Sometimes and Not Never Re-revisited: On Branching Versus Linear Time

Moshe Y. Vardi[⋆]

Rice University, Department of Computer Science, Houston, TX 77005-1892, USA

Abstract. The difference in the complexity of branching and linear model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking makes CTL a popular choice, leading to efficient model-checking tools for this logic. Can we use these tools in order to verify linear properties? In this survey paper[1] we describe two approaches that relate branching and linear model checking. In the first approach we associate with each LTL formula ψ a CTL formula ψ_A that is obtained from ψ by preceding each temporal operator by the universal path quantifier A. In particular, we characterize when ψ is logically equivalent to ψ_A. Our second approach is motivated by the fact that the alternation-free μ-calculus, which is more expressive than CTL, has the same computational advantage as CTL when it comes to model checking. We characterize LTL formulas that can be expressed in the alternation-free μ-calculus; these are precisely the formulas that are equivalent to deterministic Büchi automata. We then claim that these results are possibly of theoretical, rather than of practical interest, since in practice, LTL model checkers seem to perform rather nicely on formulas that are equivalent to CTL or alternation-free μ-calculus formulas.

1 Introduction

Temporal logics, which are modal logics geared towards the description of the temporal ordering of events, have been adopted as a powerful tool for specifying and verifying concurrent programs [36]. One of the most significant developments in this area is the discovery of algorithmic methods for verifying temporal-logic properties of *finite-state* programs [6,43,35,7,52]. This derives its significance both from the fact that many synchronization and communication protocols can be modeled as finite-state programs, as well as from the great ease of use of fully algorithmic methods. Finite-state programs can be modeled by transition systems where each state has a bounded description, and hence can be characterized by a fixed number of boolean atomic propositions. In temporal-logic *model checking*, we verify the correctness of a finite-state program with respect to a desired behavior by checking whether a labeled transition system that models the program satisfies a temporal logic formula that specifies this behavior (for a survey, see [8]).

Two possible views regarding the nature of time induce two types of temporal logics [34]. In *linear* temporal logics, time is treated as if each moment in time has a unique possible future. Thus, linear temporal logic formulas are interpreted over linear sequences and we regard them as describing a behavior of a single computation of a program. In *branching* temporal logics, each moment in time may split into various possible futures. Accordingly, the structures over which branching temporal logic formulas are interpreted can be viewed as infinite computation trees, each describing the behavior of the possible computations of a nondeterministic program.

In the linear temporal logic LTL, formulas are composed from the set of atomic propositions using the usual Boolean connectives as well as the temporal connective G (“always”),

[⋆] Supported in part by NSF grants CCR-9628400 and CCR-9700061, and by a grant from the Intel Corporation. URL: `http://www.cs.rice.edu/~vardi`.

[1] This paper is based on work reported in [30,31].

F ("eventually"), X ("next"), and U ("until"). The branching temporal logic CTL* augments LTL by the path quantifiers E ("there exists a computation") and A ("for all computations"). The branching temporal logic CTL is a fragment of CTL* in which every temporal connective is preceded by a path quantifier. Finally, the branching temporal logic $\forall$CTL is a fragment of CTL in which only universal path quantification is allowed. For an LTL formula ψ, we denote by ψ_A the $\forall$CTL formula obtained from ψ by preceding each temporal connective by the path quantifier A. For example, if ψ is $G(\text{req} \rightarrow F\text{grant})$, then ψ_A above is $AG(\text{req} \rightarrow AF\text{grant})$.

The discussion of the relative merits of linear versus branching temporal logics goes back to 1980 [34,11,1,42,14,12,5]. As analyzied in [42] linear and branching time logics correspond to two distinct views of time. It is not surprising therefore that LTL and CTL are expressively incomparable [34,12,5]. On the other hand, CTL seems to be superior to LTL when it comes to algorithmic verification, as we now explain.

Given a transition system M and a linear temporal logic formula ψ, the model-checking problem for M and ψ is to decide whether ψ holds in all the computations of M. When ψ is a branching temporal logic formula, the problem is to decide whether ψ holds in the computation tree of M. The complexity of model checking for both linear and branching temporal logics is well understood: suppose we are given a transition system of size n and a temporal logic formula of size m. For the branching temporal logic CTL, model-checking algorithms run in time $O(nm)$ [7], while, for the linear temporal logic LTL, model-checking algorithms run in time $n2^{O(m)}$ [35]. Since LTL model checking is PSPACE-complete [48], the latter bound probably cannot be improved.

The difference in the complexity of linear and branching model checking has been viewed as an argument in favor of the branching paradigm. In particular, the computational advantage of CTL model checking over LTL model checking makes CTL a popular choice, leading to efficient model-checking tools for this logic [8]. Nevertheless, designers often prefer to specify their systems using linear-time formalisms. Indeed, model-checking tools such as COSPAN, SPIN, and VIS [33,20,22,3] handle specifications that are given as automata on infinite words or LTL formulas. This raises the question whether we can use branching-time tools in order to verify linear properties. A straightforward relation between LTL and CTL model checking follows from the fact that LTL model checking can be reduced to the language-containment problem [52], which itself can be reduced to searching for fair paths, and hence to CTL model checking [52,4]. Such an approach, however, involves the definition of a new transition system whose size is exponential in the length of the LTL formula. As such, it does not enjoy the computational advantage of CTL. The real challenge is to relate branching-time and linear-time model checking in a practical way, one that would enable us to use branching-time model-checking tools in order to perform efficient model checking on some fragment of LTL.

A straightforward approach is as follows: given a transition system M and an LTL formula ψ to be checked with respect to M, try to translate the CTL* formula $A\psi$ to an equivalent CTL formula φ, and then check M with respect to φ. This approach has two drawbacks. First, the problem of deciding whether $A\psi$ has an equivalent CTL formula φ is still open, and so, of course, is the harder problem of constructing φ in cases it exists. Second, even when such φ exists, it may be substantially longer than ψ, making the whole effort useless. A partial success for this approach is presented in [25,47], which identify fragments of CTL* that can be easily translated in to CTL.

We describe here a more modest approach, due to [31]: instead looking for some equivalent CTL formula to $A\psi$, we restrict ourselves to the specific candidate ψ_A. This approach is weaker, in the sense that $A\psi$ may have an equivalent CTL formula and still not be equivalent to ψ_A. For example, $\psi = (Xp) \vee (Xq)$ is not equivalent to $\psi_A = (AXp) \vee (AXq)$, yet is equivalent to the CTL formula $AX(p \vee q)$. Nevertheless, this approach does not suffer from the drawbacks mentioned above: we know how to decide whether $A\psi$ and ψ_A are equivalent, the construction of ψ_A from ψ is straightforward, and the length of ψ_A is at most double the length of ψ. We thus focus on the problem of deciding whether $A\psi$ and ψ_A are equivalent: the problem can be solved in EXPSPACE and is PSPACE-hard. The conjecture in [31] is that the problem can be solved in polynomial space, which would match the lower bound. Unfortunately, even if this conjecture is correct, the complexity of checking the equivalence of $A\psi$ and ψ_A is too high to make this approach useful. Thus, this approach does not seem very appealing from a practical perspective.

