_B, w_0\rangle$ by the following clauses:

- $\langle M_B, w_0\rangle \models p$ iff $p \in \pi(w)$, for $p \in P$
- $\langle M_B, w_0\rangle \models g_1 \wedge g_2$ iff $\langle M_B, w_0\rangle \models g_1$ and $\langle M_B, w_0\rangle \models g_2$

- $\langle M_B, w_0 \rangle \models \neg g$ iff not $\langle M_B, w_0 \rangle \models g$
- $\langle M_B, w_0 \rangle \models \forall g$ iff for all linear structures $\langle M_L, w_0 \rangle$ extracted from $\langle M_B, w_0 \rangle$, $\langle M_L, w_0 \rangle \models g$
- $\langle M_B, w_0 \rangle \models \exists g$ iff for some linear structure $\langle M_L, w_0 \rangle$ extracted from $\langle M_B, w_0 \rangle$, $\langle M_L, w_0 \rangle \models g$

Example 2.4 The state w_0 of the interpretation described in Figure 2 satisfies the CTL formula $\exists \Box \exists \Diamond p$, but does not satisfy the CTL* formula $\exists \Box \Diamond p$. Indeed, form that state there is no single path on which p is true infinitely often. On the other hand, there exist a path (the one going infinitely through w_0) form all points of which there is a path containing a state where p is true. In the interpretation of Figure 3, the state w_0 satisfies both the CTL formula $\forall \Box \forall \Diamond p$ and the CTL* formula $\forall \Box \Diamond p$. ∎

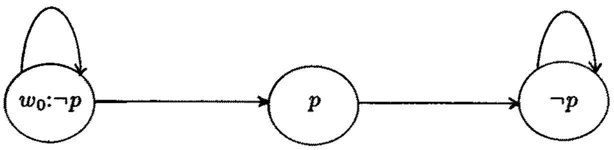

Figure 2: A model of $\exists \Box \exists \Diamond p$

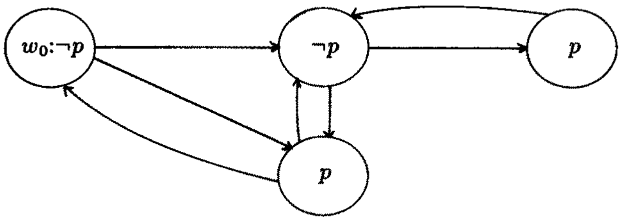

Figure 3: A model of $\forall \Box \Diamond p$

2.2.3 Models and Model Generators

As we did in the linear time case, we now consider the problem of constructing model generators for BTL. Again, the model generators will only deal with canonical models which are here infinite trees. To justify our notion of canonical model, we use a notion of equivalence of interpretations similar to the one we used for linear interpretations.

Definition 2.3 *Two branching interpretations $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ where $M_1 = (W_1, Q_1, \pi_1)$ and $M_2 = (W_2, Q_2, \pi_2)$ are equivalent iff there is relation $\phi \subseteq W_1 \times W_2$ such that:*

- *$(w_{01}, w_{02}) \in \phi$;*
- *if $(w_1, w_2) \in \phi$, then $\pi_1(w_1) = \pi_2(w_2)$;*
- *if $(w_1, w_2) \in \phi$, and there is a $w'_1 \in W_1$ such that $(w_1, w'_1) \in Q_1$ then there is a $w'_2 \in W_2$ such that $(w_2, w'_2) \in Q_2$ and $(w'_1, w'_2) \in \phi$;*
- *if $(w_1, w_2) \in \phi$, and there is a $w'_2 \in W_2$ such that $(w_2, w'_2) \in Q_2$ then there is a $w'_1 \in W_1$ such that $(w_1, w'_1) \in Q_1$ and $(w'_1, w'_2) \in \phi$.*

The notion of equivalence given by Definition 2.3 is essentially Milner's bisimulation [Mi80]. It is also related to the definition given for different purposes in [CGB86]. As in the linear case, we can easily establish that equivalent branching structures satisfy the same BTL formulas.

Lemma 2.5 *If two branching interpretations $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ are equivalent, then for any BTL formula f, $\langle M_1, w_{01}\rangle \models f$ iff $\langle M_2, w_{02}\rangle \models f$.*

Proof: Assume that $M_1 = (W_1, Q_1, \pi_1)$ and $M_2 = (W_2, Q_2, \pi_2)$. If $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ are equivalent, there is a relation $\phi \subseteq W_1 \times W_2$ satisfying the conditions of Definition 2.3. We show for any BTL state formula f and for every pair of states $(w_1, w_2) \in \phi$, $\langle M_1, w_1\rangle \models f$ iff $\langle M_2, w_2\rangle \models f$. The proof is by induction on the structure of f. The interesting case is when f is of the form $\forall g$ or $\exists g$. For this it is sufficient to observe that if one can extract a linear interpretation from one of two equivalent branching interpretations, then one can extract an equivalent (in the sense of Definition 2.1) linear interpretation from the other. ∎

For BTL, showing that every model is equivalent to a canonical model is somewhat more complicated than for LTL. Intuitively, we want to do the same as we did for linear structures and unwind arbitrary models into infinite trees. The problem is that it is not possible to give an a priori bound on the branching factor (the maximum number of immediate successors of each node) of the tree. Indeed, in a branching structure a node can have an arbitrary number of successors and it is not always possible to find an equivalent model where the branching factor is smaller. The best we can do if we choose k-ary infinite trees as canonical models is to show that there is a k-ary tree canonical model equivalent to any model whose branching factor is at most k. As we will see below, this is indeed possible. First, we need to give some definitions concerning k-ary trees.

To define k-ary trees, we use $[k]$ to denote the set $\{1, \ldots, k\}$. The set $[k]^*$ then denotes the set of nodes of the tree, the root being the empty string λ. The successors of a node x are thus the nodes

xi, $i \in [k]$. A node x is said to be of depth i if the length of x is equal to i. A k-ary infinite tree T labeled by the elements of an alphabet Σ is then a function $T : [k]^\star \to \Sigma$. A k-ary tree canonical interpretation for BTL is of the form $\langle M, \lambda \rangle$, where $M = ([k]^\star, succ, \pi)$, the relation *succ* being defined by $succ = \{(x, xi) | x \in [k]^\star, i \in [k]\}$. As in these interpretations, the set of states, the initial state and the successor relation are all fixed, they reduce to a function $\pi : [k]^\star \to 2^P$, that is, to a labeled tree over the alphabet 2^P. We now show how to construct k-ary canonical models from models whose branching factor is $\leq k$ (by definition, the branching factor of a structure (W, Q, π) is the largest ℓ such that for some $w \in W$, we have $\ell = |\{w' | (w, w') \in Q\}|$).

Lemma 2.6 *Every branching interpretation $\langle M, w_0 \rangle$ where $M = (W, Q, \pi)$ and whose branching factor is $\leq k$ is equivalent to a k-ary tree canonical interpretation $\pi' : [k]^\star \to 2^P$.*

Proof: To define the canonical interpretation π', we first define the relation ϕ that will enable us to establish that π' is equivalent to $\langle M, w_0 \rangle$. The relation ϕ is actually a function in $[k]^\star \to W$. We define it in stages. At stage i, ϕ_i will be defined on all nodes of the tree of depth $\leq i$. We start with $\phi_0(\lambda) = w_0$. We obtain ϕ_{i+1} from ϕ_i by extending its definition to all nodes of the tree of depth $i+1$ as follows. For each node $x \in [k]^\star$ of depth i, let $\{w_1, \ldots, w_\ell\}$, $\ell \leq k$ be the successors of $\phi_i(x)$. We define $\phi_{i+1}(xj) = w_j$ for $1 \leq j \leq \ell$ and $\phi_{i+1}(xj) = w_\ell$ for $\ell < j \leq k$. In other words the first ℓ successors of x simulate the ℓ successors of $\phi_i(x)$ and if $\ell < k$, we simply replicate the last successor. Finally, the relation ϕ is the limit ϕ_ω of the relations ϕ_i, and $\pi'(x) = \pi(\phi(x))$ for all $x \in [k]^\star$. With this definition of π', it is quite straightforward to show that ϕ satisfies all the conditions of Definition 2.3. ∎

Lemma 2.6 tells us that if a formula has models of branching factor at most k then these models are equivalent to k-ary canonical models. The next natural step would thus be to describe a model generator for the k-ary canonical models of BTL formulas. However, for the model generator to be interesting, we need to choose k large enough for the set of k-ary canonical models to be nonempty if the formula is satisfiable. Fortunately, this is always possible. The next lemma shows that any satisfiable BTL formula that contains n path quantifiers, has at least one $n+1$-ary canonical model. The idea of the proof of that lemma is that one only needs sufficient paths from each state of a model to satisfy all the existential path formulas that have to be true in that state (those of the form $\exists g$ or $\neg \forall g$). Moreover, the number of existential state formulas that can appear in a formula is bounded by the number of path quantifiers in that formula. This then makes it possible to show that a satisfiable formula with n path quantifiers always has an $n+1$-tree model. Indeed, having $n+1$ branches out of each node of the tree makes it possible to embed in the tree the n paths that are necessary from each node, without these interfering with one another. This is done by embedding the jth path out of a node x into the path $x(j+1)1^\star$ of the tree. That path is the one that goes to the $j+1$st successor of the node in the tree and then always takes the leftmost branch of the tree.

Lemma 2.7 *If a BTL formula f containing n path quantifiers has a model, then it has an $(n+1)$-ary canonical model.*

Proof: Before giving the proof, we define the state closure of a CTL or CTL* formula f ($scl(f)$) as the set of state subformulas of f and their negation, where we identify $\neg\neg f_1$ with f_1. We also define existential state formulas as those of the form $\exists g$ or $\neg\forall g$ where g is a path formula.

Now, suppose f is satisfied by a branching interpretation $\langle M, w_0\rangle$ where $M = (W, Q, \pi)$. We construct a $(n+1)$-ary canonical model $\pi' : [n+1]^\star \rightarrow 2^P$ satisfying f by inductively constructing a mapping $\psi : [n+1]^\star \rightarrow W$ and taking $\pi'(x) = \pi(\psi(x))$. We start with a mapping ψ_0 defined on the set $X_0 = 1^\star$ (i.e., the leftmost path starting at λ). To define ψ_0, we choose some linear interpretation $\langle M_L, w_0\rangle$ extracted from $\langle M, w_0\rangle$ where $M_L = (W_L, R_L, \pi_L)$ and define $\psi_0(1^k) = R_L^k(w_0)$, $k \geq 0$. Note that each node $x \in X_0$ has exactly one successor in X_0 which is its leftmost successor $x1$.

Now, given the mapping ψ_i defined on a set of nodes X_i, we show how to define ψ_{i+1} and the set X_{i+1}. Our construction ensures that all nodes in X_i either have $n+1$ successors within X_i or only have their leftmost successor in X_i. Consider the nodes in X_i that only have a leftmost successor in X_i. For each such node x, we extend ψ_i as follows. Consider the existential state formulas $E \in scl(f)$ such that $\langle M, \psi(x)\rangle \models E$. There are at most n such formulas, say $E_1, \ldots, E_n$, where n is the number of path quantifiers in f and where E_j is either $\exists g_j$ or $\neg\forall g_j$. For each formula E_j, we choose a linear interpretation $\langle M_{Lj}, \psi_i(x)\rangle$ extracted from $\langle M, \psi_i(x)\rangle$ such that $\langle M_{Lj}, \psi_i(x)\rangle \models g_j$ if E_j is $\exists g_j$ and $\langle M_{Lj}, \psi_i(x)\rangle \models \neg g_j$ if E_j is $\neg\forall g_j$. We then define $\psi_{i+1}(x(j+1)) = R_{L_j}(\psi_i(x))$ and $\psi_{i+1}(x(j+1)1^k) = R_{L_j}^k(\psi_i(x))$ for $k \geq 1$. If there are only $0 \leq \ell < n$ formulas E_j, for $j = \ell+1, \ldots, n$, we choose $\langle M_{Lj}, \psi_i(x)\rangle$ to be an arbitrary linear interpretation extracted from $\langle M, \psi_i(x)\rangle$.

