An Automata-Theoretic Approach to Linear Temporal Logic

Moshe Y. Vardi[⋆]

Rice University
Department of Computer Science
P.O. Box 1892
Houston, TX 77251-1892, U.S.A.
Email: vardi@cs.rice.edu
URL: http://www.cs.rice.edu/~vardi

Abstract. The automata-theoretic approach to linear temporal logic uses the theory of automata as a unifying paradigm for program specification, verification, and synthesis. Both programs and specifications are in essence descriptions of computations. These computations can be viewed as words over some alphabet. Thus, programs and specifications can be viewed as descriptions of languages over some alphabet. The automata-theoretic perspective considers the relationships between programs and their specifications as relationships between languages. By translating programs and specifications to automata, questions about programs and their specifications can be reduced to questions about automata. More specifically, questions such as satisfiability of specifications and correctness of programs with respect to their specifications can be reduced to questions such as nonemptiness and containment of automata.

Unlike classical automata theory, which focused on automata on finite words, the applications to program specification, verification, and synthesis, use automata on infinite words, since the computations in which we are interested are typically infinite. This paper provides an introduction to the theory of automata on infinite words and demonstrates its applications to program specification, verification, and synthesis.

1 Introduction

While *program verification* was always a desirable, but never an easy task, the advent of *concurrent programming* has made it significantly more necessary and difficult. Indeed, the conceptual complexity of concurrency increases the likelihood of the program containing errors. To quote from [OL82]: “There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors.”

The first step in program verification is to come up with a *formal specification* of the program. One of the more widely used specification languages for concurrent programs is *temporal logic* [Pnu77, MP92]. Temporal logic comes in two varieties: linear time and branching time ([EH86, Lam80]); we concentrate here on linear time. A linear temporal

[⋆] Part of this work was done at the IBM Almaden Research Center.

specification describes the computations of the program, so a program *satisfies* the specification (is *correct*) if all its computations satisfy the specification. Of course, a specification is of interest only if it is *satisfiable*. An unsatisfiable specification cannot be satisfied by any program. An often advocated approach to program development is to avoid the verification step altogether by using the specification to *synthesize* a program that is guaranteed to be correct.

Our approach to specification, verification, and synthesis is based on an intimate connection between linear temporal logic and automata theory, which was discussed explicitly first in [WVS83] (see also [LPZ85, Pei85, Sis83, SVW87, VW94]). This connection is based on the fact that a computation is essentially an infinite sequence of states. In the applications that we consider here, every state is described by a finite set of atomic propositions, so a computation can be viewed as an infinite word over the alphabet of truth assignments to the atomic propositions. The basic result in this area is the fact that temporal logic formulas can be viewed as *finite-state acceptors*. More precisely, given any propositional temporal formula, one can construct a finite automaton on infinite words that accepts precisely the computations satisfied by the formula [VW94]. We will describe the applications of this basic result to satisfiability testing, verification, and synthesis. (For an extensive treatment of the automata-theoretic approach to verification see [Kur94]).

Unlike classical automata theory, which focused on automata on finite words, the applications to specification, verification, and synthesis, use automata on infinite words, since the computations in which we are interested are typically infinite. Before going into the applications, we give a basic introduction to the theory of automata on infinite words. To help the readers build their intuition, we review the theory of automata on finite words and contrast it with the theory of automata on infinite words. For a more advanced introduction to the theory of automata on infinite objects, the readers are referred to [Tho90].

2 Automata Theory

We are given a finite nonempty alphabet Σ. A *finite word* is an element of Σ^*, i.e., a finite sequence $a_0, \ldots, a_n$ of symbols from Σ. An *infinite word* is an element of Σ^ω, i.e., an ω-sequence[2] $a_0, a_1, \ldots$ of symbols from Σ. Automata on finite words define (finitary) *languages*, i.e., sets of finite words, while automata on infinite words define *infinitary languages*, i.e., sets of infinite words.