In our second approach we attempt to relate LTL to the *alternation-free μ-calculus*. Typically, symbolic model-checking tools proceed by computing fixed-point expressions over the model's set of states. For example, to find the set of states from which a state satisfying some predicate p is reachable, the model checker starts with the set S of states in which p holds, and repeatedly add to S the set EXS of states that have a successor in S. Formally, the model checker calculates the least fixed-point of the expression $S = p \vee EXS$. The *μ-calculus* is a logic that contains the modal connectives EX and AX, and the fixed-point operators μ and ν [23]. As such, it describes fixed-point computations in a natural form. In particular, the *alternation-free* fragment of the μ-calculus (AFMC, for short) [15] has a restricted syntax that does not allow nesting of fixed-point operators, making the evaluation of expressions very simple, though still more expressive than CTL [15]. Formally, the model-checking problem for AFMC can be solved in time that is linear in both the size of the model and the length of the formula [9]. Thus, AFMC enjoys the same computationally attractive properties of CTL, in spite of its increased expressiveness.

When, as in the model-checking tools SMV and VIS [37,3], the specification language is the *branching-time* temporal logic CTL, the transition from the input formulas to fixed-point expressions is simple: each connectives of CTL can be expressed also by means of fixed points. Formally, one can translate a CTL formula to an AFMC formula with a linear blow up. For example, the CTL formula $AGEFp$ is equivalent to the AFMC formula $\nu Q.(\mu Y.p \vee EXY) \wedge AXQ$. Since linear-time formalisms such as LTL can express properties that are not expressible in AFMC (e.g., it follows from the results in [44,40,2], that the LTL formula FGp is not expressible in AFMC), symbolic model-checking methods become more complicated. For example, symbolic model checking of LTL involves a translation of LTL formulas to μ-calculus formulas of alternation depth 2, where nesting of fixed-point operators is allowed [15]. The evaluation of such μ-calculus formulas takes time that is quadratic in the size of the model. Since the models are very large, the difference with the linear complexity of AFMC is very significant [21].

We consider the problem of translating linear-time formalisms to AFMC; we describe a characterization, due to [30], of LTL formulas ψ such that the $\forall$CTL* formula $A\psi$ is equivalent to an AFMC formula. We also describe an algorithm, due to [30], of deciding whether a given formula meets this characterization, and the problem of translating a given formula to an equivalent AFMC formula when such a translation exists. The characterization asserts that an LTL formula can be translated to an AFMC formula iff it is equivalent to a deterministic Büchi word automaton. Unfortunately, the best known algorithm for checking whether an LTL formula is equivalent to a deterministic Büchi automaton is doubly exponential, and the translation of an LTL formula to an equivalent AFMC formula has an exponential lower bound. Thus, this approach is also not appealing from a practical perspective.

So far neither of our proposed approaches suggest a method that is guaranteed to perform better than usual LTL model checkers. Our motivation, recall, is the computational advantage of branching time over linear time. This advantage, however, refers to worst-case complexity. In the final part of the paper we claim that, in practice, standard LTL model checkers seem to perform quite nicely on typical LTL formulas. In fact, linear-time model checkers do essentially the same computation as branching-time model checkers on LTL formulas that are equivalent to CTL or AFMC formulas. We substantiate this claim by comparing the behavior of the bottom-up branching-time model-checking algorithms of [7,9] with the behavior of the automata-based LTL model-checking algorithm of [52] on some examples. Thus, in practice, it is not clear that there is real need to relate linear-time and branching-time model checking.

2 Preliminaries

2.1 Temporal logics

We review the syntax and semantics of temporal logics briefly; for more details see [10,36].

The logic *CTL** is a branching temporal logic. Formulas of CTL* are defined with respect to a set AP of atomic propositions and describe the evolution of these propositions. We present here a positive normal form for CTL* formulas where negation may be applied only to atomic propositions. In this form, a path quantifier, E ("for some path") or A ("for all paths"), can prefix an assertion built from the set of atomic propositions and their negations using the positive Boolean connectives and the temporal connectives X ("next time"), U ("until"), and $\tilde{U}$ ("dual of until"). There are two types of formulas in CTL*: *state formulas*, whose satisfaction is related to a specific state, and *path formulas*, whose satisfaction is related to a specific path. The logic *CTL* is a restricted subset of CTL*. In CTL, the temporal operators X, U, and $\tilde{U}$ must be immediately preceded by a path quantifier. Formally, it is the subset of CTL* obtained by restricting the path formulas to be $X\psi$, $\psi U \varphi$, or $\psi \tilde{U} \varphi$, where ψ and φ are CTL state formulas. The logic $\forall$CTL* is a restricted subset of CTL* that allows only the universal path quantifier A. The logic $\forall$CTL is defined similarly, as the restricted subset of CTL that allows the universal path quantifier only. The logics $\exists$CTL* and $\exists$CTL are defined analogously, as the existential fragments of CTL* and CTL, respectively. LTL formulas are constructed from Boolean and temporal connectives. An LTL formula ψ corresponds to a CTL* formula $A\psi$ in which ψ is a formula with no path quantifiers. Hence, LTL can be viewed as a fragment of CTL*. For an LTL formula ψ, we denote by ψ_A the $\forall$CTL formula obtained from ψ by preceding each temporal operator by the universal path quantifier A. For example, $(Xp \vee Xq)_A = AXp \vee AXq$. The formula ψ_E is defined similarly for the existential path quantifier E. We denote the size of a formula φ by $|\varphi|$ and we use the following abbreviations in writing formulas: $\rightarrow$ and $\leftrightarrow$, interpreted in the usual way, $F\psi = \mathbf{true}U\psi$ ("eventually"), and $G\psi = \neg F \neg \psi$ ("always").

We define the semantics of CTL* and its sublanguages with respect to *Rabin systems* (*systems*, for short). A system $M = \langle AP, W, W_0, R, L, \alpha \rangle$ consists of a set AP of atomic propositions, a set W of states, a set $W_0 \subseteq W$ of initial states, a total transition relation $R \subseteq W \times W$ (i.e., for every state $w \in W$, there exists at least w' with $R(w, w')$), a labeling function $L : W \rightarrow 2^{AP}$, and a Rabin fairness condition $\alpha \subseteq 2^W \times 2^W$, which defines a subset of W^ω. The system M is of *branching degree 1* iff for every state $w \in W$, there exists a single w' with $R(w, w')$. A *computation* of a system is an infinite sequence of states, $\pi = w_0, w_1, \ldots$ such that for every $i \geq 0$, we have that $R(w_i, w_{i+1})$. When $w_0 \in W_0$, we say that π is an *initialized computation*. For a computation π, let *Inf*(π) denote the set of states that repeat infinitely often in π. That is, *Inf*$(\pi) = \{w :$ for infinitely many $i \geq 0,$ we have $w_i = w\}$. A computation of M is *fair* iff it satisfies the fairness condition. Thus, if the fairness condition is $\alpha = \{\langle G_1, B_1 \rangle, \ldots, \langle G_k, B_k \rangle\}$, then π is fair iff there exists $1 \leq i \leq k$ for which *Inf*$(\pi) \cap G_i \neq \emptyset$ and *Inf*$(\pi) \cap B_i = \emptyset$. In other words, iff π visits G_i infinitely often and visits B_i only finitely often. Each computation $\pi = w_0, w_1, \ldots$