Finally, take ψ to be ψ_ω. The last step is to show that for all $f_s \in scl(f)$ and all nodes in $x \in [n+1]^\star$ we have $\langle \pi', x\rangle \models f_s$ iff $\langle M, \psi(x)\rangle \models g$ and hence in particular, $\langle \pi', \lambda\rangle \models f$ iff $\langle M, w_0\rangle \models f$. This is done by induction on the structure of state formulas. The case of atomic propositions and boolean combinations is immediate and the construction ensures that if a state formula of the form $\exists g$ is satisfied in a state w of M, it will be satisfied in any state $x \in [n+1]^\star$ such that $\psi(x) = w$. Also, as all paths in π' are paths in M, the same is true for formulas of the form $\forall g$. ∎

Note that the canonical model π' constructed in the proof of Lemma 2.7 is not equivalent to the initial model M. Indeed it might contain only a limited part of the states and paths of M. However, it contains sufficient states and paths to satisfy f if M satisfies f.

We are now ready to study the model generators for BTL formulas. We will only establish the results for CTL. Similar results can be obtained for CTL*, though the complexity bounds are quite different [ES84], [VS85]. The model generators for the k-ary tree canonical models of CTL will be Büchi tree automata. Büchi tree automata are similar to Büchi sequential automata, except that they operate on trees rather than sequences. They are identical to the *special automata* of [Ra70].

A k-ary *Büchi tree automaton* is a tuple $A = (\Sigma, S, \rho, S_0, F)$, where

- Σ is an alphabet,
- S is a set of states,
- $\rho : S \times \Sigma \to 2^{S^k}$ is a nondeterministic transition function (for each state and letter it gives the possible tuples of k successors)
- $S_0 \subseteq S$ is a set of starting states, and
- $F \subseteq S$ is a set of designated states.

The only difference between a Büchi sequential automaton and a Büchi tree automaton is that in the latter, the transition function maps states and letters to sets of k-tuples of states rather than to sets of states.

A *run* of an automaton A over a tree $T : [k]^* \to \Sigma$ is an n-ary tree $\phi : [k]^* \to S$ where $\phi(\lambda) \in S_0$ and for every $x \in [k]^*$, we have $(\phi(x1), \ldots, \phi(xn)) \in \rho(\phi(x), T(x))$. In intuitive terms, one can think of a run of a Büchi tree automaton A on a tree T as a labeling of T with states of A that is compatible with the transition relation of A. To define accepting runs, we consider paths in the run ϕ. A path from a node x in an infinite tree is an infinite sequence $p = x_0, x_1, x_2, \ldots, x_i, \ldots$ of nodes of the tree such that $x_0 = x$ and for all $i \geq 0$, x_{i+1} is an immediate successor of x_i (i.e., $x_{i+1} = x_i j$ for some $1 \leq j \leq k$). A run ϕ of A over T is *accepting* if and only if, on all infinite paths p starting at λ, some state in F appears infinitely often. The automaton A *accepts* T if it has an accepting run on T. We denote by $T(A)$ the set of trees accepted by A.

Example 2.5 The Büchi tree automaton of Figure 4 where $F = \{s_1\}$ and $S = \{s_0\}$ accepts all binary trees in which all paths contain b infinitely often. ∎

Theorem 2.8 *Given a CTL formula f and a constant k, one can construct a k-ary Büchi tree automaton $A = (\Sigma, S, \rho, S_0, F)$, where $\Sigma = 2^P$, and $|S| \leq 2^{O(|f|)}$, such that A accepts exactly all the k-ary tree models of f.*

Proof: A proof of this theorem can be obtained by the techniques described in [VW86a]. Here, we will describe a different construction that we have made as simple as possible and that is quite similar to the proof we have given for Theorem 2.3.

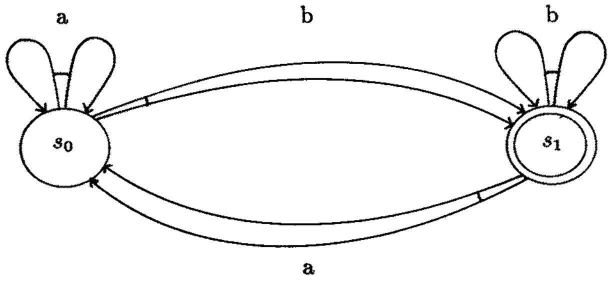

Figure 4: A Büchi tree automaton

The construction uses the notion of the *closure* of a formula f, denoted $cl(f)$. In the case of CTL, the notions of closure and *state closure* coincide. This is not the case for CTL*. For a CTL formula f, $cl(f)$ consists of all the subformulas of f and their negation, where we identify $\neg\neg f_1$ with f_1. Precisely, $cl(f)$ is defined as follows:

- $f \in cl(f)$
- $f_1 \wedge f_2 \in cl(f) \rightarrow f_1, f_2 \in cl(f)$
- $\neg f_2 \in cl(f) \rightarrow f_2 \in cl(f)$
- $f_2 \in cl(f) \rightarrow \neg f_2 \in cl(f)$
- $\exists \bigcirc f_2 \in cl(f) \rightarrow f_2 \in cl(f)$
- $\forall \bigcirc f_2 \in cl(f) \rightarrow f_2 \in cl(f)$
- $\exists f_1 \, U \, f_2 \in cl(f) \rightarrow f_1, f_2 \in cl(f)$
- $\forall f_1 \, U \, f_2 \in cl(f) \rightarrow f_1, f_2 \in cl(f)$

Note that we have $|cl(f)| \leq 2|f|$. The Büchi tree automaton we build for a formula f is taken as the combination of three automata: the *local automaton*, the *existential eventuality automaton* and the *universal eventuality automaton*. As in the proof of Theorem 2.3, the local automaton checks that the tree satisfies local consistency conditions. On the other hand, the eventuality automaton has to be split into two parts: one that checks eventualities that have to be satisfied on some path (existential eventualities) and eventualities that have to be satisfied on all paths (universal eventualities).

Constructing the Local Automaton

The local automaton is $L = (2^{cl(f)}, N_L, \rho_L, N_f, N_L)$. The state set N_L is the set of all sets $\mathbf{s}$ of formulas in $cl(f)$ that do not have any propositional inconsistency. Namely they must satisfy the following conditions:

- for all $f_1 \in cl(f)$, we have that $f_1 \in \mathbf{s}$ iff $\neg f_1 \notin \mathbf{s}$.
- for all $f_1 \wedge f_2 \in cl(f)$, we have that $f_1 \wedge f_2 \in \mathbf{s}$ iff $f_1 \in \mathbf{s}$ and $f_2 \in \mathbf{s}$.

For the transition relation $\rho_L : N_L \times 2^{cl(f)} \rightarrow 2^{N_L^k}$, we have that $(\mathbf{t}_1, \ldots, \mathbf{t}_k) \in \rho_L(\mathbf{s}, a)$ iff $a = \mathbf{s}$ and:

- for all $\exists \bigcirc f_1 \in cl(f)$, we have that $\exists \bigcirc f_1 \in \mathbf{s}$ iff $f_1 \in \mathbf{t}_j$, for some $1 \leq j \leq k$
- for all $\forall \bigcirc f_1 \in cl(f)$, we have that $\forall \bigcirc f_1 \in \mathbf{s}$ iff $f_1 \in \mathbf{t}_j$, for all $1 \leq j \leq k$
- for all $\exists f_1 \, U \, f_2 \in cl(f)$, we have that $\exists f_1 \, U \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or $f_1 \in \mathbf{s}$ and $f_1 \, U \, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$.
- for all $\forall f_1 \, U \, f_2 \in cl(f)$, we have that $\forall f_1 \, U \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or $f_1 \in \mathbf{s}$ and $f_1 \, U \, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$.

Finally, the set of starting states N_f consists of all sets $\mathbf{s}$ such that $f \in \mathbf{s}$. The local automaton does not impose any acceptance conditions and so its set of accepting states is the whole set of states.

The Universal Eventuality Automaton

Given a CTL formula f, we define the set $ue(f)$ of its universal eventualities as the subset of $cl(f)$ that contains all formulas of the form $\forall f_1 \, U \, f_2$. The eventuality automaton is $UE = (2^{cl(f)}, 2^{ue(f)}, \rho_{UE}, \{\emptyset\}, \{\emptyset\})$, where for the transition relation ρ_{UE}, we have that $(\mathbf{t}_1, \ldots, \mathbf{t}_k) \in \rho_{UE}(\mathbf{s}, a)$ iff:

- $\mathbf{s} = \emptyset$ and for all $\forall f_1 \, U \, f_2 \in a$, we have that $\forall f_1 \, U \, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ iff $f_2 \notin a$.
- $\mathbf{s} \neq \emptyset$ and for all $\forall f_1 \, U \, f_2 \in \mathbf{s}$, we have that $\forall f_1 \, U \, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ iff $f_2 \notin a$.

Intuitively, the universal eventuality automaton defined here is the same as the eventuality automaton used in the proof of Theorem 2.3 except that it runs down all the branches of the tree. Note that this can be done quite easily as the eventuality automaton is deterministic.

The Existential Eventuality Automaton

Given a CTL formula f, we define the set $ee(f)$ of its existential eventualities as the subset of $cl(f)$ that contains all formulas of the form $\exists f_1 \, U \, f_2$. The existential eventuality automaton is similar to the universal eventuality automaton. Indeed, it has to allow for different existential eventualities being satisfied on different paths while ensuring that all of them are satisfied on some path. $EE = (2^{cl(f)}, 2^{ee(f)}, \rho_{EE}, \{\emptyset\}, \{\emptyset\})$, where for the transition relation ρ_{EE}, we have that $(\mathbf{t}_1, \ldots, \mathbf{t}_k) \in \rho_{EE}(\mathbf{s}, a)$ iff:

- $\mathbf{s} = \emptyset$ and for all $\exists f_1 \, U \, f_2 \in a$, we have that $\exists f_1 \, U \, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ iff $f_2 \notin a$.
- $\mathbf{s} \neq \emptyset$ and for all $\exists f_1 \, U \, f_2 \in \mathbf{s}$, we have that $\exists f_1 \, U \, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ iff $f_2 \notin a$.

It is slightly more difficult to understand why the existential eventuality automaton indeed checks that all existential eventualities are satisfied. Let us sketch a proof of the fact that a tree is accepted by the existential eventuality automaton iff all the existential eventualities labeling its nodes are satisfied. First, let us assume that the tree T is accepted. Consider an existential eventuality ee labeling a node x of the tree T. We consider two possible cases. In the first case, in the accepting run of EE over T, EE is in state $\emptyset$ at node x. Then it is clear that the eventuality ee will be satisfied. In the second case, at node x, EE is not in the state $\emptyset$. As the run is accepting it will reach the state $\emptyset$ on every path from x. Now, because of the conditions imposed by the local automaton, ee will appear in one of these nodes unless it has been satisfied previously. The second case thus reduces to the first case.

Proving that a tree on which the existential eventualities are satisfied is accepted is easier. One can generate an accepting computation of EE by choosing transitions of EE in such a way that the computation corresponding to each eventuality follows a path on which that eventuality is satisfied. The fact that all eventualities are satisfied ensures that such a computation exists.

Combining the Automata

To obtain the automaton accepting the k-ary canonical models of the formula f, we take the intersection of the three automata we have just described. This can be done by the construction given in [VW86a]. The number of states of the resulting automaton will be twice the number of states in the product of the component automata and will thus be bounded by $2^{O(|f|)}$.