2.1 Automata on Finite Words – Closure

A (nondeterministic finite) automaton A is a tuple $(\Sigma, S, S^0, \rho, F)$, where Σ is a finite nonempty *alphabet*, S is a finite nonempty set of *states*, $S^0 \subseteq S$ is a nonempty set of *initial* states, $F \subseteq S$ is the set of *accepting* states, and $\rho : S \times \Sigma \rightarrow 2^S$ is a *transition function*. Intuitively, $\rho(s, a)$ is the set of states that A can move into when it is in state s and it reads the symbol a. Note that the automaton may be *nondeterministic*, since it may have many initial states and the transition function may specify many possible

[2] ω denotes the first infinite ordinal.

transitions for each state and symbol. The automaton A is *deterministic* if $|S^0| = 1$ and $|\rho(s,a)| \leq 1$ for all states $s \in S$ and symbols $a \in \Sigma$. An automaton is essentially an edge-labeled directed graph: the states of the automaton are the nodes, the edges are labeled by symbols in Σ, a certain set of nodes is designated as initial, and a certain set of nodes is designated as accepting. Thus, $t \in \rho(s,a)$ means that that there is edge from s to t labeled with a. When A is deterministic, the transition function ρ can be viewed as a partial mapping from $S \times \Sigma$ to S, and can then be extended to a partial mapping from $S \times \Sigma^*$ to S as follows: $\rho(s,\varepsilon) = s$ and $\rho(s,xw) = \rho(\rho(s,x),w)$ for $x \in \Sigma$ and $w \in \Sigma^*$.

A *run* r of A on a finite word $w = a_0, \ldots, a_{n-1} \in \Sigma^*$ is a sequence $s_0, \ldots, s_n$ of $n+1$ states in S such that $s_0 \in S^0$, and $s_{i+1} \in \rho(s_i, a_i)$ for $0 \leq i < n$. Note that a nondeterministic automaton can have many runs on a given input word. In contrast, a deterministic automaton can have at most one run on a given input word. The run r is *accepting* if $s_n \in F$. One could picture the automaton as having a green light that is switched on whenever the automaton is in an accepting state and switched off whenever the automaton is in a non-accepting state. Thus, the run is accepting if the green light is on at the end of the run. The word w is *accepted* by A if A has an accepting run on w. When A is deterministic, $w \in L(A)$ if and only if $\rho(s^0, w) \in F$, where $S^0 = \{s^0\}$. The (finitary) *language* of A, denoted $L(A)$, is the set of finite words accepted by A.

An important property of automata is their closure under Boolean operations. We start by considering closure under union and intersection.

Proposition 1. [RS59] *Let A_1, A_2 be automata. Then there is an automaton A such that $L(A) = L(A_1) \cup L(A_2)$.*

Proof: Let $A_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$. Without loss of generality, we assume that S_1 and S_2 are disjoint. Intuitively, the automaton A nondeterministically chooses A_1 or A_2 and runs it on the input word.

Let $A = (\Sigma, S, S^0, \rho, F)$, where $S = S_1 \cup S_2$, $S^0 = S_1^0 \cup S_2^0$, $F = F_1 \cup F_2$, and

$$\rho(s,a) = \begin{cases} \rho_1(s,a) \text{ if } s \in S_1 \\ \rho_2(s,a) \text{ if } s \in S_2 \end{cases}$$

It is easy to see that $L(A) = L(A_1) \cup L(A_2)$. ▮

We call A in the proof above the *union* of A_1 and A_2, denoted $A_1 \cup A_2$.

Proposition 2. [RS59] *Let A_1, A_2 be automata. Then there is an automaton A such that $L(A) = L(A_1) \cap L(A_2)$.*

Proof: Let $A_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$. Intuitively, the automaton A runs both A_1 and A_2 on the input word.

Let $A = (\Sigma, S, S^0, \rho, F)$, where $S = S_1 \times S_2$, $S^0 = S_1^0 \times S_2^0$, $F = F_1 \times F_2$, and $\rho((s,t),a) = \rho_1(s,a) \times \rho_2(t,a)$. It is easy to see that $L(A) = L(A_1) \cap L(A_2)$. ▮

We call A in the proof above the *product* of A_1 and A_2, denoted $A_1 \times A_2$.

Note that both the union and the product constructions are effective and polynomial in the size of the constituent automata.

Let us consider now the issue of complementation. Consider first deterministic automata.

Proposition 3. [RS59] *Let $A = (\Sigma, S, S^0, \rho, F)$ be a deterministic automaton, and let $\overline{A} = (\Sigma, S, S^0, \rho, S - F)$, then $L(\overline{A}) = \Sigma^* - L(A)$.*

That is, it is easy to complement deterministic automata; we just have to complement the acceptance condition. This will not work for nondeterministic automata, since a nondeterministic automaton can have many runs on a given input word; it is not enough that *some* of these runs reject (i.e., not accept) the input word, *all* runs should reject the input word. Thus, it seems that to complement nondeterministic automaton we first have to *determinize* it.