of M induces a *trace* $L(\pi) = L(w_0), L(w_1), \ldots$ in $(2^{AP})^\omega$. We sometimes refer to the *language*, $\mathcal{L}(M)$, of a system M, meaning the set of traces induced by M's initialized fair computations. We say that M is *nonempty* if $\mathcal{L}(M) \neq \emptyset$. For a system M and a state $w \in W$, we use M^w to denote the system obtained from M by taking the set of initial states to be the singleton $\{w\}$. For simplicity, we assume that each initial state $w_0 \in W_0$ is the first state of at least one initialized fair computation. Thus, M^{w_0} is nonempty.

With respect to a system M, we use $w \models \varphi$ to indicate that a state formula φ holds at state w, and we use $\pi \models \varphi$ to indicate that a path formula φ holds at path π. A system M satisfies a formula ψ iff ψ holds in all the initial states of M. The *model-checking problem* is to decide, given M and ψ, whether $M \models \psi$.

2.2 Alternation Free μ-Calculus

The *alternation-free μ-calculus* (*AFMC*, for short) is a fragment of the modal μ-calculus [23]. We define the AFMC by means of equational blocks, as in [9]. Formulas of AFMC are defined with respect to a set AP of atomic propositions and a set Var of atomic variables. A *basic* AFMC formula is either p, X, $\varphi \vee \psi$, $\varphi \wedge \psi$, $AX\varphi$, or $EX\varphi$, for $p \in AP$, $X \in \mathsf{Var}$, and basic AFMC formulas φ and ψ. The semantics of the basic formulas is defined with respect to a system $S = \langle AP, W, W_{in}, R, L\rangle$ without a fairness condition and a *valuation* $\mathcal{V} = \{\langle P_1, W_1\rangle, \ldots, \langle P_n, W_n\rangle\}$ that assigns subsets of W to the variables in Var. Each basic AFMC formula defines a subset of the states of S in the standard way. We denote by $\varphi^S_\mathcal{V}$ the set of states define by φ under the evaluation $\mathcal{V}$. For example, $p^S_\mathcal{V} = \{w \in W : p \in L(w)\}$, ${X_i}^S_\mathcal{V} = W_i$, $(\varphi \vee \psi)^S_\mathcal{V} = \varphi^S_\mathcal{V} \cup \psi^S_\mathcal{V}$, and $(EX\varphi)^S_\mathcal{V} = \{w \in W : \exists w' \in \varphi^S_\mathcal{V}$ and $R(w, w')\}$. An *equational block* has two forms, $\nu\{E\}$ or $\mu\{E\}$, where E is a list of equations of the form $P_i = \varphi_i$, where φ_i is a basic AFMC formula and the P_i are all different atomic variables. An atomic variable P that appears in the right-hand size of an equation in some block B may appear in the left-hand side of an equation in some other block B'. We then say that P is *free* in B and that B *depends* on B'. Such dependencies cannot be circular. This ensures that the formula is free of alternations. The semantics of an equational block is defined with respect to a system S and a valuation $\mathcal{V}$ that assigns subsets of W to all the free variables in the block. A block of the form $\nu\{E\}$ represents the greatest fixed-point of E and a block of the form $\mu\{E\}$ represents the least fixed-point of E. For example, $\nu\{P = p \wedge EXP\}$ defines the set of states in S from which there exists a computation in which p always holds. For a set of blocks, evaluation proceeds so that whenever a block B is evaluated, all the blocks B' for which B depends on B' are already evaluated, thus all the free variables in B have values in $\mathcal{V}$. A designated variable, say P_0, holds the final value of the formula. For the full definition see [9].

Given a system S and an AFMC formula φ, the model-checking problem is to determine whether all the initial states of S satisfy φ, that is, whether $W_{in} \subseteq \varphi^S_\emptyset$.

2.3 Simulation relation and maximal models

In [18], Grumberg and Long define an order relation between system that captures what it means for a system M' to have "more behaviors" than a system M. The definition in [18] extend previous definitions [39], which relate systems with no fairness conditions. Let $M = \langle AP, W, R, W_0, L, \alpha\rangle$ and $M' = \langle AP, W', R', W'_0, L', \alpha'\rangle$ be two systems and let w and w' be states in W and W', respectively. A relation $H \subseteq W \times W'$ is a *simulation relation* from $\langle M, w\rangle$ to $\langle M', w'\rangle$ iff the following conditions hold:

(1) $H(w, w')$.

(2) For all s and s', we have that $H(s, s')$ implies the following:

(2.1) $L(s) \cap AP' = L(s')$.

(2.2) For every fair computation $\pi = s_0, s_1, \ldots$ in M, with $s_0 = s$, there exists a fair computation $\pi' = s'_0, s'_1, \ldots$ in M', with $s'_0 = s'$, such that for all $i \geq 0$, we have $H(s_i, s'_i)$.

A simulation relation H is a *simulation from* M *to* M' iff for every $w \in W_0$, there exists $w' \in W'_0$ such that $H(w, w')$. If there exists a simulation from M to M', we say that M' *simulates* M and we write $M \leq M'$. Intuitively, it means that the system M' has more behaviors than the system M. In fact, every possible behavior of M is also a possible behavior of M'.

Theorem 1. [18,32] *For every M and M' such that $M \leq M'$, and for every $\forall$CTL* formula φ, we have that $M' \models \varphi$ implies $M \models \varphi$.*

A system M is a *maximal model* for an $\forall$CTL* formula φ if it allows all behaviors consistent with φ. Formally, M_φ is a maximal model of φ if $M_\varphi \models \varphi$ and for every system M we have that $M \leq M_\varphi$ if $M \models \varphi$. Note that by the preceding theorem, if $M \leq M_\varphi$, then $M \models \varphi$. Thus, M_φ is a maximal model for φ if for every system M, we have that $M \leq M_\varphi$ iff $M \models \varphi$. Let φ and ψ be two $\forall$CTL* formulas. The implication $\varphi \rightarrow \psi$ is valid iff ψ holds in all systems M in which φ holds. Since the more behaviors M has the less likely it is to satisfy ψ, it makes sense to examine the implication by checking ψ in a maximal model of φ:

Theorem 2. [18] *Let φ and ψ be $\forall$CTL* formulas, and let M_φ be a maximal model of φ. Then φ implies ψ iff $M_\varphi \models \psi$.*