This automaton, accepts words over $2^{cl(f)}$ so we still have to take its projection on the set 2^P which is done exactly as in the proof of Theorem 2.3. ∎

Example 2.6 The Büchi tree automaton of Figure 4 is a model generator for the formula $\forall \Box \forall \Diamond p$ when a is taken to be $\emptyset$ and b to be $\{p\}$. Note that the automaton obtained from the construction given in the proof of Theorem 2.8 is somewhat more complicated, though it is equivalent (generates the same infinite trees) to the automaton we have described. ∎

We can now give the analogue of Theorem 2.4 concerning the *emptiness problem* for Büchi tree automata, i.e., the problem of determining, for a given Büchi tree automaton A, whether $T(A)$ is empty.

Theorem 2.9 *(1) [Ra70] The emptiness problem for Büchi tree automata is solvable in quadratic time.*

(2) [VW86a] The emptiness problem for Büchi tree automata is logspace complete for PTIME.

It is straightforward to obtain a decision procedure for CTL by combining Theorems 2.8, 2.9 (1) and Lemma 2.7. Given a formula f, one builds (using Theorem 2.8) the tree automaton accepting the k-ary tree models of this formula for a k greater than the number of path quantifiers in the formula. By Lemma 2.7 it is then sufficient to check using Theorem 2.9(1) that this automaton is nonempty. This yields the exponential time upper bound for the satisfiability problem of CTL (originally proven in [EH85a]). A matching lower bound is also proven in [EH85a]. Note that for the logic CTL*, the best known decision procedure (also obtained by automata theoretic techniques) is of nondeterministic double exponential time complexity [Em85], [VS85]. For that logic, the best known lower bound is double exponential time [VS85].

2.3 Partial Order Temporal Logic

In [PW84] a new version of temporal logic was introduced to reason about partially ordered computations. However, from a purely logical point of view, this logic is a version of branching time temporal logic that includes past operators. Its connection to partial orders is not intrinsic, but is due to the way in which it is used. We will study this issue in detail in a later section. Here, we will define POTL and study its inherent properties such as its models and model generators.

2.3.1 Syntax

We will only consider one version of POTL, namely the one corresponding to CTL. This version is a slight generalization of the logic defined in [PW84] where the operator U was not considered. The syntax of POTL is identical to that of CTL except that we also allow two *backwards path operators*: $\overline{\bigcirc}$ (previous) and $\overline{U}$ (since). As in CTL, path formulas (either backwards or forwards) can only contain a single temporal operator. Besides the abbreviations used in CTL, we use $\exists\overline{\Diamond} f$ and $\forall\overline{\Diamond} f$ to abbreviate $\exists(\mathit{true}\,\overline{U}\, f)$ and $\forall(\mathit{true}\,\overline{U}\, f)$ respectively. Also, $\exists\overline{\Box} f$ will be an abbreviation for $\neg\forall\overline{\Diamond}\neg f$ and $\forall\overline{\Box} f$ for $\neg\exists\overline{\Diamond}\neg f$.

In POTL, we will need to distinguish between forwards and backwards path formulas (i.e. path formulas where the temporal operators is $\overline{\bigcirc}$ or $\overline{U}$). Note that in POTL path formulas only contain one operator and hence do not mix forwards and backwards operators. For ease of exposition, we define the *reverse* $\overline{g}$ of a path formula g as g where $\bigcirc$ is replaced by $\overline{\bigcirc}$ and U by $\overline{U}$ if g is a forwards path formula, and as g where $\overline{\bigcirc}$ and $\overline{U}$ are respectively replaced by $\bigcirc$ and U if g is a backwards formula.

Example 2.7 $\forall \overline{\Box} \exists \bigcirc \forall p \overline{U} q$ is a POTL formula. ∎

2.3.2 Semantics

The logic POTL is interpreted over exactly the same type of structure as BTL. The only additional requirement is that the accessibility relation should be total in both directions. In other words, each state should have at least one successor and at least one predecessor. To define the semantics of POTL, we need to talk about backwards paths extracted from structures. This leads us to a definition symmetric to Definition 2.2.

Definition 2.4 *A backwards linear interpretation extracted from a branching interpretation* $\langle M_B, w_0\rangle$ *where* $M_B = (W_B, Q_B, \pi_B)$ *is a pair* $\langle M_L, w_0\rangle$ *where* $M_L = (W_L, R_L, \pi_L)$ *with* $w_0 \in W_L \subseteq W_B$, $\pi_L(w) = \pi_B(w)$ *and* $(R_L(w), w) \in Q_B$ *for all* $w \in W_L$.

The only difference between Definitions 2.2 and 2.4 is that in Definition 2.4, we have extracted the linear interpretation in the direction opposite to the one defined by the relation Q_B. In other words, moving forwards in the backwards linear interpretation corresponds to moving backwards in the branching model.

The semantics of POTL are identical to those of CTL except where backwards path formulas appear. For these, we have the following two clauses:

- $\langle M_B, w_0\rangle \models \forall g$ where g is a backwards path formula iff for all backwards linear structures $\langle M_L, w_0\rangle$ extracted from $\langle M_B, w_0\rangle$, $\langle M_L, w_0\rangle \models \overline{g}$
- $\langle M_B, w_0\rangle \models \exists g$ where g is a backwards path formula iff for some backwards linear structures $\langle M_L, w_0\rangle$ extracted from $\langle M_B, w_0\rangle$, $\langle M_L, w_0\rangle \models \overline{g}$

2.3.3 Models and Model Generators

Partial order temporal logic does not have trees as canonical models. Indeed a formula like $\exists \overline{\bigcirc} p \wedge \exists \overline{\bigcirc} \neg p$ is only true in a node that has at least two predecessors. This is clearly not possible in a tree where each node has only one successor. In [PW84] a class of models called ***backwards-forwards trees*** is considered. An easy way to understand backwards-forwards trees is to think of trees where the edges leaving each node point either forwards or backwards (see Figure 5). An

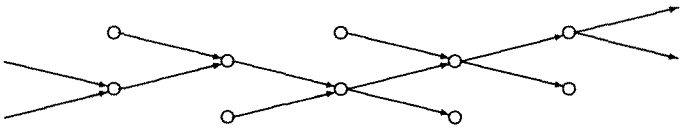

Figure 5: A backwards-forwards tree

equivalent representation is to have all the edges of the tree in the same direction, but to label them by either + or − depending on their direction (see Figure 6). As in a tree each node has

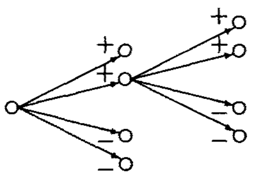

Figure 6: A backwards-forwards labeled tree

only one edge leading to it, the + and − labels can be placed on the nodes rather than on the edges. We will call such a tree with nodes labeled by + or − a ± tree. If besides the ± labels, the tree also has labels from an alphabet Σ, we will call it a labeled ± tree. Formally, a k-ary labeled ± tree T is a function $T : [k]^* \rightarrow \Sigma \times [\pm]$, where $[\pm]$ denotes the set $\{+, -\}$. For POTL, our canonical models will be k-ary ± trees labeled over the alphabet 2^P, i.e., interpretations $\langle M, \lambda \rangle$ where $M = ([k]^*, succ, \tau)$, τ being a function from $[k]^*$ to $2^P \times [\pm]$. Note that a ± tree is simply a representation of a special type of branching structure. The branching structure corresponding to a ± tree can be obtained by simply inverting all the edges leading to nodes labeled −. As a matter of terminology, we will use the words "son" and "father" when referring to the respective position of nodes in a ± tree. We will use the words successor and predecessor to talk about the respective position of such nodes when talking into account the polarity (± label) of the nodes. For instance, the − son of a node is its predecessor, whereas the father of a − node is its successor.

Now we turn to our notion of equivalence of interpretations. It needs to be adapted as equivalent interpretations should have the same transitions *in both directions.* We will call the notion of equivalence we use on interpretations for POTL formulas *two-way equivalence.*

Definition 2.5 *Two branching interpretations $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ where $M_1 = (W_1, Q_1, \pi_1)$ and $M_2 = (W_2, Q_2, \pi_2)$ are two-way equivalent iff there is relation $\phi \subseteq W_1 \times W_2$ such that:*

- $(w_{01}, w_{02}) \in \phi$;

- *if* $(w_1, w_2) \in \phi$, *then* $\pi_1(w_1) = \pi_2(w_2)$;

- *if* $(w_1, w_2) \in \phi$, *and there is an* $w_1' \in W_1$ *such that* $(w_1, w_1') \in Q_1$ *then there is an* $w_2' \in W_2$ *such that* $(w_2, w_2') \in Q_2$ *and* $(w_1', w_2') \in \phi$;

- *if* $(w_1, w_2) \in \phi$, *and there is an* $w_2' \in W_2$ *such that* $(w_2, w_2') \in Q_2$ *then there is an* $w_1' \in W_1$ *such that* $(w_1, w_1') \in Q_1$ *and* $(w_1', w_2') \in \phi$;

- *if* $(w_1, w_2) \in \phi$, *and there is an* $w_1' \in W_1$ *such that* $(w_1', w_1) \in Q_1$ *then there is an* $w_2' \in W_2$ *such that* $(w_2', w_2) \in Q_2$ *and* $(w_1', w_2') \in \phi$;

- *if* $(w_1, w_2) \in \phi$, *and there is an* $w_2' \in W_2$ *such that* $(w_2', w_2) \in Q_2$ *then there is an* $w_1' \in W_1$ *such that* $(w_1', w_1) \in Q_1$ *and* $(w_1', w_2') \in \phi$.

The difference between two-way equivalence and the equivalence notion of Definition 2.3 is that two way equivalence considers both the edges leaving and the edges leading to a node. Note that in the restricted case of $\pm$ trees, an equivalent way to state Definition 2.5 is to use Definition 2.3 and require that if $(w_1, w_2) \in \phi$, then w_1 and w_2 also agree on their $\pm$ label. We now establish the analogue of Lemma 2.5.

Lemma 2.10 *If two branching interpretations $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ are two-way equivalent, then for any POTL formula f, $\langle M_1, w_{01}\rangle \models f$ iff $\langle M_2, w_{02}\rangle \models f$.*

Proof: The proof is identical to the proof of Lemma 2.5, except that here, we have to notice that if $\langle M_1, w_{01}\rangle$ and $\langle M_2, w_{02}\rangle$ are two-way equivalent branching structure, then, from these interpretations, one can extract equivalent forwards *and backwards* linear interpretations. ∎

To establish the existence of canonical models for POTL, we establish the analogue of Lemma 2.6. For this, we need to take into account both incoming and outgoing edges in the definition of the branching factor of a structure. To distinguish this branching factor from the one used in Lemma 2.6, we will call it the *two-way branching factor.* By definition, the two-way branching factor of a structure (W, Q, π) is the largest ℓ such that for some $w \in W$, we have $\ell = |\{w' | (w, w') \in Q \lor (w', w) \in Q\}|$.