Proposition 4. [RS59] *Let A be a nondeterministic automaton. Then there is a deterministic automaton A_d such that $L(A_d) = L(A)$.*

Proof: Let $A = (\Sigma, S, S^0, \rho, F)$. Then $A_d = (\Sigma, 2^S, \{S^0\}, \rho_d, F_d)$. The state set of A_d consists of all sets of states in S and it has a single initial state. The set $F_d = \{T \mid T \cap F \neq \emptyset\}$ is the collection of sets of states that intersect F nontrivially. Finally, $\rho_d(T, a) = \{t \mid t \in \rho(s, a)$ for some $s \in T\}$. ∎

Intuitively, A_d collapses all possible runs of A on a given input word into one run over a larger state set. This construction is called the *subset* construction. By combining Propositions 4 and 3 we can complement a nondeterministic automata. The construction is effective, but it involves an exponential blow-up, since determinization involves an exponential blow-up (i.e., if A has n states, then A_d has 2^n states). As shown in [MF71], this exponential blow-up for determinization and complementation is unavoidable.

For example, fix some $n \geq 1$. The set of all finite words over the alphabet $\Sigma = \{a, b\}$ that have an a at the nth position from the right is accepted by the automaton $A = (\Sigma, \{0, 1, 2, \ldots, n\}, \{0\}, \rho, \{n\})$, where $\rho(0, a) = \{0, 1\}$, $\rho(0, b) = \{0\}$, and $\rho(i, a) = \rho(i, b) = \{i + 1\}$ for $0 < i < n$. Intuitively, A guesses a position in the input word, checks that it contains a, and then checks that it is at distance n from the right end of the input.

Suppose that we have a deterministic automaton $A_d = (\Sigma, S, \{s^0\}, \rho_d, F)$ with fewer than 2^n states that accepts this same language. Recall that ρ_d can be viewed as a partial mapping from $S \times \Sigma^*$ to S. Since $|S| < 2^n$, there must be two words uav_1 and ubv_2 of length n for which $\rho_d(s^0, uav_1) = \rho_d(s^0, ubv_2)$. But then we would have that $\rho_d(s^0, uav_1u) = \rho_d(s^0, ubv_2u)$; that is, either both uav_1u and ubv_2u are members of $L(A_d)$ or neither are, contradicting the assumption that $L(A_d)$ consists of exactly the words with an a at the nth position from the right, since $|av_1u| = |bv_2u| = n$.

2.2 Automata on Infinite Words – Closure

Suppose now that an automaton $A = (\Sigma, S, S^0, \rho, F)$ is given as input an infinite word $w = a_0, a_1, \ldots$ over Σ. A *run* r of A on w is a sequence $s_0, s_1, \ldots$, where $s_0 \in S^0$ and $s_{i+1} \in \rho(s_i, a_i)$, for all $i \geq 0$. Since the run is infinite, we cannot define acceptance by the type of the final state of the run. Instead we have to consider the *limit* behavior of the run. We define lim(r) to be the set $\{s \mid s = s_i$ for infinitely many i's$\}$, i.e., the set of states that occur in r infinitely often. Since S is finite, lim(r) is necessarily nonempty. The run r is *accepting* if there is some accepting state that repeats in r infinitely often, i.e., lim$(r) \cap F \neq \emptyset$. If we picture the automaton as having a green light that is switched on precisely when the automaton is in an accepting state, then the run is accepting if the green light is switched on infinitely many times. The infinite word w is *accepted* by A if there is an accepting run of A on w. The infinitary language of A, denoted $L_\omega(A)$, is the set of infinite words accepted by A.

Thus, A can be viewed both as an automaton on finite words and as an automaton on infinite words. When viewed as an automaton on infinite words it is called a *Büchi* automaton [Büc62].

Do automata on infinite words have closure properties similar to those of automata on finite words? In most cases the answer is positive, but the proofs may be more involved. We start by considering closure under union. Here the union construction does the right thing.