2.4 Alternating Automata

For an integer $d \geq 1$, let $[d] = \{1, \ldots, d\}$. An *infinite d-tree* is the set $T = [d]^*$. The elements of $[d]$ are *directions*, the elements of T are *nodes*, and the empty word ϵ is the *root* of T. For every $x \in T$, the nodes $x \cdot c$, for $c \in [d]$, are the *successors* of x. A *path* of T is a set $\rho \subseteq T$ such that $\epsilon \in \rho$ and for each $x \in \rho$, exactly one successor of x is in ρ. Given an alphabet Σ, a *Σ-labeled d-tree* is a pair $\langle T, V \rangle$, where T is a d-tree and $V : T \rightarrow \Sigma$ maps each node of T to a letter in Σ. A Σ-labeled 1-tree is a *word* over Σ. For a language $\mathcal{L}$ of words over Σ, the *derived language* of $\mathcal{L}$, denoted $der(\mathcal{L})$ is the set of all Σ-labeled trees all of whose paths are labeled by words in $\mathcal{L}$. For $d \geq 2$, we denote by $der_d(\mathcal{L})$ the set of Σ-labeled d-trees in $der(\mathcal{L})$. For a system S with branching degrees in d, we denote by $tree(S)$ the 2^{AP}-labeled d-tree obtained by unwinding S from its initial state.

An *alternating tree automaton* [41] $\mathcal{A} = \langle \Sigma, d, Q, q_0, \delta, \alpha \rangle$ runs on Σ-labeled d-trees. It consists of a finite set Q of states, an initial state $q_0 \in Q$, a transition function δ, and an acceptance condition α (a condition that defines a subset of Q^ω). For the set $[d]$ of directions, let $\mathcal{B}^+([d] \times Q)$ be the set of positive Boolean formulas over $[d] \times Q$; i.e., Boolean formulas built from elements in $[d] \times Q$ using $\wedge$ and $\vee$, where we also allow the formulas $\mathbf{true}$ and $\mathbf{false}$. The transition function $\delta : Q \times \Sigma \rightarrow \mathcal{B}^+([d] \times Q)$ maps a state and an input letter to a formula that suggests a new configuration for the automaton. For example, when $d = 2$, having

$$\delta(q, \sigma) = ((1, q_1) \wedge (1, q_2)) \vee ((1, q_2) \wedge (2, q_2) \wedge (2, q_3))$$

means that when the automaton is in state q and reads the letter σ, it can either send two copies, in states q_1 and q_2, to direction 1 of the tree, or send a copy in state q_2 to direction 1 and two copies, in states q_2 and q_3, to direction 2.

A *run* of an alternating automaton $\mathcal{A}$ on an input Σ-labeled d-tree $\langle T, V\rangle$ is a labeled tree $\langle T_r, r\rangle$ (without a fixed branching degree) in which the root is labeled by q_0 and every other node is labeled by an element of $[d]^* \times Q$. Each node of T_r corresponds to a node of T. A node in T_r, labeled by (x, q), describes a copy of the automaton that reads the node x of T and visits the state q. For example, if $\langle T, V\rangle$ is a 2-tree with $V(\epsilon) = a$ and $\delta(q_0, a) = ((1, q_1) \vee (1, q_2)) \wedge ((1, q_3) \vee (2, q_2))$, then the nodes of $\langle T_r, r\rangle$ at level 1 include the label $(1, q_1)$ or $(1, q_2)$, and include the label $(1, q_3)$ or $(2, q_2)$. Each infinite path ρ in $\langle T_r, r\rangle$ is labeled by a word $r(\rho)$ in Q^ω. Let $inf(\rho)$ denote the set of states in Q that appear in $r(\rho)$ infinitely often. A run $\langle T_r, r\rangle$ is accepting iff all its infinite paths satisfy the acceptance condition. We consider two types of acceptance conditions:

- *Büchi*, where $\alpha \subseteq Q$, and an infinite path ρ satisfies α iff $inf(\rho) \cap \alpha \neq \emptyset$.
- *Rabin*, where $\alpha \subseteq 2^Q \times 2^Q$, and an infinite path ρ satisfies an acceptance condition $\alpha = \{\langle G_1, B_1\rangle, \ldots, \langle G_k, B_k\rangle\}$ iff there exists $1 \leq i \leq k$ for which $inf(\rho) \cap G_i \neq \emptyset$ and $inf(\rho) \cap B_i = \emptyset$.

An automaton accepts a tree iff there exists an accepting run on it. We denote by $\mathcal{L}(\mathcal{A})$ the language of the automaton $\mathcal{A}$; i.e., the set of all labeled trees that $\mathcal{A}$ accepts.

When $d = 1$, we say that $\mathcal{A}$ is a *word automaton*, we omit d from the specification of the automaton, and we describe its transitions by formulas in $\mathcal{B}^+(Q)$. We say that $\mathcal{A}$ is a *nondeterministic* automaton iff all the transitions of $\mathcal{A}$ have only disjunctively related atoms sent to the same direction; i.e., if the transitions are written in DNF, then every disjunct contains at most one atom of the form (c, q), for all $c \in [d]$. Note that a transition of nondeterministic word automata is a disjunction of states in Q, and we denote it by a set. We say that $\mathcal{A}$ is a *deterministic* automaton iff all the transitions of $\mathcal{A}$ have only disjunctively related atoms, all sent to different directions. A language of infinite words defined by a nondeterministic Büchi automaton is called ω*-regular*.

The following theorem relates LTL and automata.

Theorem 3. [53] *Given an LTL formula ψ, there exists a Büchi automaton $\mathcal{A}_\psi$ of size $2^{O(|\psi|)}$, such that $\mathcal{L}(\mathcal{A}_\psi)$ is exactly the set of computations satisfying ψ.*

We denote each of the different types of automata by three letter acronyms in $\{D, N, A\} \times \{B, R\} \times \{W, T\}$, where the first letter describe the branching mode of the automaton (deterministic, nondeterministic, or alternating), the second letter describes the acceptance condition (Büchi or Rabin), and the third letter describes the object over which the automaton runs (words or trees). We use the acronyms also to refer to the set of words (or trees) that can be defined by the various automata. For example, DBW denotes deterministic Büchi word automata, as well as the set of ω-regular languages that can be recognized by a deterministic word automaton. Which interpretation we refer to would be clear from the context.

In [40], Muller et al. introduce *weak alternating tree automata* (AWT). In an AWT, the acceptance condition is $\alpha \subseteq Q$ and there exists a partition of Q into disjoint sets, Q_i, such that for each set Q_i, either $Q_i \subseteq \alpha$, in which case Q_i is an *accepting set*, or $Q_i \cap \alpha = \emptyset$, in which case Q_i is a *rejecting set*. In addition, there exists a partial order $\leq$ on the collection of the Q_i's such that for every $q \in Q_i$ and $q' \in Q_j$ for which q' occurs in $\delta(q, \sigma)$ for some $\sigma \in \Sigma$, we have $Q_j \leq Q_i$. Thus, transitions from a state in Q_i lead to states in either the same Q_i or a lower one. It follows that every infinite path of a run of an AWT ultimately gets "trapped" within some Q_i. The path then satisfies the acceptance condition if and only if Q_i is an accepting set.