Lemma 2.11 *Every branching interpretation $\langle M, w_0\rangle$ where $M = (W, Q.\pi)$ and whose two-way branching factor is $\leq k$ is equivalent to a k-ary $\pm$ tree canonical interpretation $\tau : [k]^* \rightarrow 2^P \times [\pm]$.*

Proof: We define the canonical interpretation τ simultaneously with the relation ϕ that will enable us to establish that it is equivalent to $\langle M, w_0\rangle$. The relation ϕ is actually a function in $[k]^* \rightarrow W$. We define τ and ϕ in stages. At stage i, ϕ_i will be defined on all nodes of the $\pm$ tree of depth $\leq i$. We start with $\phi_0(\lambda) = w_0$ and $\tau_0(\lambda) = (\pi(\phi_0(\lambda), +)$. Note that for $\tau_0(\lambda)$, the choice of the label $+$ is arbitrary. We obtain ϕ_{i+1} from ϕ_i by defining it on all nodes of depth $i+1$ as follows. For each node $x \in [k]^*$ of depth i, let $\{w_{s_1}, \ldots, w_{s_\ell}\}$, be the successors of $\phi_i(x)$ (i.e., nodes w_j such that $(\phi(x), w_j) \in Q$) and let $\{w_{p_1}, \ldots, w_{p_\ell}\}$ be the predecessors of $\phi_i(x)$ (i.e., nodes w_j such that $(w_j, \phi(x)) \in Q$). By hypothesis, we have that $s_\ell + p_\ell \leq k$. We define $\phi_{i+1}(xj) = w_j$ for $s_1 \leq j \leq s_\ell$ or $p_1 \leq j \leq p_\ell$ and for all other $1 \leq j \leq k$, we take $\phi_{i+1}(xj) = w_{s_\ell}$. In other words the first s_ℓ sons of x simulate the s_ℓ successors of $\phi_i(x)$, the next p_ℓ sons of x simulate the p_ℓ predecessors of x and if $s_\ell + p_\ell < k$, we simply replicate the last successor. We define τ_{i+1} on the nodes xj of depth $i+1$ by $\tau_{i+1}(xj) = (\pi(\phi_{i+1}(xj)), +)$ for $s_1 \leq j \leq s_\ell$ and $s_\ell + p_\ell < j \leq k$ and by $\tau_{i+1}(xj) = (\pi(\phi_{i+1}(xj)), -)$ for $p_1 \leq j \leq p_\ell$. The relation ϕ and the function τ are taken to be ϕ_ω and τ_ω. ∎

Our next step is to show that $\pm$-tree models with limited arity are an interesting class of models for POTL formulas. For this, we prove that if a POTL formula containing n path quantifiers has a model, then it has an $(n+1)$-ary $\pm$ canonical tree model. The proof is basically identical to the proof of Lemma 2.7 except that $\pm$ labels have to be introduced appropriately and that we have to ensure that each node has at least one successor and one predecessor.

Lemma 2.12 *If a POTL formula f containing n path quantifiers has a model, then it has an $(n+1)$-ary $\pm$ tree model.*

Proof: Suppose f is satisfied by an interpretation $\langle M, w_0\rangle$ where $M = (W, Q, \pi)$. We construct a $n+1$-ary $\pm$ tree model τ satisfying f by inductively constructing a mapping $\psi : [n+1]^* \rightarrow W$ and defining τ in terms of ψ. We start with a mapping ψ_0 and a labeling τ_0 defined on the set $X_0 = 1^*$ (i.e. the leftmost path starting at λ). To define ψ_0 and τ_0, we choose either a forwards or a backwards linear interpretation $\langle M_L, w_0\rangle$ extracted from $\langle M, w_0\rangle$ where $M_L = (W_L, R_L, \pi_L)$. If all the eventualities that are satisfied in w_0 are forwards, we choose a backwards path. If they are all backwards, we choose a forwards path. If some are forwards and some are backwards, the choice does not matter. The reason for this particular choice is to ensure that the root of the tree has both at least one $+$ son and one $-$ son. If the chosen path is forwards, we define $\psi_0(1^k) = R_L^k(w_0)$ and $\tau_0(1^k) = (\pi(\psi_0(1^k)), +)$, if it is backwards, τ_0 is defined by $\tau_0(1^k) = (\pi(\psi_0(1^k)), -)$. Note that each node $x \in X_0$ has exactly one son in X_0 which is its leftmost son $x1$.

Now, given the mapping ψ_i defined on the set of nodes X_i, we show how to define ψ_{i+1} and the set X_{i+1}. Our construction ensures that all nodes in X_i either have $n+1$ sons within X_i or only have their leftmost son in X_i. Consider the nodes in X_i that only have a leftmost successor in X_i. For each such node x, we extend ψ_i as follows. Consider all existential state formulas $E \in scl(f)$ such that $\langle M, \psi_i(x)\rangle \models E$. There are at most n such formulas, say $E_1, \dots, E_n$, where n is the number of path quantifiers in f and where E_j is either $\exists g_j$ or $\neg\forall g_j$. For each formula E_j, we choose a linear interpretation $\langle M_{Lj}, \psi_i(x)\rangle$ extracted from $\langle M, \psi_i(x)\rangle$ (forwards or backwards depending on whether g_j is a forwards or backwards formula). This linear interpretation must be such that $\langle M_{Lj}, \psi_i(x)\rangle \models g_j$ if E_j is $\exists g_j$ and g_j is a forwards formula and such that and $\langle M_{Lj}, \psi_i(x)\rangle \models \overline{g_j}$ if g_j is a backwards formula. If E_j is $\neg\forall g_j$, then the linear interpretation must satisfy either $\neg g_j$ or $\overline{\neg g_j}$ depending on whether g_j is forwards or backwards. We then define $\psi_{i+1}(x(j+1)) = R_{L_j}(\psi_i(x))$ and $\psi_{i+1}(x(j+1)1^k) = R^k_{L_j}(\psi_i(x))$ for $k \geq 1$. The mapping τ_{i+1} is then defined on these points by $\tau_{i+1}(x(j+1)^k) = (\pi(\psi_i(x(j+1)^k)), +)$ if the eventuality E_j is forwards. If it is backwards, the $+$ in the definition of τ_{i+1} is replaced by $-$. If there are only $0 \leq \ell < n$ formulas E_j, for $j = \ell+1, ..., n$, we choose $\langle M_{Lj}, \psi_i(x)\rangle$ to be an arbitrary linear interpretation extracted from $\langle M, \psi_i(x)\rangle$. Note that this construction ensures that each node of the $\pm$ tree model has a successor and a predecessor. Indeed, a node and its leftmost son always have the same $\pm$ label. Hence if the son of that node is a successor, its father is a predecessor and vice-versa. The relation ϕ and the function τ are taken to be ϕ_ω and τ_ω. ∎

Our last step is to show how to build a Büchi tree automaton that accepts all the k-ary $\pm$ tree models of a POTL formula f. The apparent difficulty in this construction is that we might have to go up and down the tree to take into account the fact that we are dealing with forwards and backwards formulas. Fortunately, it is possible to avoid checking conditions going up the tree by constructing an automaton that operates on trees labeled by elements of an extension of the closure of the formula f. This technique was introduced in [VW86a] and further developed in [Va85b].

Theorem 2.13 *Given a POTL formula f and a constant k, one can construct a k-ary Büchi tree automaton $A = (\Sigma, S, \rho, S_0, F)$, where $\Sigma = 2^P \times [\pm]$, and $|S| \leq 2^{O(|f|)}$, such that A accepts exactly all the k-ary $\pm$ tree models of f.*

Proof: To build the automaton, we use an extension of the closure of the formula f. The extended closure of f ($ecl(f)$) is defined by the following rules:

- $f \in ecl(f)$
- $f_1 \wedge f_2 \in ecl(f) \rightarrow f_1, f_2 \in ecl(f)$
- $\neg f_2 \in ecl(f) \rightarrow f_2 \in ecl(f)$

- $f_2 \in ecl(f) \rightarrow \neg f_2 \in ecl(f)$
- $\exists \bigcirc f_2 \in ecl(f) \rightarrow f_2, \exists_d \bigcirc f_2, \exists_u \bigcirc f_2 \in ecl(f)$
- $\forall \bigcirc f_2 \in ecl(f) \rightarrow f_2, \forall_d \bigcirc f_2, \forall_u \bigcirc f_2 \in ecl(f)$
- $\exists f_1 U f_2 \in ecl(f) \rightarrow f_1, f_2, \exists_d f_1 U f_2, \exists_u f_1 U f_2 \in ecl(f)$
- $\forall f_1 U f_2 \in ecl(f) \rightarrow f_1, f_2, \forall_d f_1 U f_2, \forall_u f_1 U f_2 \in ecl(f)$
- $\exists \overline{\bigcirc} f_2 \in ecl(f) \rightarrow f_2, \exists_d \overline{\bigcirc} f_2, \exists_u \overline{\bigcirc} f_2 \in ecl(f)$
- $\forall \overline{\bigcirc} f_2 \in ecl(f) \rightarrow f_2, \forall_d \overline{\bigcirc} f_2, \forall_u \overline{\bigcirc} f_2 \in ecl(f)$
- $\exists f_1 \overline{U} f_2 \in ecl(f) \rightarrow f_1, f_2, \exists_d f_1 \overline{U} f_2, \exists_u f_1 \overline{U} f_2 \in ecl(f)$
- $\forall f_1 \overline{U} f_2 \in ecl(f) \rightarrow f_1, f_2, \forall_d f_1 \overline{U} f_2, \forall_u f_1 \overline{U} f_2 \in ecl(f)$

Besides, the addition of clauses for the past temporal operators, the difference between the $cl(f)$ defined in the proof of Theorem 2.8 and the $ecl(f)$ we have here is that for formulas of the form $\exists g$ and $\forall g$, we add the formulas $\exists_d g$, $\exists_u g$ and $\forall_u g$, $\forall_d g$ respectively. The intended use of these formulas is that $\exists_d g$ will be true in a node of a tree iff the path formula g is satisfied on a path going strictly down the tree whereas $\exists_u g$ will be truth on a path initially going up the tree. This distinction will for instance make it possible to decide if to check whether a formula $\exists \bigcirc p$ is satisfied in a node of the tree we have to look at the father or at the sons of this node. Note that if the node is labeled $-$, the father as well as all the sons of the node labeled $+$ are successors of the node. Similarly, $\forall_d g$ will be true in a node if g is true in all paths from that node going strictly downwards in the tree and $\forall_u g$ will be true if g is true on all paths initially going through the father of the node. Note that we take for the meaning of $\forall_d g$ that g is true on all paths that are strictly downwards. This makes things a little simpler and is sufficient as it is easy to show that for the path formulas of *POTL*, a formula is satisfied on all paths iff it is satisfied on all upwards and strictly downwards paths. Indeed, once a forwards path starts going down the tree, it cannot go back up (the $\pm$ label would be wrong) and similarly for a backwards path.

As in the proof of Theorem 2.8, the automaton we are constructing is defined as the combination of three parts: the ***local automaton***, the ***existential eventuality automaton*** and the ***universal eventuality automaton***. We start with the local automaton.

Constructing the Local Automaton

The local automaton is $L = (2^{ecl(f)} \times [\pm], N_L, \rho_L, N_f, N_L)$. The state set N_L is the subset of elements $(\mathbf{s}, \ell)$ of $2^{ecl(f)} \times [\pm]$ that satisfy the following conditions:

- for all $f_1 \in ecl(f)$, we have that $f_1 \in \mathbf{s}$ iff $\neg f_1 \notin \mathbf{s}$.

- for all $f_1 \wedge f_2 \in ecl(f)$, we have that $f_1 \wedge f_2 \in \mathbf{s}$ iff $f_1 \in \mathbf{s}$ and $f_2 \in \mathbf{s}$.