Proposition 5. [Cho74] *Let A_1, A_2 be Büchi automata. Then $L_\omega(A_1 \cup A_2) = L_\omega(A_1) \cup L_\omega(A_2)$.*

One might be tempted to think that similarly we have that $L_\omega(A_1 \times A_2) = L_\omega(A_1) \cap L_\omega(A_2)$, but this is not the case. The accepting set of $A_1 \times A_2$ is the product of the accepting sets of A_1 and A_2. Thus, $A_1 \times A_2$ accepts an infinite word w if there are accepting runs r_1 and r_2 of A_1 and A_2, respectively, on w, where both runs go infinitely often and *simultaneously* through accepting states. This requirement is too strong. As a result, $L_\omega(A_1 \times A_2)$ could be a strict subset of $L_\omega(A_1) \cap L_\omega(A_2)$. For example, define the two Büchi automata $A_1 = (\{a\}, \{s,t\}, \{s\}, \rho, \{s\})$ and $A_2 = (\{a\}, \{s,t\}, \{s\}, \rho, \{t\})$ with $\rho(s,a) = \{t\}$ and $\rho(t,a) = \{s\}$. Clearly we have that $L_\omega(A_1) = L_\omega(A_2) = \{a^\omega\}$, but $L_\omega(A_1 \times A_2) = \emptyset$.

Nevertheless, closure under intersection does hold.

Proposition 6. [Cho74] *Let A_1, A_2 be Büchi automata. Then there is a Büchi automaton A such that $L_\omega(A) = L_\omega(A_1) \cap L_\omega(A_2)$.*

Proof: Let $A_1 = (\Sigma, S_1, S_1^0, \rho_1, F_1)$ and $A_2 = (\Sigma, S_2, S_2^0, \rho_2, F_2)$. Let $A = (\Sigma, S, S^0, \rho, F)$, where $S = S_1 \times S_2 \times \{1,2\}$, $S^0 = S_1^0 \times S_2^0 \times \{1\}$, $F = F_1 \times S_2 \times \{1\}$, and $(s', t', j) \in \rho((s,t,i), a)$ if $s' \in \rho_1(s,a)$, $t' \in \rho_2(t,a)$, and $i = j$, unless $i = 1$ and $s \in F_1$, in which case $j = 2$, or $i = 2$ and $t \in F_2$, in which case $j = 1$.

Intuitively, the automaton A runs both A_1 and A_2 on the input word. Thus, the automaton can be viewed has having two "tracks", one for each of A_1 and A_2. In addition to remembering the state of each track, A also has a pointer that points to one of the tracks (1 or 2). Whenever a track goes through an accepting state, the pointer moves to the other track. The acceptance condition guarantees that both tracks visit accepting states infinitely often, since a run accepts iff it goes infinitely often through $F_1 \times S_2 \times \{1\}$. This means that the first track visits infinitely often an accepting state with the pointer pointing to the first track. Whenever, however, the first track visits an accepting state with the pointer pointing to the first track, the pointer is changed to point to the second track. The pointer returns to point to the first track only if the second track visits an accepting state. Thus, the second track must also visit an accepting state infinitely often. ▮

Thus, Büchi automata are closed under both union and intersection, though the construction for intersection is somewhat more involved than a simple product. The situation is considerably more involved with respect to closure under complementation. First, as we shall shortly see, Büchi automata are not closed under determinization, i.e., nondeterministic Büchi automata are more expressive than deterministic Büchi automata. Second, it is not even obvious how to complement deterministic Büchi automata. Consider the deterministic Büchi automaton $A = (\Sigma, S, S^0, \rho, F)$. One may think that it suffices to complement the acceptance condition, i.e., to replace F by $S - F$ and define $\overline{A} = (\Sigma, S, S^0, \rho, S - F)$. Not going infinitely often through F, however, is not the same as going infinitely often through $S - F$. A run might go through both F and $S - F$ infinitely often. Thus, $L_\omega(\overline{A})$ may be a strict superset of $\Sigma^\omega - L_\omega(A)$. For example, Consider the Büchi automaton $A = (\{a\}, \{s, t\}, \{s\}, \rho, \{s\})$ with $\rho(s, a) = \{t\}$ and $\rho(t, a) = \{s\}$. We have that $L_\omega(A) = L_\omega(\overline{A}) = \{a^\omega\}$.

Nevertheless, Büchi automata (deterministic as well as nondeterministic) are closed under complementation.

Proposition 7. [Büc62] *Let A be a Büchi automaton over an alphabet Σ. Then there is a (possibly nondeterministic) Büchi automaton $\overline{A}$ such that $L_\omega(\overline{A}) = \Sigma^\omega - L_\omega(A)$.*

The construction in [Büc62] is doubly exponential. This is improved in [SVW87] to a singly exponential construction with a quadratic exponent (i.e., if A has n states then $\overline{A}$ has c^{n^2} states, for some constant $c > 1$). In contrast, the exponent in the construction of Proposition 4 is linear. We will come back later to the complexity of complementation.