3 LTL vs. CTL

In this section we describe an attempt to utilize the tight syntactic relation between ψ and ψ_A in order to relate linear and branching model checking. In other words, we are given a system M and an LTL formula ψ, and we try to make use of ψ_A and to benefit from CTL model-checking tools in the process of deciding whether M satisfies ψ. A natural thing to start with is to check the equivalence of $A\psi$ and ψ_A. For an LTL formula ψ, we say that ψ is *branchable* iff $A\psi$ and ψ_A are equivalent. Clearly, if ψ is branchable, then checking whether M satisfies ψ can be reduced to checking whether M satisfies ψ_A. We first claim that one side of the equivalence between ψ and ψ_A is trivial.

Lemma 4. [31] *For every LTL formula ψ, the implication $\psi_A \rightarrow A\psi$ is valid.*

By Lemma 4, deciding whether ψ is branchable can be reduced to checking the implication $A\psi \rightarrow \psi_A$. This implication is not valid for all LTL formulas ψ. For example, the formula $AFGp$ does not imply the formula $AFAGp$. We now solve this implication problem. As $A\psi$ and ψ_A are CTL* formulas, the known 2EXPTIME upper bound for CTL* satisfiability [13] suggests an obvious 2EXPTIME upper bound for the problem. Moreover, as $A\psi$ and ψ_A are $\forall$CTL* formulas, the problem can be solved in EXPSPACE [27]. Can we hope for a better bound? This at first seems unlikely: the implication problem $\psi \rightarrow \varphi$ is EXPSPACE-hard already for ψ in LTL and φ in $\forall$CTL [27]. Here, however, we handle the special case where $\varphi = \psi_A$. Hopefully, the tight syntactic relation between ψ and ψ_A would enable a more efficient check. It is conjectured in [31] that the problem can be solved in polynomial space, which matches our lower bound. The algorithm we present below requires exponential space, and in the worst case shows no improvement over the known EXPSPACE upper bound. Yet, it is much simpler than the algorithm in [27], as it avoids Safra's complicated co-determinization construction that is used in the definition of the maximal models described there.

Theorem 5. [31] *For an LTL formula ψ, checking $A\psi \rightarrow \psi_A$ is in EXPSPACE and is PSPACE-hard.*

Proof: (sketch) We start with the upper bound. By Theorem 2, checking the implication $A\psi \rightarrow \psi_A$ can be reduced to model checking of ψ_A in the maximal model $M_{A\psi}$ of $A\psi$. To complete the reduction of implication to model checking, we have to describe the construction of maximal models for LTL formulas. A construction of maximal models for $\forall$CTL* formulas is described in [27]. The restriction to LTL formulas enables us to come with a much simpler construction.

Given an LTL formula ψ, let $\mathcal{A}_\psi = \langle 2^{AP}, Q, Q_0, \delta, F\rangle$ be the Büchi automaton that corresponds to ψ (see Theorem 3). Following the construction in [53], each state s of $\mathcal{A}_\psi$ is a set of subformulas of ψ. When in state s, the automaton $\mathcal{A}_\psi$ accepts exactly all the computations that satisfy all the formulas in s. In particular, for every state s, the set $\delta(s,\sigma)$ is not empty iff $s \cap AP = \sigma$. Defining the maximal model M_ψ, we apply to $\mathcal{A}_\psi$ the classical *subset construction* of [45], that is, we extend δ to a mapping from $2^Q \times 2^{AP}$ to 2^Q where $\delta(S,a) = \{q \in Q : q \in \delta(s,a) \text{ for some } s \in S\}$. We also extends δ to words in $(2^{AP})^*$, where $\delta(S,\varepsilon) = S$ and for $w \in (2^{AP})^*$ and $a \in 2^{AP}$, we have $\delta(S, w\cdot a) = \delta(\delta(S,w),a)$. We define $M_\psi = \langle AP, W, W_0, R, L, \alpha\rangle$ as follows.

– $W \subseteq 2^Q \times Q \times \{0,1\}$ is such that $\langle S,s,b\rangle \in W$ iff $s \in S$. Intuitively, the 2^Q-element S follows the subset construction. Since the Q-element s is always a member of S, then whenever we move from the state $\langle S,s,b\rangle$ to a state $\langle S',s',b'\rangle$, it may be the case that the update of the Q-element is consistent with δ (i.e., s' is a successor of s in $\mathcal{A}_\psi$), and it may also be the case that the update of the Q-element is not consistent with δ (i.e., s' is a successor of some other state in S). The Boolean flag b distinguishes between the two cases.

– $W_0 = \{\langle Q_0, q_0, 0\rangle : q_0 \in Q_0\}$.
– $L(\langle S, s, b\rangle) = s \cap AP$. Thus, the label of each state is the set of atomic propositions that hold in its Q-element.
– $R(\langle S, s, b\rangle, \langle S', s', b'\rangle)$ if $\delta(S, L(\langle S, s, b\rangle)) = S'$, $s' \in S'$, and one of the following holds
  - $s' \in \delta(s, L(\langle S, s, b\rangle))$ and $b' = 0$, or
  - $b' = 1$.
– $\alpha = \{\langle 2^Q \times F \times \{0,1\}, 2^Q \times Q \times \{1\}\rangle\}$. That is, a computation of M_ψ is fair iff its projection on the Q-element visits F infinitely often and its projection on the Boolean flag visits 1 only finitely often. Accordingly, a computation is fair if it has a suffix in which the Q-element describes an accepting run of $\mathcal{A}_\psi$.

It can be shown prove that M_ψ is indeed a maximal model of ψ. Since the size of $\mathcal{A}_\psi$ is exponential in ψ, the size of M_ψ is $2^{2^{O(|\psi|)}}$. The model-checking problem for CTL with respect to fair Rabin systems with a single pair can be solved in space that is polynomial in the length of the formula and is polylogarithmic in the size of the system [27]. Moreover, the algorithm described in [27] proceeds on-the-fly, without keeping the whole system in memory at once. Therefore, the EXPSPACE upper bound follows.

For the lower bound, we do a reduction from LTL satisfiability, proved to be hard for PSPACE in [48]. Consider the LTL formula $\theta = (Xp) \vee (Xq) \vee X((\neg p) \wedge (\neg q))$. It is easy to see that while θ is valid, the ∀CTL formula $\theta_A = (AXp) \vee (AXq) \vee AX((\neg p) \wedge (\neg q))$ is not valid. Given an LTL formula φ (we assume that the set of φ's atomic propositions does not contain p and q), let $\psi = \varphi \wedge \theta$. While ψ is equivalent to φ, it is not necessarily true that ψ_A is equivalent to φ_A. It can be shown that $A\psi \to \psi_A$ iff φ is not satisfiable. ▮

In practice, the algorithm in Theorem 5 requires time that is double exponential in the length of ψ. A naive check of a system M with respect to ψ requires time that is polynomial in the size of M and only exponential in the length of ψ. So, though ψ is usually much smaller than M, checking ψ for being branchable may be more expensive than checking M with respect to ψ, making our first attempt not very appealing. (See [31] for other attempts to take advantage of the relationship between an LTL formula ψ and the CTL formula ψ_A.)