In the case where $\ell = +$, we also require:

- for all $\exists \bigcirc f_1 \in ecl(f)$ we have that $\exists \bigcirc f_1 \in \mathbf{s}$ iff $\exists_d \bigcirc f_1 \in \mathbf{s}$
- for all $\exists f_1 \, U \, f_2 \in ecl(f)$ we have that $\exists f_1 \, U \, f_2 \in \mathbf{s}$ iff $\exists_d f_1 \, U \, f_2 \in \mathbf{s}$
- for all $\forall \bigcirc f_1 \in ecl(f)$ we have that $\forall \bigcirc f_1 \in \mathbf{s}$ iff $\forall_d \bigcirc f_1 \in \mathbf{s}$
- for all $\forall f_1 \, U \, f_2 \in ecl(f)$ we have that $\forall f_1 \, U \, f_2 \in \mathbf{s}$ iff $\forall_d f_1 \, U \, f_2 \in \mathbf{s}$
- for all $\exists \overline{\bigcirc} f_1 \in ecl(f)$ we have that $\exists \overline{\bigcirc} f_1 \in \mathbf{s}$ iff $\exists_d \overline{\bigcirc} f_1 \in \mathbf{s}$ or $\exists_u \overline{\bigcirc} f_1 \in \mathbf{s}$
- for all $\exists f_1 \, \overline{U} \, f_2 \in ecl(f)$ we have that $\exists f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff $\exists_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ or $\exists_u f_1 \, \overline{U} \, f_2 \in \mathbf{s}$
- for all $\forall \overline{\bigcirc} f_1 \in ecl(f)$ we have that $\forall \overline{\bigcirc} f_1 \in \mathbf{s}$ iff $\forall_d \overline{\bigcirc} f_1 \in \mathbf{s}$ and $\forall_u \overline{\bigcirc} f_1 \in \mathbf{s}$
- for all $\forall f_1 \, \overline{U} \, f_2 \in ecl(f)$ we have that $\forall f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff $\forall_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ and $\forall_u f_1 \, \overline{U} \, f_2 \in \mathbf{s}$

On the other hand, if $\ell = -$, we require:

- For all $\exists \bigcirc f_1 \in ecl(f)$ we have that $\exists \bigcirc f_1 \in \mathbf{s}$ iff $\exists_d \bigcirc f_1 \in \mathbf{s}$ or $\exists_u \bigcirc f_1 \in \mathbf{s}$
- for all $\exists f_1 \, U \, f_2 \in ecl(f)$ we have that $\exists f_1 \, U \, f_2 \in \mathbf{s}$ iff $\exists_d f_1 \, U \, f_2 \in \mathbf{s}$ or $\exists_u f_1 \, U \, f_2 \in \mathbf{s}$
- for all $\forall \bigcirc f_1 \in ecl(f)$ we have that $\forall \bigcirc f_1 \in \mathbf{s}$ iff $\forall_d \bigcirc f_1 \in \mathbf{s}$ and $\forall_u \bigcirc f_1 \in \mathbf{s}$
- for all $\forall f_1 \, U \, f_2 \in ecl(f)$ we have that $\forall f_1 \, U \, f_2 \in \mathbf{s}$ iff $\forall_d f_1 \, U \, f_2 \in \mathbf{s}$ and $\forall_u f_1 \, U \, f_2 \in \mathbf{s}$
- for all $\exists \overline{\bigcirc} f_1 \in ecl(f)$ we have that $\exists \overline{\bigcirc} f_1 \in \mathbf{s}$ iff $\exists_d \overline{\bigcirc} f_1 \in \mathbf{s}$
- for all $\exists f_1 \, \overline{U} \, f_2 \in ecl(f)$ we have that $\exists f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff $\exists_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$
- for all $\forall \overline{\bigcirc} f_1 \in ecl(f)$ we have that $\forall \overline{\bigcirc} f_1 \in \mathbf{s}$ iff $\forall_d \overline{\bigcirc} f_1 \in \mathbf{s}$
- for all $\forall f_1 \, \overline{U} \, f_2 \in ecl(f)$ we have that $\forall f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff $\forall_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$

For the transition relation $\rho_L : N_L \times (2^{ecl(f)} \times [\pm]) \to 2^{N_L^k}$, we have that $((\mathbf{t}_1, \ell_1), \ldots, (\mathbf{t}_k, \ell_k)) \in \rho_L((\mathbf{s}, \ell), (a, \ell'))$ iff $(a, \ell') = (\mathbf{s}, \ell)$ and the conditions we are going to describe hold. We separate them in two groups: those dealing with downwards path formula and those dealing with upwards path formulas. First, we deal with downwards formulas

- for all $\exists_d \bigcirc f_1 \in ecl(f)$, we have that $\exists_d \bigcirc f_1 \in \mathbf{s}$ iff for some $1 \leq j \leq k$, $f_1 \in \mathbf{t}_j$ and $\ell_j = +$
- for all $\forall_d \bigcirc f_1 \in ecl(f)$, we have that $\forall_d \bigcirc f_1 \in \mathbf{s}$ iff for all $1 \leq j \leq k$ such that $\ell_j = +$, we have $f_1 \in \mathbf{t}_j$

- for all $\exists_d f_1 \, U \, f_2 \in ecl(f)$, we have that $\exists_d f_1 \, U \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or, $f_1 \in \mathbf{s}$ and for some $1 \leq j \leq k$, $\exists_d f_1 \, U \, f_2 \in \mathbf{t}_j$ and $\ell_j = +$
- for all $\forall_d f_1 \, U \, f_2 \in ecl(f)$, we have that $\forall_d f_1 \, U \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or, $f_1 \in \mathbf{s}$ and for all $1 \leq j \leq k$ for which $\ell_j = +$, $\forall_d f_1 \, U \, f_2 \in \mathbf{t}_j$
- for all $\exists_d \overline{\bigcirc} f_1 \in ecl(f)$, we have that $\exists_d \overline{\bigcirc} f_1 \in \mathbf{s}$ iff for some $1 \leq j \leq k$, $f_1 \in \mathbf{t}_j$ and $\ell_j = -$
- for all $\forall_d \overline{\bigcirc} f_1 \in ecl(f)$, we have that $\forall_d \overline{\bigcirc} f_1 \in \mathbf{s}$ iff for all $1 \leq j \leq k$ such that $\ell_j = -$, we have $f_1 \in \mathbf{t}_j$
- for all $\exists_d f_1 \, \overline{U} \, f_2 \in ecl(f)$, we have that $\exists_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or, $f_1 \in \mathbf{s}$ and for some $1 \leq j \leq k$, $\exists_d f_1 \, \overline{U} \, f_2 \in \mathbf{t}_j$ and $\ell_j = -$
- for all $\forall_d f_1 \, \overline{U} \, f_2 \in ecl(f)$, we have that $\forall_d f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ iff either $f_2 \in \mathbf{s}$ or, $f_1 \in \mathbf{s}$ and for all $1 \leq j \leq k$ for which $\ell_j = -$, $\forall_d f_1 \, \overline{U} \, f_2 \in \mathbf{t}_j$

We now give the conditions for upwards formulas. As each node in a tree has only one father, these conditions are identical for formulas of the form $\exists_u g$ and of the form $\forall_u g$. We thus only give them for existential formulas, but they do apply to both existential and universal upwards formulas.

- for all $\exists_u \bigcirc f_1 \in ecl(f)$, we have that $\exists_u \bigcirc f_1 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ iff $\ell_j = -$ and $f_1 \in \mathbf{s}$
- for all $\exists_u f_1 \, U \, f_2 \in ecl(f)$, we have that $\exists_u f_1 \, U \, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$, iff either $f_2 \in \mathbf{t}_j$ or, $f_1 \in \mathbf{t}_j$, $\exists f_1 \, U \, f_2 \in \mathbf{s}$ and $\ell_j = -$
- for all $\exists_u \overline{\bigcirc} f_1 \in ecl(f)$, we have that $\exists_u \overline{\bigcirc} f_1 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ iff $\ell_j = +$ and $f_1 \in \mathbf{s}$
- for all $\exists_u f_1 \, \overline{U} \, f_2 \in ecl(f)$, we have that $\exists_u f_1 \, \overline{U} \, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$, iff either $f_2 \in \mathbf{t}_j$ or, $f_1 \in \mathbf{t}_j$, $\exists f_1 \, \overline{U} \, f_2 \in \mathbf{s}$ and $\ell_j = +$

Finally, the set of starting states N_f consists of all pairs $(\mathbf{s}, \ell)$ such that $\mathbf{s}$ does not contain any $\exists_u$ or $\forall_u$ formulas and such that $f \in \mathbf{s}$. The local automaton does not impose any acceptance conditions and thus its set of accepting states is the whole set of states.

The Universal Eventuality Automaton

The eventuality automata (universal and existential) have to check that in all nodes of the tree, all eventualities are satisfied. The important observation here is that it is sufficient to do this for downwards eventualities. Indeed, if an eventuality is satisfied on an upwards path, either it will be satisfied while the path is moving up the tree and no further checking is necessary or it will eventually propagate into a downwards eventuality. This is the case as the local automaton forces unsatisfied eventualities to propagate and, when moving up the tree, one eventually reaches the root which is not allowed to contain upwards formulas.

These observations make the universal and existential eventuality automata very similar to the ones we used for CTL. Given a *POTL* formula f, we define the set $due(f)$ of its downwards universal eventualities as the subset of $ecl(f)$ that contains all formulas of the form $\forall_d f_1\, U\, f_2$ and $\forall_d f_1\, \overline{U}\, f_2$. The eventuality automaton is $UE = (2^{ecl(f)} \times [\pm], 2^{due(f)}, \rho_{UE}, \{\emptyset\}, \{\emptyset\})$, where for the transition relation ρ_{UE}, we have that $((\mathbf{t}_1, \ell_1), \ldots, (\mathbf{t}_k, \ell_k)) \in \rho_{UE}((\mathbf{s}, \ell), (a, \ell'))$ iff:

- $\mathbf{s} = \emptyset$ and
  - for all $\forall_d f_1\, U\, f_2 \in a$, we have that $f_2 \notin a$ iff $\forall_d f_1\, U\, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ such that $\ell_j = +$
  - for all $\forall_d f_1\, \overline{U}\, f_2 \in a$, we have that $f_2 \notin a$ iff $\forall_d f_1\, U\, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ such that $\ell_j = -$
- $\mathbf{s} \neq \emptyset$ and
  - for all $\forall_d f_1\, U\, f_2 \in \mathbf{s}$, $f_2 \notin a$ iff we have that $\forall_d f_1\, U\, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ such that $\ell_j = +$
  - for all $\forall_d f_1\, \overline{U}\, f_2 \in \mathbf{s}$, $f_2 \notin a$ iff we have that $\forall_d f_1\, \overline{U}\, f_2 \in \mathbf{t}_j$ for all $1 \leq j \leq k$ such that $\ell_j = -$

This automaton is essentially identical to the one defined in the proof of Theorem 2.8 except that it checks forwards eventualities on + nodes and backwards eventualities on − nodes.

The Existential Eventuality Automaton

Given an *POTL* formula f, we define the set $dee(f)$ of its downwards existential eventualities as the subset of $ecl(f)$ that contains all formulas of the form $\exists_d f_1\, U\, f_2$ and $\exists_d f_1\, \overline{U}\, f_2$. The existential eventuality automaton is also very similar to the one used in the proof of Theorem 2.8. It is the following: $EE = (2^{ecl(f)} \times [\pm], 2^{dee(f)}, \rho_{EE}, \{\emptyset\}, \{\emptyset\})$, where for the transition relation ρ_{EE}, we have that $((\mathbf{t}_1, \ell_1), \ldots, (\mathbf{t}_k, \ell_k)) \in \rho_{EE}((\mathbf{s}, \ell), (a, \ell'))$ iff:

- $\mathbf{s} = \emptyset$ and
  - for all $\exists_d f_1\, U\, f_2 \in a$, we have that $f_2 \notin a$ iff $\exists_d f_1\, U\, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ such that $\ell_j = +$
  - for all $\exists_d f_1\, \overline{U}\, f_2 \in a$, we have that $f_2 \notin a$ iff $\exists_d f_1\, U\, f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ such that $\ell_j = -$
- $\mathbf{s} \neq \emptyset$ and

- for all $\exists_d f_1 U f_2 \in \mathbf{s}$, $f_2 \notin a$ iff we have that $\exists_d f_1 U f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ such that $\ell_j = +$
- for all $\exists_d f_1 \overline{U} f_2 \in \mathbf{s}$, $f_2 \notin a$ iff we have that $\exists_d f_1 \overline{U} f_2 \in \mathbf{t}_j$ for some $1 \leq j \leq k$ such that $\ell_j = -$

Combining the Automata

Combining the automata and projecting on the alphabet 2^P is done exactly as in the proof of Theorem 2.8. ∎

Combining Theorem 2.13 with Theorem 2.9 and Lemma 2.12, we obtain an exponential decision procedure for POTL. Clearly, the matching lower bound proven for CTL also applies.