Let us return to the issue of determinization. We now show that nondeterministic Büchi automata are more expressive than deterministic Büchi automata. Consider the infinitary language $\Gamma = (0+1)^* 1^\omega$, i.e., Γ consists of all infinite words in which 0 occurs only finitely many times. It is easy to see that Γ can be defined by a nondeterministic Büchi automaton. Let $A_0 = (\{0, 1\}, \{s, t\}, \{s\}, \rho, \{t\})$, where $\rho(s, 0) = \rho(s, 1) = \{s, t\}$, $\rho(t, 1) = \{t\}$ and $\rho(t, 0) = \emptyset$. That is, the states are s and t with s the initial state and t the accepting state, As long as it is in the state s, the automaton A_0 can read both inputs 0 and 1. At some point, however, A_0 makes a nondeterministic transition to the state t, and from that point on it can read only the input 1. It is easy to see that $\Gamma = L_\omega(A_0)$. In contrast, Γ cannot be defined by any deterministic Büchi automaton.

Proposition 8. *Let $\Gamma = (0 + 1)^* 1^\omega$. Then there is no deterministic Büchi automaton A such that $\Gamma = L_\omega(A)$.*

Proof: Assume by way of contradiction that $\Gamma = L_\omega(A)$, where $A = (\Sigma, S, \{s^0\}, \rho, F)$ for $\Sigma = \{0, 1\}$, and A is deterministic. Recall that ρ can be viewed as a partial mapping from $S \times \Sigma^*$ to S.

Consider the infinite word $w_0 = 1^\omega$. Clearly, w_0 is accepted by A, so A has an accepting run on w_0. Thus, w_0 has a finite prefix u_0 such that $\rho(s^0, u_0) \in F$. Consider now the infinite word $w_1 = u_0 0 1^\omega$. Clearly, w_1 is also accepted by A, so A has an accepting run on w_1. Thus, w_1 has a finite prefix $u_0 0 u_1$ such that $\rho(s^0, u_0 0 u_1) \in F$. In a similar fashion we can continue to find finite words u_i such that $\rho(s^0, u_0 0 u_1 0 \ldots 0 u_i) \in F$. Since S is finite, there are i, j, where $0 \leq i < j$, such that $\rho(s^0, u_0 0 u_1 0 \ldots 0 u_i) = \rho(s^0, u_0 0 u_1 0 \ldots 0 u_i 0 \ldots 0 u_j)$. It follows that A has an accepting run on

$$u_0 0 u_1 0 \ldots 0 u_i (0 \ldots 0 u_j)^\omega.$$

But the latter word has infinitely many occurrences of 0, so it is not in Γ. ∎

Note that the complementary language $\Sigma^\omega - \Gamma = ((0+1)^* 0)^\omega$ (the set of infinite words in which 0 occurs infinitely often) is acceptable by the deterministic Büchi automaton $A = (\{0,1\}, \{s,t\}, \{s\}, \rho, \{s\})$, where $\rho(s,0) = \rho(t,0) = \{s\}$ and $\rho(s,1) = \rho(t,1) = \{t\}$. That is, the automaton starts at the state s and then it simply remembers the last symbol it read (s corresponds to 0 and t corresponds to 1). Thus, the use of nondeterminism in Proposition 7 is essential.

To understand why the subset construction does not work for Büchi automata, consider the following two automata over a singleton alphabet: $A_1 = (\{a\}, \{s,t\}, \{s\}, \rho_1, \{t\})$ and $A_2 = (\{a\}, \{s,t\}, \{s\}, \rho_2, \{t\})$, where $\rho_1(s,a) = \{s,t\}$, $\rho_1(t,a) = \emptyset$, $\rho_2(s,a) = \{s,t\}$, and $\rho_2(t,a) = \{s\}$. It is easy to see that A_1 does not accept any infinite word, since no infinite run can visit the state t. In contrast, A_2 accepts the infinite word a^ω, since the run $(st)^\omega$ is accepting. If we apply the subset construction to both automata, then in both cases the initial state is $\{s\}$, $\rho_d(\{s\}, a) = \{s,t\}$, and $\rho_d(\{s,t\}, a) = \{s,t\}$. Thus, the subset construction can not distinguish between A_1 and A_2.