4 LTL vs. AFMC

We start by relating AFMC and AWT. While tree automata run on trees with some finite fixed set of branching degrees and can distinguish between the different successors of a node, AFMC formulas define trees of arbitrary branching degrees and cannot distinguish between different successors. Accordingly, discussion is restricted to trees over some fixed branching degree and the AFMC is *directed* (that is, the next-time operator is annotated with an explicit direction) [19]. For every $d \geq 1$, let AWT(d) be the set of AWT (and similarly for other types of tree automata) that contains AWT of branching degree d, and let AFMC(d) be the set of directed AFMC formulas where the next-time operator is annotated by directions in $[d]$.

Theorem 6. [30] *For every $d \geq 1$, we have AWT(d) = AFMC(d).*

We now characterize ω-regular languages $\mathcal{L}$ for which $der(\mathcal{L})$ can be characterized by an AFMC formula. Since trees in $der(\mathcal{L})$ are defined by means of a universal requirement on their paths, we do not need directed AFMC.

Theorem 7. [30] *Given an ω-regular language $\mathcal{L}$, the following are equivalent.*

1. *$der(\mathcal{L})$ can be characterized by a AFMC formula.*
2. *$\mathcal{L}$ can be characterized by a DBW.*

Proof: (sketch) Assume first that $der(\mathcal{L})$ has an equivalent AFMC formula. Then, $der_2(\mathcal{L})$ has an equivalent AFMC(2) formula and therefore, by Theorem 6, $der_2(\mathcal{L})$ can be recognized by an AWT(2). It is proved in [40] that if a language of trees can be recognized by an AWT(2), then it can also be recognized by an NBT(2). It follows that $der_2(\mathcal{L})$ can be recognized by an NBT(2). It is proved in [26] that for every ω-regular language $\mathcal{R}$, if $der_2(\mathcal{R})$ can be recognized by an NBT(2), then $\mathcal{R}$ can be recognized by a DBW. It follows that $\mathcal{L}$ can be recognized by a DBW.

Assume now that $\mathcal{L}$ can be recognized by a DBW. Let $\mathcal{A} = \langle \Sigma, Q, q_0, \delta, \alpha \rangle$ be a DBW that recognizes $\mathcal{L}$. We define an AFMC formula $\psi_{\mathcal{L}}$ such that for every system S, we have $tree(S) \in der(\mathcal{L})$ iff S satisfies $\psi_{\mathcal{L}}$. The formula $\psi_{\mathcal{L}}$ has Σ as its set of atomic propositions. Each state $q \in Q$ induces two atomic variables P_q and P'_q with the following two equations:

$$P_q = \bigvee_{\sigma \in \Sigma} \sigma \wedge AXP_{\delta(q,\sigma)} \wedge AXP'_{\delta(q,\sigma)}.$$

$$P'_q = \begin{bmatrix} \mathbf{true} & \text{if } q \in \alpha, \\ \bigvee_{\sigma \in \Sigma} \sigma \wedge AXP'_{\delta(q,\sigma)} & \text{if } q \notin \alpha. \end{bmatrix}$$

The set of equations is partitioned into two blocks. Equations with a left had side variable P_q constitute a greatest fixed-point block, and equations with a left had side variable P'_q constitute a least fixed-point block. Intuitively, once a fixed point is reached, each variable P'_q contains states w of S such that the run of $\mathcal{A}$ with initial state q on each of the computations starting at w eventually visits a state in α. Each variable P_q has one disjunct for every $\sigma \in \Sigma$. Its conjunct of the form AXP_t follows the transition function, and its conjunct of the form AXP'_t guarantees that a state from α is eventually visited. Since all variables P_q have a conjunct of the form AXP'_t in all the disjuncts in their equations, it is guaranteed that α is visited infinitely often. ∎

We now consider the case where specifications are given by ω-regular langauges. We first study the problem of deciding whether a given automaton $\mathcal{A}$ can be translated to a DBW. By Theorem 7, the latter holds iff the specification given by $\mathcal{A}$ can be translated to the AFMC.

Theorem 8. [30] *Deciding NBW $\mapsto$ DBW is PSPACE-complete.*

Theorem 8 consider the problem of checking whether a specification given by an automaton can be translated to a DBW, and hence also to an AFMC formula. We now consider the blow-up in the translation.

Theorem 9. [30] *When exists, the translation*

(1) *$\{$DBW,DRW$\}$ $\mapsto$ AFMC is linear.*
(2) *NBW $\mapsto$ AFMC is exponential.*

Proof: (sketch) A linear translation of DBW to AFMC is described in the proof of Theorem 7. It is proved in [24], that if a DRW $\mathcal{A}$ is in DBW, then it has a DBW of the same size. NBW has an exponential translation to DRW [46]. Therefore, by **(1)**, if $\mathcal{A}$ is in DBW, it has an exponential equivalent AFMC formula. ▮

It follows from [38] that the exponential translation from NBW to DBW cannot be improved. This, however, does not imply an exponential lower bound for the translation to the AFMC.

We now consider the case where specifications are given by LTL formulas. We first consider the problem of checking whether a given LTL formula can be translated to a DBW and hence, also to an AFMC formula.

Theorem 10. [30] *Deciding LTL $\mapsto$ DBW is in EXPSPACE and is PSPACE-hard.*

Proof: (sketch) Given an LTL formula ψ of length n, let $\mathcal{B}_\psi$ be an NBW with $2^{O(n)}$ states that recognizes ψ (Theorem 3). Membership in EXPSPACE then follows from Theorem 8. For the lower bound, we do a reduction from LTL satisfiability. Given an LTL formula ψ over some set AP of propositions, let p be a proposition not in AP. It can can be shown that ψ is not satisfiable iff $\psi \wedge FGp$ is in DBW. ▮

We now discuss the blow-up involved in translating a given LTL formula ψ to an equivalent AFMC formula (when exists). One possibility is to first translate ψ to a DBW. Unfortunately, such an approach is inherently doubly exponential.

Theorem 11. [30] *When exists, the translation LTL $\mapsto$ DBW is doubly exponential.*

Thus, translating LTL to AFMC by going through DBW involves a doubly exponential blow up. Hopefully, this blow-up can be improved to an exponential one, matching the known lower bound.

Theorem 12. [30] *When exists, the translation LTL $\mapsto$ AFMC is doubly exponential and is at least exponential.*

Thus, our second approach to utilizing branching-time tools in linear-time model checking does not seem to be very useful either.

5 In Practice

In the previous sections, we describe two attempts to utilize the relation between linear-time and branching time logics in order to use branching-time model-checking tools in the process of model checking linear-time properties. Our motivation, recall, is the computational advantage of branching time over linear time. This advantage, however, was in terms of worst-case complexity. In this section we claim that, in practice, LTL model checkers perform nicely on typical LTL formulas. In fact, they often proceed essentially as branching-time model checkers.