2.4 Summary

We have described three versions of temporal logic: linear time, branching time and partial order. For each we have given syntax and semantics and shown how, given a formula, one can construct a *model generator* describing up to equivalence all the models of the formula that are in a given class. The results are summarized in Table 1.

Logic	Models	Model Generators	Class of Models Described
linear	sequences	Büchi sequential automata	all models
branching	trees	Büchi tree automata	k-ary models
partial order	$\pm$ trees	Büchi tree automata	k-ary models

Table 1: Summary of Temporal Logics

3 Program Verification

In this section, we review and classify the applications of temporal logic to program verification. Our classification is organised around the different views one can have of programs and their computations. We will consider successively deterministic programs with linear executions, nondeterministic programs with linear executions and programs having partially ordered sets as executions. For each of these views, we show how verification methods can be obtained by relating the programs and computations to either models or model generators of the various temporal logics.

We will consider only finite-state programs. This has the advantage of simplifying the presentation and of relating nicely to the propositional temporal logics we described in the previous section. However, our classification and a number of the results we state here are also applicable to infinite-state programs and first-order temporal logic.

3.1 First View: Deterministic Programs

Verifying deterministic programs is not the most usual application of temporal logic. However it is a simple case and serves as a good introduction to our methodology.

A deterministic program is a tuple (W, R, w_0, π) where

- W is a set of states,
- R is a total transition function,
- w_0 is a unique initial state,
- $\pi : W \rightarrow \Sigma$ is a labeling function.

One can think of the label of a state as decribing the properties of that state and specifying the actions the program performs when entering that state. Notice that a deterministic program has a unique computation (or execution) which is the infinite sequence of states generated from the initial state: $w_0, R(w_0), R^2(w_0), \ldots$

Our approach is based on relating programs and their computations to either models or model generators for some version of temporal logic. In the case of deterministic programs this relation is straightforward. A deterministic program is exactly the same type of structure as a model for an LTL formula. The same is true of the computation of that program. Actually, the program and its computation are equivalent according to Definition 2.1, so we will not distinguish them in what follows.

As we explained in the introduction, to obtain a verification method, we need to choose some way of describing the program and some way of describing the desired properties of the program. Our choices are the following:

Program: Can be described explicitely by its states and transitions or can be described by a formula. As we are considering deterministic programs, the formula should be such that it has only one model up to equivalence.

Property: Even if the program is deterministic, the property need not be so, as it can be true of more than one program. The most natural way to describe the property is to use an LTL formula. Another possibility is to use a model generator (i.e., a Büchi sequential automaton) rather than an LTL formula.

Note that given Theorem 2.3 any property or program that can be described by an LTL formula can also be described by a Büchi automaton. However, strictly speaking, the converse is not true. To be able to give a formula corresponding to any automaton, one needs to extend LTL. One possibility is to use the extended temporal logic of [Wo83], [WVS83], [VW88]. Another possibility is to use additional propositions in the formula to encode the states of the automaton. The transitions of the automaton can then easily be described by LTL formulas. The problem with this approach is that the resulting formula conveys more information than is desired. Besides the description of the acceptable behaviors, it also includes the encoding of the states of the automaton. To hide this encoding, one can consider the propositions describing the states as existentially quantified. This takes us beyond LTL and towards *quantified linear time propositional temporal logic*. Quantified temporal logic has been studied in [Si83], [SVW87]. It has been shown that it is possible to build Büchi automata from quantified temporal logic, but the complexity of the algorithm is much higher than for LTL (it is nonelementary). Fortunately, as the preceding discussion suggests, one level of existential quantification is sufficient for encoding automata and in this case Theorem 2.3 still applies. This type of encoding of the states by existentially quantified formulas is the method usually used when one wants to represent a program by a temporal formula.

The verification problem is, given a description of a program and a property, to determine if the program (or equivalently its computation) satisfies the property. Depending on how we choose to describe the program and the property, we get various methods as shown in Table 2.

	Property	
Program	LTL Formula	Büchi Sequential Automaton
States and Transitions	Model Checking	Automaton Verification
LTL Formula	Axiomatic Verification	Reverse Model Checking

Table 2: Verification Methods for Deterministic Programs

We now give more details about the various methods.

Model Checking

In this approach, the verification problem is to check that the program which is described explicitely satisfies (i.e., is a model of) the specification. This method was introduced in [CE81], [CES83] for nondeterministic programs and BTL specifications, and was applied to nondeterministic programs and LTL specifications in [LP85] and [VW86b] (see section 3.2). It has not so far been advocated in the context of deterministic programs and LTL specifications, probably because in practice this is not an interesting case. Algorithmically, it is much simpler than the case of nondeterministic programs and LTL specifications. It can be solved with a simple polynomial time algorithm which is an adaptation of the algorithm used for doing model checking for nondeterministic programs and branching time logic (see [CES86], [EL85a], [EL85b]).

Automaton Verification

This is a rather recent approach that was introduced for nondeterministic programs in [AS85] and further developed in [AS87], [MP87]. In the case we are considering now, the problem is to determine if the unique computation of a deterministic program is a word accepted by the automaton describing the desired property of the program. This can be done by viewing the program as a deterministic automaton, taking the product of this automaton with the automaton describing the property and checking that the result is nonempty.

Axiomatic Verification

This is the earliest verification method based on temporal logic. It was introduced in [Pn77] and [Pn81] and was further developed in [MP81], [OL82], [MP83a], [MP83b], [MP83c] and [MP84]. It is usually presented for nondeterministic or concurrent programs, but is applicable as is to deterministic programs. Here, one verifies the program by proving in a suitable axiomatic system that the formula describing the program implies the formula describing the property.

Reverse Model Checking

This verification method has never been studied. It uses the rather unnatural approach of specifying the program by a formula and the property to be verified by an automaton. Algorithms for this problem can be obtained by techniques similar to the ones used in model checking. Basically, one starts by building the Büchi automaton corresponding to the program formula. If the program is deterministic, this automaton should also be deterministic. It is then sufficient to check that the intersection of this automaton and the automaton specifying the desired property is nonempty. We do not advocate reverse model checking, but include it only for the sake of presenting a complete picture of possible verification methods.

3.2 Second View: Nondeterministic Programs

The most common use of temporal logic is for the verification of nondeterministic programs or of concurrent programs where the concurrency is modeled by nondeterminism.

A nondeterministic program is a tuple (W, Q, w_0, π) where

- W is a set of states,

- Q is a transition relation,
- w_0 is a unique initial state,
- $\pi : W \rightarrow \Sigma$ is a labeling function.

The computations of such programs are infinite sequences of states $w_0, w_1, \ldots w_i, \ldots$ such that w_0 is the initial state of the program, and for all $i \geq 0$, $(w_i, w_{i+1}) \in Q$. A program can have many computations.

The most straightforward way to relate nondeterministic programs to temporal logic, is to view programs as interpretations for BTL formulas and computations as linear interpretations extracted from the branching interpretation. Given our definitions, this correspondence is immediate. Also, it is easy to see that if we consider the set of computations of a program and organize these into a tree, this tree will be equivalent to the program according to Definition 2.3. We will thus not distinguish between this tree and the program.

There is however a second way to relate nondeterministic programs to temporal logic. It consists of viewing the programs as model generators for LTL (i.e. as Büchi sequential automata) and the computations as models of LTL. This might require some clarification as the programs we have defined are different from Büchi automata in two respects: the labels are on states and not on edges and there is no set of accepting states. These differences are fortunately only superficial. Indeed, one can transform a nondeterministic program into a Büchi automaton by labeling all the edges leaving a state with the label of that state and by taking the set of accepting states to be the whole set of states. Notice that these conventions imply that the set of words accepted by the Büchi automaton associated with a program is exactly the set of computations of the program.

Given this correspondence between Büchi automata and programs, a question that occurs naturally is: why not allow a Büchi style acceptance condition on the program? This is indeed possible and is considered to be a way of describing a *fairness* condition on the execution of the program [CES86], [Fr86], [CVW86], [ACW87]. The set of computations of the program is then restricted to those computations that satisfy the acceptance condition, i.e., go infinitely often through one of the accepting states.

Pushing this line of thought one step further, one can think of interpreting BTL over structures that include a Büchi-style acceptance condition. This was done in [CVW86]. There, one interprets a state formula $\exists g$ ($\forall g$) as meaning "g is true on some (all) paths satisfying the acceptance condition". Although the resulting logic is syntactically identical to BTL, it has different valid formulas and hence a different decision procedure than BTL interpreted over structures without an acceptance condition. We will not study further this interpretation of branching time temporal logic, but we will note that the results we give below for BTL interpreted over structures without an acceptance condition also apply to the case of branching time logic interpreted over structures with an acceptance condition. Interestingly, with this interpretation, the similarity between the model generators of LTL and the models of BTL is complete.

Our choices for verifying nondeterministic programs using temporal logic are summarized in Table 3.

	Program	
Computation	LTL Model Generator	BTL Model
LTL Model	Linear Approach	Mixed Approach 1
BTL Model (Computation Tree)	Mixed Approach 2	Branching Approach

Table 3: Modeling Nondeterministic Programs

We will now examine in more details the linear and branching approaches to verifying nondeterministic programs. We will not consider further the mixed approaches. Indeed, they haven't been used as such and there is little to learn from their study.

3.2.1 The Linear Approach for Nondeterministic Programs

Let us recall that in this context, a program is viewed as a model generator for LTL and computations as models of LTL. The possible verification methods are summarized in Table 4.

	Property	
Program	LTL Formula	Büchi Sequential Automaton
States, Transitions and Acceptance Condition	Model Checking	Automaton Verification
LTL formula	Axiomatic Verification	Reverse Model Checking

Table 4: LTL-Based Verification for Nondeterministic Programs

Our options are actually identical to the ones we had for verifying deterministic programs using LTL. The difference between the two cases, is that finding algorithms for the various verification methods is more difficult for nondeterministic programs than for deterministic programs. Let us examine the known results.

Model Checking

Here, one has to check that all the computations of the program satisfy a given LTL formula. This problem was considered in [LP85] and [VW86b]. The result is that model checking can be done in time linear in the size of the program and exponential in the size of the formula. A very simple description of the algorithm can be given [VW86b]. One takes the negation $\neg f$ of the formula f and builds the corresponding Büchi automaton $A_{\neg f}$. This automaton is at most exponential in the size of the formula. The next step is to combine $A_{\neg f}$ with the automaton A_P corresponding to the program and check that the resulting automaton is empty. This last step can be done in linear time. Note that we build the automaton for $\neg f$ and not for f. This is because it is much easier to check that the intersection of two Büchi automata is empty than to check that the language generated by one automaton is included in the language generated by the other (a PSPACE-complete problem).

Automaton Verification

In this case, one has to check that all the computations of the program are accepted by the specification automaton. In other words, one has to verify that the language generated by the automaton corresponding to the program ($L(A_P)$) is a subset of the language accepted by the specification automaton ($L(A_{spec})$). Unfortunately, this is a PSPACE-complete problem. The usual algorithm to solve it is to build the complement $\overline{A}_{spec}$ of the automaton A_{spec} (this is the expensive part of the algorithm [SVW87]) and check that $A_P \cap \overline{A}_{spec} = \phi$.

In the papers where this method is considered, the algorithm is simplified by imposing some restriction on the automaton A_{spec}. In [AS85] and [AS87], the automaton is a deterministic automaton or a combination of deterministic automata. In [MP87], "forall automata" are used for specification. These automata are the dual of nondeterministic automata in the sense that they accept a word iff *all* the computations of the automaton on that word are accepting. This makes these automata easy to complement into nondeterministic automata and hence they can easily be used for automaton verification. Their disadvantage is that they are a rather unnatural way of describing properties of programs. Finally, let us mention that in [AS85], [AS87] as well as in [MP87], the method is applied to infinite-state programs and first-order specifications. In simple terms, the transitions of the automaton A_{spec} are labeled by first-order predicates on the states of the program and the verification is done by extracting *proof obligations* from the combination of the specification automaton and the program. These proof obligations are Hoare-like verification conditions on the transitions of the program that have to be satisfied for the program to meet its specification. However, the fundamental basis of the method is identical to what we have described in the finite-state case.