To be able to determinize automata on finite words, we have to consider a more general acceptance condition. Let S be a finite nonempty set of states. A *Rabin* condition is a subset G of $2^S \times 2^S$, i.e., it is a collection of pairs of sets of states, written $[(L_1, U_1), \ldots, (L_k, U_k)]$ (we drop the external brackets when the condition consists of a single pair). A Rabin automaton A is an automaton on infinite words where the acceptance condition is specified by a Rabin condition, i.e., it is of the form $(\Sigma, S, S^0, \rho, G)$. A run r of A is accepting if for some i we have that $\lim(r) \cap L_i \neq \emptyset$ and $\lim(r) \cap U_i = \emptyset$, that is, there is a pair in G where the left set is visited infinitely often by r while the right set is visited only finitely often by r.

Rabin automata are not more expressive than Büchi automata.

Proposition 9. [Cho74] *Let A be a Rabin automaton, then there is a Büchi automaton A_b such that $L_\omega(A) = L_\omega(A_b)$.*

Proof: Let $A = (\Sigma, S, S^0, \rho, G)$, where $G = [(L_1, U_1), \ldots, (L_k, U_k)]$. It is easy to see that $L_\omega(A) = \cup_{i=1}^k L_\omega(A_i)$, where $A_i = (\Sigma, S, S^0, \rho, (L_i, U_i))$. Since Büchi automata are closed under union, by Proposition 5, it suffices to prove the claim for Rabin conditions that consists of a single pair, say (L, U).

The idea of the construction is to take two copies of A, say A_1 and A_2. The Büchi automaton A_b starts in A_1 and stays there as long as it "wants". At some point it nondeterministically makes a transition into A_2 and it stays there avoiding U and visiting L infinitely often. Formally, $A_b = (\Sigma, S_b, S_b^0, \rho_b, L)$, where $S_b = S \times \{0\} \cup (S - U)$, $S_b^0 = S^0 \times \{0\}$, $\rho_b(s,a) = \rho(s,a) - U$ for $s \in S - U$, and $\rho_b(\langle s, 0 \rangle, a) = \rho(s,a) \times \{0\} \cup (\rho(s,a) - U)$. ∎

Note that the construction in the proposition above is effective and polynomial in the size of the given automaton.

If we restrict attention, however, to deterministic automata, then Rabin automata are more expressive than Büchi automata. Recall the infinitary language $\Gamma = (0+1)^*1^\omega$. We showed earlier that it is not definable by a deterministic Büchi automaton. It is easily definable, however, by a Rabin automaton. Let $A = (\{0,1\}, \{s,t\}, \{s\}, \rho, (\{t\}, \{s\}))$, where $\rho(s,0) = \rho(t,0) = \{s\}$, $\rho(s,1) = \rho(t,1) = \{t\}$. That is, the automaton starts at the state s and then it simply remembers the last symbol it read (s corresponds to 0 and t corresponds to 1). It is easy to see that $\Gamma = L_\omega(A)$.

The additional expressive power of Rabin automata is sufficient to provide closure under determinization.

Proposition 10. [McN66] *Let A be a Büchi automaton. There is a deterministic Rabin automaton A_d such that $L_\omega(A_d) = L_\omega(A)$.*

Proposition 10 was first proven in [McN66], where a *doubly* exponential construction was provided. This was improved in [Saf88], where a *singly* exponential, with an almost linear exponent, construction was provided (if A has n states, then A_d has $2^{O(n \log n)}$ states and $O(n)$ pairs). Furthermore, it was shown in [Saf88, EJ89]) how the determinization construction can be modified to yield a *co-determinization* construction, i.e., a construction of a deterministic Rabin automaton A'_d such that $L_\omega(A_d) = \Sigma^\omega - L_\omega(A_d)$, where Σ is the underlying alphabet. The co-determinization construction is also singly exponential with an almost linear exponent (again, if A has n states, then A'_d has $2^{O(n \log n)}$ states and $O(n)$ pairs). Thus, combining the co-determinization construction with the polynomial translation of Rabin automata to Büchi automata (Proposition 9), we get a complementation construction whose complexity is singly exponential with an almost linear exponent. This improves the previously mentioned bound on complementation (singly exponential with a quadratic exponent) and is essentially optimal [Mic88]. In contrast, complementation for automata on finite words involves an exponential blow-up with a linear exponent (Section 2.1). Thus, complementation for automata on infinite words is provably harder than complementation for automata on finite words. Both constructions are exponential, but in the finite case the exponent is linear, while in the infinite case the exponent is nonlinear.