For simplicity, we consider systems M with no fairness conditions; i.e., systems in which all the computations are fair. As the “representative” CTL model checker we take the bottom-up labeling procedure of [7]. There, in order to check whether M satisfies φ, we label the states of M by subformulas of φ, starting from the innermost formulas and proceeding such that, when labeling a formula, all its subformulas are already labeled.

Labeling subformulas that are atomic propositions, Boolean combinations of other subformulas, or of the form $AX\theta$ or $EX\theta$ is straightforward. Labeling subformulas of the form $A\theta_1 U\theta_2$, $E\theta_1 U\theta_2$, $A\theta_1\tilde{U}\theta_2$, or $E\theta_1\tilde{U}\theta_2$ involves a backward reachability test. As the "representative" LTL model checker, we take the automata-based algorithm of [52]. There, in order to check whether M satisfies ψ, we construct a Büchi word automaton $\mathcal{A}_{\neg\psi}$ for $\neg\psi$ and check whether the intersection of the language of M with that of $\mathcal{A}_{\neg\psi}$ is nonempty. In practice, the latter check proceeds by checking whether there exists an initial state in the intersection that satisfies CTL formula $EG\mathbf{true}$. For the construction of $\mathcal{A}_{\neg\psi}$, we follow the definition in [17], which improves [52] by being demand-driven; that is, the state space of $\mathcal{A}_{\neg\psi}$ is restricted to states that are reachable from the initial state. (See [31] for a more through analysis of the relationship between LTL and CTL model checkers. See also experimental work in [16], which confirms this analysis.)

Example 13. Consider the LTL formula $\psi = G(\neg p \vee X\neg p)$. Note that $A\psi$ is equivalent to the CTL formula $\psi_A = AG(\neg p \vee AX\neg p)$. A standard CTL model checkers proceeds in three phases. In the first phase the model checker captures the set of states that satisfy $AX\neg p$, i.e., that do not satisfy EXp. In the second phase it captures the states that satisfy $p \rightarrow AX\neg p$, i.e., that do not satisfy $p \wedge EXp$. In the third phase it captures the states that satisfies ψ_A, these are the state from which there is no path to a state that satisfies $p \wedge EXp$.

The formula $\varphi = F(p \wedge Xp)$ complements ψ. The automaton that corresponds to φ by Theorem 3 is the automaton $\mathcal{A}_\varphi = \langle 2^{\{p\}}, \{\varphi, p, \mathbf{true}, \mathbf{false}\}, \{\varphi\}, \delta, \{\mathbf{true}\}\rangle$, where δ is as follows.

- $\delta(\varphi, \{p\}) = \{\varphi, p\}$.
- $\delta(\varphi, \emptyset) = \{\varphi\}$.
- $\delta(p, \{p\}) = \{\mathbf{true}\}$.
- $\delta(p, \emptyset) = \{\mathbf{false}\}$.
- For all $\sigma \in 2^{AP}$, we have $\delta(\mathbf{true}, \sigma) = \{\mathbf{true}\}$.
- For all $\sigma \in 2^{AP}$, we have $\delta(\mathbf{false}, \sigma) = \{\mathbf{false}\}$.

An LTL model checker takes the product of the systems M with $\mathcal{A}_\varphi$ and then check for nonemptiness, by repeatedly computing the set of states from which an accepting state can be reached. In our example, the accepting states are $W \times \{\mathbf{true}\}$, where W is the state set of M. Thus, the model checker will proceed in three phases. In the first phase, it will capture thet states that lead to $W \times \{\mathbf{true}\}$ in one step; these are precisely the states that satisfy EXp. In the second phase it captures the states that satisfy $p \wedge EXp$. In the third phase it captures the states from which there is a path to a state that satisfy $p \wedge EXp$.. Thus, the branching-time and linear-time model checkers proceed in an analogous way and do essentially the same work.∎

Example 14. Consider the LTL formula $\psi = F(p \wedge Xp)$. By [12], $A\psi$ is not equivalent to a CTL formula. Using the techniques of Section 4 we can show that ψ is equivalent to the AFMC formula $\mu\ P_0\,((\neg p \wedge AXP_0) \vee (p \wedge AX\mu P_1(p \vee AXP_0))$. The evaluation of this formulas proceeds as follows. Initially, P_1 consists of all the states in which p holds, while P_0 is empty. In subsequent iterations, until convergence, P_1 gets all states all of whose successors are in P_0, while P_1 gets all states where p holds and all successors are in P_1 or p does not hold and all successors in P_0.

The formula $\varphi = G(p \rightarrow X\neg p)$ complements ψ. The automaton that corresponds to φ by Theorem 3 is $\mathcal{A}_\varphi = \langle 2^{\{p\}}, \{\varphi, \varphi \wedge \neg p, \mathbf{false}\}, \{\varphi\}, \delta, \{\varphi, \varphi \wedge \neg q\}\rangle$, where δ is as follows.

- $\delta(\varphi, \emptyset) = \{\varphi\}$.
- $\delta(\varphi, \{p\}) = \{\varphi \wedge \neg p\}$.
- $\delta(\varphi \wedge \neg p, \{p\}) = \{\mathbf{false}\}$.
- $\delta(\varphi \wedge \neg p, \emptyset) = \{\varphi\}$
- For all $\sigma \in 2^{AP}$, we have $\delta(\mathbf{false}, \sigma) = \{\mathbf{false}\}$.

An LTL model checker takes the product of the systems M with $\mathcal{A}_\varphi$ and then check for nonemptiness. It proceeds as follows. It first eliminates all states $(s, \mathbf{false})$ or $(s, \varphi \wedge \neg p)$, where p holds in s. In subsequent iterations, until convergence, it eliminates states all of whose successors have been eliminated. A careful study shows that this computation is essentially the dual of the AFMC model-checking computation; the sets of states captured by the AFMC model checker are simply complementary to the sets of states eliminated by the LTL model checker. Thus, the branching-time model checker and the linear-time model checker proceed in an analogous way and do essentially the same work.█

6 Conclusions

This paper was motivated by the "folk" wisdom according to which model checking is easier for branching time than for linear time. As we argued in the previous section, this belief is based on worst-case complexity rather than practical complexity. It turns out that even from the perspective of worst-case complexity the computational superiority of branching time is also not that clear [51]. For example, comparing the complexities of CTL and LTL model checking for concurrent programs, both are PSPACE-complete [52,2]. As shown in [49,27], the advantage that CTL enjoys over LTL disappears also when the complexity of modular verification is considered. The distinction between closed an open systems questions the computational superiority of the branching-time paradigm further [28,29,50].

Our conclusion is that the debate about the relative merit of the linear and branching paradigms will not be settled by technical arguments such as expressive power or computational complexity. Rather, the discussion should focus on the attractiveness of the approaches to practitioners who practice computer-aided verification in realistic settings. We believe that this discussion will end up with the conclusion that both approaches have their merits and computer-aided verification tools should therefore combine the two approaches rather than "religiously" adhere to one or the other.