Axiomatic Verification

This verification method is identical to the corresponding one for deterministic programs. The only difference is that the formula specifying the program can have several models, each of which corresponds to a computation of the program.

Reverse Model Checking

An algorithm for this verification method can be obtained by building the automaton corresponding to the formula describing the program. This reduces the problem to the automaton verification problem.

3.2.2 The Branching Approach for Nondeterministic Programs

We now consider the case of the program as well as its computation tree being viewed as models of BTL. The potential verification methods are described in Table 5.

	Property	
Program	BTL Formula	Büchi Tree Automaton
States and Transitions	Model Checking	Automaton Verification
BTL formula	Axiomatic Verification	Reverse Model Checking

Table 5: BTL-Based Verification for Nondeterministic Programs

Model Checking

Model checking for BTL was introduced in [CE81] and [CES83]. The best algorithm for model checking for CTL is of linear complexity in both the size of the formula and of the model. For CTL*, the complexity in the size of the formula is exponential while the complexity in the size of the model remains linear [EL85b].

Automaton Verification

In this case, the problem is to determine if the computation tree of the program is accepted by the Büchi tree automaton describing the desired property. This can be done by viewing the program as a trivial deterministic tree automaton (accepting only one tree), taking the product of this automaton with the automaton for the property and checking that the result is nonempty. This yields a quadratic time algorithm since the emptiness problem for Büchi tree automata is solvable in quadratic time (Theorem 2.9).

Axiomatic Verification

Here, one uses an axiomatic system to prove that the formula specifying the program implies the specification formula. Note that as the computation tree of the program corresponds to a model of a BTL formula, the formula describing the program should only have one model up to equivalence.

Reverse Model Checking

We have to check that all the models of the formula describing the program are accepted by the automaton giving the specification. If, as should be the case, the program formula only describes one program, the corresponding tree automaton should only accept one tree and the problem can be reduced to the automaton verification problem we have just discussed.

3.3 Third View: Partially Ordered Computations

We would like to attract the reader's attention to the fact that the branching approach for nondeterministic programs is analogous to the application of LTL to deterministic programs. In both cases, programs and their set of computations correspond to models of the logic: in one case they are models of LTL and in the other of BTL. In this section, we will consider a use of BTL that is the analogue of the use of LTL for nondeterministic programs.

A number of authors [PW84], [Pr82], [KP87] have argued that when considering distributed programs, it is necessary to view computations as partial orders of local states rather than as total orders of global states. Formally, one can say that a partially ordered computation is a tuple (W, Q, w_0, π) where

- W is a set of states,
- Q is a transition relation,
- w_0 is an initial state,
- $\pi : W \rightarrow \Sigma$ is a labeling function.

and where the transitive closure of the relation Q is irreflexive. This last condition is not very important given that we do not wish to distinguish between computations that are equivalent according to Definition 2.3 and hence we can always unwind one that is not a partial order into one that is. The important characteristic of partially ordered computations is that once they are unwound, some states might be incomparable by the transitive closure of the relation Q. This is never the case in the totally ordered computations we have considered so far. A special case of a partially ordered computation to which we will pay special attention is that of an infinite tree.

If a computation is a partial order, one might wonder what a program is. The natural answer is that it is a structure generating partial orders, a special case of which is a tree automaton generating infinite trees. It is of course possible to consider more general structures generating arbitrary partial orders and not just trees. However, we will limit ourselves to the case of tree automata for two reasons. The first is simplicity, the second is that as we have shown, an arbitrary branching structure is always equivalent in the sense of Definition 2.5 to a labeled tree and such equivalent structures can not be distinguished by formulas of the logics we are considering.

To make things more concrete, let us relate our abstract definition of partially ordered computations to distributed programs. One can consider a state of a computation having several successors as corresponding to a *fork* (i.e. a process splitting into several processes) and a state having several predecessors as corresponding to a *join* (i.e. several processes synchronizing and merging into a single process). Now a program will potentially generate several partial orders as it might include some nondeterminism beyond the fork and join operations.

There have been two attempts to use temporal logic for reasoning about partially ordered computations. The first deals directly with the partial orders [PW84], the second first converts them into a tree of global states [KP87].

3.3.1 Applying POTL to Partially Ordered Computations

In [PW84], the logic POTL is applied to partially ordered computations. The connection between the logic and the programs is established by identifying the models of POTL with partially ordered computations. Given our definition of partially ordered computations, this identification is immediate. It should be contrasted with the branching approach to verifying these programs (Section 3.2.2) where a model of BTL corresponds to the *set* of computations (computation tree) of a program and not to a *single* computation as in this section. Once we have established that a computation is a model of POTL, a program is then a model generator for POTL which is an automaton on $\pm$ trees.

The possible verification methods are then those appearing in Table 6.

	Property	
Program	POTL Formula	Büchi Tree Automaton
Tree Automaton	Model Checking	Automaton Verification
POTL formula	Axiomatic Verification	Reverse Model Checking

Table 6: POTL-Based Verification

Model Checking

To solve the model checking problem for POTL and partially ordered computations, it is possible to use an automata-theoretic approach as in LTL-based model checking for nondeterministic programs. However, we will have to use tree automata rather than sequential automata, but the basic approach is the same. If the specification is f, we build the tree automaton for $\neg f$ and check that its intersection with the tree automaton describing the program is empty. There is however one difficulty which is to make sure that the branching factors of the program and the automaton for the formula are identical. A description of a model-checking algorithm for POTL can be found in [KP86]

Automaton Verification

Here, we have to check that all trees generated by the program automaton are accepted by the specification automaton. For this, one needs to complement the specification automaton and check that its intersection with the program automaton is empty. Unfortunately, Büchi tree automata are not closed under complementation. One thus needs a more general type of tree automata to solve this problem, for instance Rabin tree automata [Ra69]. However, the complementation algorithm for Rabin tree automata is quite intricate and expensive.

Axiomatic Verification

The axiomatic approach was the one advocated for verification in [PW84]. The program is described by a POTL statement and one then proves using an axiomatic system that this statement implies the specification.

Reverse Model Checking

This problem can be reduced to the automaton verification problem by building the tree automaton corresponding to the program formula.

3.3.2 Applying BTL to Partially Ordered Computations

In [KP87], one finds a different way of using temporal logic to deal with partially ordered computations. That approach starts by mapping the partially ordered computation into a tree of global states. The goal is to make it possible to describe both properties of the global state and of the partially ordered computation. An important point to remember is that *one* tree of global states corresponds to *one* partially ordered computation whereas in the case of nondeterministic programs, *one* tree corresponds to *many* computations (the computation tree of the program).

To verify programs in this framework, it is thus natural to relate the models of BTL to computations and its model generators (automata on infinite trees) to programs, similarly to what we did in the case of POTL. The difference is in the way the trees representing the computations are interpreted. The applicable verification methods are thus the same as in the POTL case and we will not reexamine them. In [KP87], the axiomatic method is proposed for verification. Note that from a purely logical point of view, it is exactly BTL that is used. Indeed, the only difference between this and the use of BTL for nondeterministic programs is that in the latter a model of BTL is considered as a *single* computation rather than as computation tree.

In [KP87], this method is pushed one step further. The idea is to develop a new logic that can talk simultaneously about several computations each of which is an infinite tree. This logic is called QISTL (*Quantified Interleaving Set Temporal Logic*) in [KP87] and can be thought of as a *branching branching* temporal logic (BBTL). It is to branching time logic what branching time logic is to linear time logic. To describe it schematically, one can say that it includes

- tree formulas interpreted over infinite trees, and
- state formulas that are either atomic propositions, tree quantifiers applied to tree formulas or boolean combinations of state formulas.

Its models are thus structures similar to tree automata or equivalently sets of trees. The model generators for this logic would have one more level of branching and have not yet been named.

To use BBTL for verification, one then takes a distributed program or its *set* of computations to be a model of BBTL. An interesting special case is when the only BBTL formula used are of the form $\forall_{tree} tf$ where tf is a BTL formula. In this case, a program (which is a set of trees) is verified if it satisfies the BBTL formula $\forall_{tree} tf$. Equivalently, all the trees (computations) of the program have to satisfy the BTL formula tf. This is exactly the verification that is done when applying BTL to partially ordered computations.

3.4 Summary

As a summary, we have collected in Table 7 the various connections established between models, model generators, programs and computations.

For each of the connections, we obtain verification methods by representing the program and the specification by either an automaton or a formula. Finally, let us note that the most common method to obtain an algorithm for a verification method is to convert whatever is given as a formula into an automaton using the results of Section 2. One then "only" has to deal with an automata-theoretic problem.

4 Program Synthesis

Here we examine the applications of temporal logic to program synthesis. In general, program synthesis is building an executable program from a non-executable specification. In our case the non-executable specification will be a propositional temporal logic formula and the synthesis method will be based on the correspondence between formulas and their models or model generators. Clearly, only a limited class of programs can be specified in propositional temporal

	View 1	View 2 (linear)	View 2 (branching)	View 3
Linear Model	*Computation* Program	*Computation*		
Linear Generator		*Computation Tree* Program		
Branching Model			*Computation Tree* Program	*Computation*
Branching Generator				Program

Table 7: Summary of Verification Approaches

logic (in fact finite-state programs or a subset thereof depending on the version of propositional temporal logic that is used). However, it has been argued [Wo82], [EC82], [MW84] that this type of synthesis is useful if the target is the synchronization part of a concurrent program. Indeed, synchronization code is usually finite state but is often complex and hard to write correctly.

We now review the various synthesis methods that have been proposed. Our classification of these methods follows the same lines as our classification of verification methods.

4.1 First View: Deterministic Programs

In Section 3.1, we noticed that a deterministic program is the same type of structure as a model of LTL. Thus, if one specifies a program by an LTL formula, a deterministic program whose only computation satisfies that formula can be constructed by building a model of the formula. Building a model is done by first building the model generator (Theorem 2.3) and then extracting a model from this model generator. This method was proposed in [Wo82] and [MW84].

4.2 Second View: Non-Deterministic Programs

Here we have the same choices as in Section 3.2. A non-deterministic program can be regarded either as a model of a BTL formula or as a model generator for an LTL formula. The first approach was taken in [EC82] and the second in [Wo82], [MW84].

To synthesize a program from a BTL specification, one thus constructs a model of this specification. This can be done by first building the tree automata that generates the models of the specification (Theorem 2.8), and then extracting a model from this automaton. In [EC82], the synthesis algorithm is described in terms of *semantic tableaux*. However, there is no essential difference between this algorithm and the one obtained by building the tree automaton for the formula and extracting a model from that automaton.

If one uses an LTL specification, the synthesis algorithm is then to build the sequential Büchi automaton that generates all the models of the specification. This can be done using Theorem 2.3. However, once this automaton is built, one problem remains to be solved: how does one implement the requirement that the only legal computations of the program are those that go infinitely often through some designated state of the Büchi automaton? In [Wo82] and [MW84], this problem was solved by considering the acceptance condition of the Büchi automaton as a fairness condition [Fr86]. The program was then split up into processes in such a way that it would satisfy this condition if the processes were executed under a reasonable fairness assumption. Finally, let us note that it is also possible to synthesize a program with a fairness condition from BTL if one uses the logic of [CVW86].

Acknowledgement

I wish to thank M. Baudinet, P. Gribomont, V. Lovinfosse and M. Vardi for reading drafts of this paper and providing helpful comments.

References

[ACW87] S. Aggarwal, C. Courcoubetis, P. Wolper, "Adding Liveness Properties to Coupled Finite-State Machines", to appear.