2.3 Automata on Finite Words – Algorithms

An automaton is "interesting" if it defines an "interesting" language, i.e., a language that is neither empty nor contains all possible words. An automaton A is *nonempty* if $L(A) \neq \emptyset$; it is *nonuniversal* if $L(A) \neq \Sigma^*$. One of the most fundamental algorithmic issues in automata theory is testing whether a given automaton is "interesting", i.e., nonempty and nonuniversal. The *nonemptiness problem* for automata is to decide, given an automaton A, whether A is nonempty. The *nonuniversality problem* for automata is to decide, given an automaton A, whether A is nonuniversal. It turns out that testing nonemptiness is easy, while testing nonuniversality is hard.

Proposition 11. [RS59, Jon75]

1. *The nonemptiness problem for automata is decidable in linear time.*
2. *The nonemptiness problem for automata is NLOGSPACE-complete.*

Proof: Let $A = (\Sigma, S, S^0, \rho, F)$ be the given automaton. Let s, t be states of S. We say that t is *directly connected* to s if there is a symbol $a \in \Sigma$ such that $t \in \rho(s, a)$. We say that t is *connected* to s if there is a sequence $s_1, \ldots, s_m$, $m \geq 1$, of states such that $s_1 = s$, $s_n = t$, and s_{i+1} is directly connected to s_i for $1 \leq i < m$. Essentially, t is connected to s if there is a path in A from s to t, where A is viewed as an edge-labeled directed graph. Note that the edge labels are ignored in this definition. It is easy to see that $L(A)$ is nonempty iff there are states $s \in S^0$ and $t \in F$ such that t is connected to s. Thus, automata nonemptiness is equivalent to *graph reachability*. The claims now follow from the following observations:

1. A breadth-first-search algorithm can construct in linear time the set of all states conncected to a state in S^0 [CLR90]. A is nonempty iff this set intersects F nontrivially.
2. Graph reachability can be tested in *nondeterministic* logarithmic space. The algorithm simply guesses a state $s_0 \in S^0$, then guesses a state s_1 that is directly connected to s_0, then guesses a state s_2 that is directly connected to s_1, etc., until it reaches a state $t \in F$. (Recall that a nondeterministic algorithm accepts if there *is* a sequence of guesses that leads to acceptance. We do not care here about sequences of guesses that do not lead to acceptance [GJ79].) At each step the algorithm needs to remember only the current state and the next state; thus, if there are n states the algorithm needs to keep in memory $O(\log n)$ bits, since $\log n$ bits suffice to describe one state. On the other hand, graph reachability is also NLOGSPACE-hard [Jon75].

▮

Proposition 12. [MS72]

1. *The nonuniversality problem for automata is decidable in exponential time.*
2. *The nonuniversality problem for automata is PSPACE-complete.*

Proof: Note that $L(A) \neq \Sigma^*$ iff $\Sigma^* - L(A) \neq \emptyset$ iff $L(\overline{A}) \neq \emptyset$, where $\overline{A}$ is the complementary automaton of A (see Section 2.1). Thus, to test A for nonuniversality, it suffices to test $\overline{A}$ for nonemptiness. Recall that $\overline{A}$ is exponentially bigger than A. Since nonemptiness can be tested in linear time, it follows that nonuniversality can be tested in exponential time. Also, since nonemptiness can be tested in nondeterministic logarithmic space, nonuniversality can be tested in polynomial space.

The latter argument requires some care. We cannot simply construct $\overline{A}$ and then test it for nonemptiness, since $\overline{A}$ is exponentially big. Instead, we construct $\overline{A}$ "on-the-fly"; whenever the nonemptiness algorithm wants to move from a state t_1 of $\overline{A}$ to a state t_2, the algorithm guesses t_2 and checks that it is directly connected to t_1. Once this has been verified, the algorithm can discard t_1. Thus, at each step the algorithm needs to keep in memory at most two states of $\overline{A}$ and there is no need to generate all of $\overline{A}$ at any single step of the algorithm.

This yields a nondeterministic polynomial space algorithm. To eliminate nondeterminism, we appeal to a well-known theorem of Savitch [Sav70] which states that

$NSPACE(f(n)) \subseteq DSPACE(f(n)^2)$, for $f(n) \geq \log n$; that is, any nondeterministic algorithm that uses at least logarithmic space can be simulated by a deterministic algorithm that uses at most quadratically larger amount of space. In particular, any nondeterministic polynomial-space algorithm can be simulated by a deterministic polynomial-space algorithm.