Acknowledgement: Orna Kupferman's help is greatly appreciated.

References

1. M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.
2. O. Bernholtz, M.Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In D. L. Dill, editor, *Computer Aided Verification, Proc. 6th Int. Conference*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, June 1994. Springer-Verlag, Berlin.
3. R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, T. Kukimoto, A. Pardo, S. Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, and T. Villa. VIS: a system for verification and synthesis. In *Computer Aided Verification, Proc. 8th Int. Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 428–432. Springer-Verlag, 1996.
4. E. M. Clarke, I. A. Draghicescu, and R. P. Kurshan. A unified approach for showing language containment and equivalence between various types of ω-automata. *Information Processing Letters* **46**, pages 301–308, (1993).

5. E.M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *Proc. Workshop on Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 428–437. Springer-Verlag, 1988.
6. E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proc. Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.
7. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.
8. E.M. Clarke, O. Grumberg, and D. Long. Verification tools for finite-state concurrent systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Decade of Concurrency – Reflections and Perspectives (Proceedings of REX School)*, volume 803 of *Lecture Notes in Computer Science*, pages 124–175. Springer-Verlag, 1993.
9. R. Cleaveland. A linear-time model-checking algorithm for the alternation-free modal μ-calculus. *Formal Methods in System Design*, 2:121–147, 1993.
10. E.A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990.
11. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In *Proc. 7th Int'l Colloq. on Automata, Languages and Programming*, pages 169–181, 1980.
12. E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.
13. E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 368–377, White Plains, October 1988.
14. E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.
15. E.A. Emerson and C.-L. Lei. Efficient model checking in fragments of the proposoitional Mu-calculus. In *Proc. 1st Symposium on Logic in Computer Science*, pages 267–278, Cambridge, June 1986.
16. K. Fisler and M.Y. Vardi. Bisimulation minimization in an automata-theoretic verification framework. Unpublished manuscript, 1998.
17. R. Gerth, D. Peled, M.Y. Vardi, and P. Wolper. A simple on-the-fly automatic verification for linear temporal logic. In *Protocol Specification, Testing, and Verification*, pages 3–18. Chapman & Hall, August 1995.
18. O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans. on Programming Languages and Systems*, 16(3):843–871, 1994.
19. Th. Hafer and W. Thomas. Computation tree logic CTL* and path quantifiers in the monadic theory of the binary tree. In *Proc. 14th International Coll. on Automata, Languages, and Programming*, volume 267 of *Lecture Notes in Computer Science*, pages 269–279. Springer-Verlag, 1987.
20. R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *Computer Aided Verification, Proc. 8th Int. Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 423–427. Springer-Verlag, 1996.
21. R.H. Hardin, R.P. Kurshan, S.K. Shukla, and M.Y. Vardi. A new heuristic for bad cycle detection using BDDs. In *Computer Aided Verification, Proc. 9th Int. Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 268–278. Springer-Verlag, 1997.
22. G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
23. D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
24. S.C. Krishnan, A. Puri, and R.K. Brayton. Deterministic ω-automata vis-a-vis deterministic Buchi automata. In *Algorithms and Computations*, volume 834 of *Lecture Notes in Computer Science*, pages 378–386. Springer-Verlag, 1994.

25. O. Kupferman and O. Grumberg. Buy one, get one free!!! *Journal of Logic and Computation*, 6(4):523–539, 1996.
26. O. Kupferman, S. Safra, and M.Y. Vardi. Relating word and tree automata. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 322–333, DIMACS, June 1996.
27. O. Kupferman and M.Y. Vardi. On the complexity of branching modular model checking. In *Proc. 6th Conferance on Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 408–422, Philadelphia, August 1995. Springer-Verlag.
28. O. Kupferman and M.Y. Vardi. Module checking. In *Computer Aided Verification, Proc. 8th Int. Conference*, volume 1102 of *Lecture Notes in Computer Science*, pages 75–86. Springer-Verlag, 1996.
29. O. Kupferman and M.Y. Vardi. Module checking revisited. In *Computer Aided Verification, Proc. 9th Int. Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 1997.
30. O. Kupferman and M.Y. Vardi. Freedom, weakness, and determinism: from linear-time to branching-time. In *Proc. 13th IEEE Symposium on Logic in Computer Science*, Indiana, June 1998.
31. O. Kupferman and M.Y. Vardi. Relating linear and branching model checking. In *IFIP Working Conference on Programming Concepts and Methods*, pages 304 – 326, New York, June 1998. Chapman & Hall.
32. O. Kupferman and M.Y. Vardi. Verification of fair transition systems. *Chicago Journal of Theoretical Computer Science*, 1998(2), March 1998.
33. R.P. Kurshan. *Computer Aided Verification of Coordinating Processes.* Princeton Univ. Press, 1994.
34. L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proc. 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.
35. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.
36. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, January 1992.
37. K.L. McMillan. *Symbolic model checking*. Kluwer Academic Publishers, 1993.
38. M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.
39. R. Milner. An algebraic definition of simulation between programs. In *Proc. 2nd International Joint Conference on Artificial Intelligence*, pages 481–489. British Computer Society, September 1971.
40. D.E. Muller, A. Saoudi, and P.E. Schupp. Alternating automata, the weak monadic theory of the tree and its complexity. In *Proc. 13th Int. Colloquium on Automata, Languages and Programming*. Springer-Verlag, 1986.
41. D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54,:267–276, 1987.
42. A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proc. 12th Int. Colloquium on Automata, Languages and Programming*, pages 15–32. Lecture Notes in Computer Science, Springer-Verlag, 1985.
43. J.P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proc. 5th International Symp. on Programming*, volume 137, pages 337–351. Springer-Verlag, Lecture Notes in Computer Science, 1981.
44. M.O. Rabin. Weakly definable relations and special automata. In *Proc. Symp. Math. Logic and Foundations of Set Theory*, pages 1–23. North Holland, 1970.
45. M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:115–125, 1959.
46. S. Safra. On the complexity of ω-automata. In *Proc. 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, White Plains, October 1988.
47. K. Schneider. CTL and equivalent sublanguages of CTL*. In *Proceedings of IFIP Conference on Computer Hardware Description Languages and Applications*, pages 40–59, Toledo, April 1997. Chapman and Hall.

48. A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal ACM*, 32:733–749, 1985.
49. M.Y. Vardi. On the complexity of modular model checking. In *Proc. 10th IEEE Symposium on Logic in Computer Science*, pages 101–111, June 1995.
50. M.Y. Vardi. Verifiation of open systems. In S. Ramesh and G. Sivakuma, editors, *Proc. 17th Conf. on Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computers Science, pages 250–266. Springer-Verlag, 1997.
51. M.Y. Vardi. Linear vs. branching time: A complexity-theoretic perspective. In *Proc. 13th IEEE Sym.. on Logic in Computer Science*, 1998.
52. M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.
53. M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.