[AS85] B. Alpern, F. Schneider, "Verifying Temporal Properties Without Using Temporal Logic", Technical Report TR 85-723, Dept. of Computer Science, Cornell University, December 1985.

[AS87] B. Alpern, F. Schneider, "Proving Boolean Combinations of Deterministic Properties", *Proc. Symp. on Logic in Computer Science,* Ithaca, June 1987, pp. 131–137.

[BMP81] M. Ben-Ari, Z. Manna, A. Pnueli, "The Logic of Nexttime", *Eighth ACM Symposium on Principles of Programming Languages*, Williamsburg, January 1981, pp. 164–176.

[BKP84] H. Barringer, R. Kuiper, A. Pnueli, "Now You May Compose Temporal Logic Specifications", *Proc. 16th ACM Symp. on Theory of Computing,* Washington, April 1984, pp. 51–63.

[BKP85] H. Barringer, R. Kuiper, A. Pnueli. "A Compositional Temporal Approach to a CSP-like Language", *Proceedings of the IFIP Working Conference "The Role of Abstract Models in Information Processing"*, E.J. Neuhold and G. Chroust, editors, North Holland, Vienna, 1985, pp. 207–227.

[BKP86] H. Barringer, R. Kuiper, A. Pnueli, "A Really Abstract Concurrent Model and its Temporal Logic", *Proceedings of the Thirteenth ACM Symposium on the Principles of Programming Languages*, St. Petersberg Beach, Florida, January 1986, pp. 173–183.

[Bu62] J. R. Büchi, "On a Decision Method in Restricted Second Order Arithmetic", *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, Stanford University Press, 1962, pp. 1–12.

[CE81] E. M. Clarke, E. A. Emerson, "Synthesis of Synchronization Skeletons from Branching Time Temporal Logic", *Proceedings of the Workshop on Logics of Programs*, Yorktown-Heights, NY, Lecture Notes in Computer Science vol. 131, Springer-Verlag, Berlin, 1982, pp. 52–71.

[CES83] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic Verfication of Finite-state Concurrent Systems Using Temporal Logic Specifications: A Practical Approach", *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1984, pp. 117–126.

[CES86] E. M. Clarke, E. A. Emerson, A. P. Sistla, "Automatic Verfication of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems,* vol. 8, no. 2 (1986), pp. 244–263.

[CGB86] E.M. Clarke, O. Grumberg, M.C. Browne, "Reasoning about Networks with Many Identical Finite-State Processes", *Proc. 5th ACM Symp. on Principles of Distributed Computing*, Minaki, Ontario, August 1985, pp. 240–248.

[CVW86] C. Courcoubetis, M. Y. Vardi, P. Wolper, "Reasoning about Fair Concurrent Programs", *Proc. 18th Symp. on Theory of Computing*, Berkeley, May 1986, pp. 283–294.

[EC82] E. A. Emerson, E. M. Clarke, "Using Banching Time Logic to Synthesize Synchronization Skeletons", *Science of Computer Programming*, **2** (1982), pp. 241–266.

[EH85a] E. A. Emerson, J. Y. Halpern, "Decision Procedures and Expressiveness in the Temporal Logic of Branching Time", *Journal of Computer and System Sciences,* **30** (1985), pp. 1–24.

[EH85b] E.A. Emerson, J.Y. Halpern, " "Sometimes" and "Not Never" Revisited: On Branching versus Linear Time Temporal Logic", *Journal of the ACM,* vol. 33, no. 1 (1986), pp. 151–178.

[EL85a] E. A. Emerson, Ching-Laung Lei, "Temporal Model Checking under Generalized Fairness Constraints", *Proc. 18th Hawaii International Conference on System Sciences,* 1985.

[EL85b] E. A. Emerson, Ching-Laung Lei, "Modalities for Model Checking: Branching Time Strikes Back", *Proc. 12th ACM Symposium on Principles of Programming Languages,* New Orleans, January 1985, pp. 84–96.

[Em85] E. A. Emerson, "Automata, Tableaux and Temporal Logics", *Proc. Workshop on Logic of Programs,* Brooklyn 1985, Lecture Notes in Computer Science, vol. 193, Springer-Verlag, pp. 79–88.

[ES84] E. A. Emerson, A. P. Sistla, "Deciding Branching Time Logic", *Information and Control,* **61** (1984), pp. 175–201.

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

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah and J. Stavi, "The Temporal Analysis of Fairness", *Seventh ACM Symposium on Principles of Programming Languages*, Las Vegas, January 1980, pp. 163–173.

[HMM83] J. Halpern, Z. Manna, B. Moszkowski, "A Hardware Semantics Based on Temporal Intervals", *Proc. 10th International Colloquium on Automata Languages and Programming*, Lecture Notes in Computer Science, vol. 154, Springer-Verlag, Berlin, 1983.

[KP86] Y. Kornatsky, S. S. Pinter, "A Model Checker for Partial Order Temporal Logic", EE PUB no. 597, Department of Electrical Enginering, Technion-Israel Institute of Technology, June 1986.

[KP87] S. Katz, D. Peled, "Interleaving Set Temporal Logic", *Proc. 6th ACM Symp. on Principles of Distributed Computing,* Vancouver, August 1987, pp. 178–190.

[La80] L. Lamport, "Sometimes is Sometimes Not Never", *Seventh ACM Symposium on Principles of Programming Languages,* Las Vegas, NV, January 1980, pp. 174–185.

[LP85] O. Lichtenstein, A. Pnueli, "Checking that Finite State Concurrent Programs Satisfy their Linear Specifications", *Proc. 12th ACM Symposium on Principles of Programming Languages,* New Orleans, January 1985, pp. 97–107.

[LPZ85] O. Lichtenstein, A. Pnueli, L. Zuck, "The Glory of the Past", *Proc. Workshop on Logic of Programs,* Brooklyn 1985, Lecture Notes in Computer Science, vol. 193, Springer-Verlag, pp. 196–218.

[Mi80] R. Milner, "A Calculus of Communicating Systems", Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlin, 1980.

[Mo83] B. Moszkowski, "Reasoning about Digital Circuits", Ph.D. Thesis, Department of Computer Science, Stanford University, July 1983.

[MP81] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: the Temporal Framework", *The Correctness Problem in Computer Science,* R. S. Boyer and J S. Moore, eds., International Lecture Series in Computer Science, Academic Press, London, 1981, pp. 215–273.

[MP83a] Z. Manna, A. Pnueli, "How to Cook a Temporal Proof System for your Pet Language, *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, Austin, January 1984, pp. 141–154.

[MP83b] Z. Manna, A. Pnueli, "Proving Properties: the Temporal Way", *Proc. 10th International Colloquium on Automata Languages and Programming,* Lecture Notes in Computer Science, vol. 154, Springer-Verlag, Berlin, 1983, pp. 491–512.

[MP83c] Z. Manna, A. Pnueli, "Verification of Concurrent Programs: A Temporal Proof System", Foundations of Computer Science IV, J. W. deBakker, J. Van Leeuwen, Eds., *Mathematical Center Tracts*, vol. 159, Amsterdam, 1983, pp. 163-225.

[MP84] Z. Manna, A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs", *Science of Computer Programming,* 4 (1984), pp. 257–289.

[MP87] Z. Manna, A. Pnueli, "Specification and Verification of Concurrent Programs by ∀-Automata", *Proc. 14th ACM Symp. on Principles of Programming Languages,* Munich, January 1987, pp. 1–12.

[MW84] Z. Manna, P. Wolper, "Synthesis of Communicating Processes from Temporal Logic Specifications", *ACM Transactions on Programming Languages and Systems,* vol. 6, no. 1, January 1984, pp. 68–93.

[NGO85] V. Nguyen, D. Gries, S. Owicki, "A Model and Temporal Proof System for Networks of Processes", *Proc. 12th ACM Symp. on Principles of Programming Languages,* New Orleans, January 1985, pp. 121–131.

[OL82] S. Owicki, L. Lamport, "Proving Liveness Properties of Concurrent Programs", *ACM Transactions on Programming Languages and Systems,* vol. 4, no. 3, 1982, pp. 455–496.

[Pn77] A. Pnueli, "The Temporal Logic of Programs", *Proceedings of the Eighteenth Symposium on Foundations of Computer Science*, Providence, RI, November 1977, pp. 46–57.

[Pn81] A. Pnueli, "The Temporal Logic of Concurrent Programs", *Theoretical Computer Science,* 13(1981), pp. 45–60.

[Pr82] V. R. Pratt, "On the Composition of Processes", *Ninth Symposium on Principles of Programming Languages,* Albuquerque, January 1982, pp. 213–223.

[PW84] S.S. Pinter, P. Wolper, "A Temporal Logic for Reasoning about Partially Ordered Computations", *Proc. 3rd ACM Symposium on Principles of Distributed Computing,* Vancouver, August 1984, pp. 28–37.

[PZ86] A. Pnueli, L. Zuck, "Probabilistic Verification by Tableaux", *Proc. Symp. on Logic in computer Science,* Cambridge, June 1986, pp. 322–331.

[Ra69] M. O. Rabin, "Decidability of Second Order Theories and Automata on Infinite Trees", *Transactions of American Mathematical Society,* vol. 141, July 1969, pp. 1–35.

[Ra70] M. O. Rabin, "Weakly Definable Relations and Special Automata", *Proc. Symp. on Mathematical Logic and Foundations of Set Theory,* Y. Bar-Hillel, ed., North Holland, Amsterdam, 1970, pp. 1–23.

[Si83] A.P. Sistla, "Theoretical Issues in the Design and Verification of Distributed Systems", PhD thesis, Harvard University, Harvard, 1983.

[SC82] A. P. Sistla, E. M. Clarke, "The Complexity of Propositional Linear Temporal Logic", *Proceedings of the 14th ACM Symposium on Theory of Computing,* San Francisco, CA, May 1982, pp. 159–168.

[SC85] A. P. Sistla, E. M. Clarke, "The Complexity of Propositional Linear Temporal Logics", *Journal of the ACM,* vol. 32, no. 3, July 1985, pp. 733-749.

[SMV83] R. L. Schwartz, P. M. Melliar-Smith, F. H. Vogt, "An Interval Logic for Higher-Level Temporal Reasoning", *Proc. 2nd ACM Symposium on Principles of Distributed Computing,* Montreal, Canada, August 1983, pp. 173–186.

[SVW87] A. P. Sistla, M. Y. Vardi, P. Wolper, "The Complementation Problem for Büchi Automata with Applications to Temporal Logic", *Theoretical Computer Science,* 49 (1987), pp. 217–237.

[Va85a] M. Vardi, "Automatic Verification of Probabilistic Concurrent Finite-State Programs", *Proc. 26th Symp. on Foundations of Computer Science,* Portland, October 1985, pp. 327–338.

[Va85b] M. Vardi, The Taming of Converse: Reasoning about Two-Way Computations", *Proc. Workshop on Logic of Programs,* Brooklyn 1985, Lecture Notes in Computer Science, vol. 193, Springer-Verlag, pp. 413–424.

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

[VW86a] M. Y. Vardi, P. Wolper, "Automata-Theoretic Techniques for Modal Logics of Programs", *Journal of Computer and System Science,* vol. 32, no. 2 (April 1986), pp. 183–21.

[VW86b] M. Y. Vardi, P. Wolper, "An Automata-Theoretic Approach To Automatic Program Verification", *Proc. Symp. on Logic in Computer Science,* Cambridge, June 1986, pp. 322–331.

[VW88] M. Y. Vardi, P. Wolper, "Reasoning about Infinite Computation Paths", to appear.

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

[Wo83] P. Wolper, "Temporal Logic Can Be More Expressive", *Information and Control,* vol. 56, nos. 1–2, 1983, pp. 72–99.

[WVS83] P. Wolper, M. Y. Vardi, A. P. Sistla, "Reasoning about Infinite Computation Paths", *Proc. 24th IEEE Symposium on Foundations of Computer Science,* Tucson, 1983, pp. 185–194.