To prove PSPACE-hardness, it can be shown that any PSPACE-hard problem can be reduced to the nonuniversality problem. That is, there is a logarithmic-space algorithm that given a polynomial-space-bounded Turing machine M and a word w outputs an automaton $A_{M,w}$ such that M accepts w iff $A_{M,w}$ is non-universal [MS72, HU79]. ∎

2.4 Automata on Infinite Words – Algorithms

The results for Büchi automata are analogous to the results in Section 2.3.

Proposition 13.

1. [EL85b, EL85a] *The nonemptiness problem for Büchi automata is decidable in linear time.*
2. [VW94] *The nonemptiness problem for Büchi automata is NLOGSPACE-complete.*

Proof: Let $A = (\Sigma, S, S^0, \rho, F)$ be the given automaton. We claim that $L_\omega(A)$ is nonempty iff there are states $s_0 \in S^0$ and $t \in F$ such that t is connected to s_0 and t is connected to itself. Suppose first that $L_\omega(A)$ is nonempty. Then there is an accepting run $r = s_0, s_1, \ldots$ of A on some input word. Clearly, s_{i+1} is directly connected to s_i for all $i \geq 0$. Thus, s_j is connected to s_i whenever $i < j$. Since r is accepting, some $t \in F$ occurs in r infinitely often; in particular, there are i, j, where $0 < i < j$, such that $t = s_i = s_j$. Thus, t is connected to $s_0 \in S^0$ and t is also connected to itself. Conversely, suppose that there are states $s_0 \in S^0$ and $t \in F$ such that t is connected to s_0 and t is connected to itself. Since t is connected to s_0, there are a sequence of states $s_1, \ldots, s_k$ and a sequence of symbols $a_1, \ldots, a_k$ such that $s_k = t$ and $s_i \in \rho(s_{i-1}, a_i)$ for $1 \leq i \leq k$. Similarly, since t is connected to itself, there are a sequence of states $t_0, t_1, \ldots, t_l$ and a sequence of symbols $b_1, \ldots, b_l$ such that $t_0 = t_k = t$ and $t_i \in \rho(t_{i-1}, b_i)$ for $1 \leq i \leq l$. Thus, $(s_0, s_1, \ldots, s_{k-1})(t_0, t_1, \ldots, t_{l-1})^\omega$ is an accepting run of A on $(a_1, \ldots, a_k)(b_1, \ldots, b_l)^\omega$, so A is nonempty.

Thus, Büchi automata nonemptiness is also reducible to graph reachability.

1. A depth-first-search algorithm can construct a decomposition of the graph into strongly connected components [CLR90]. A is nonempty iff from a component that intersects S^0 nontrivially it is possible to reach a nontrivial component that intersects F nontrivially. (A strongly connected component is nontrivial if it contains an edge, which means, since it is strongly connected, that it contains a cycle).
2. The algorithm simply guesses a state $s_0 \in S^0$, then guesses a state s_1 that is directly connected to s_0, then guesses a state s_2 that is directly connected to s_1, etc., until it reaches a state $t \in F$. At that point the algorithm remembers t and it continues to move nondeterministically from a state s to a state s' that is directly connected to s until it reaches t again. Clearly, the algorithm needs only a logarithmic memory, since it needs to remember at most a description of three states at each step.

NLOGSPACE-hardness follows from NLOGSPACE-hardness of nonemptiness for automata on finite words.

▮

Proposition 14. [SVW87]

1. *The nonuniversality problem for Büchi automata is decidable in exponential time.*
2. *The nonuniversality problem for Büchi automata is PSPACE-complete.*

Proof: Again $L_\omega(A) \neq \Sigma^\omega$ iff $\Sigma^\omega - L_\omega(A) \neq \emptyset$ iff $L_\omega(\overline{A}) \neq \emptyset$, where $\overline{A}$ is the complementary automaton of A (see Section 2.2). Thus, to test A for nonuniversality, it suffices to test $\overline{A}$ for nonemptiness. Since $\overline{A}$ is exponentially bigger than A and nonemptiness can be tested in linear time, it follows that nonuniversality can be tested in exponential time. Also, since nonemptiness can be tested in nondeterministic logarithmic space, nonuniversality can be tested in polynomial space. Again, the polynomial-space algorithm constructs $\overline{A}$ "on-the-fly".

PSPACE-hardness follows easily from the PSPACE-hardness of the universality problem for automata on finite words [Wol82]. ▮

2.5 Automata on Finite Words – Alternation

Nondeterminism gives a computing device the power of existential choice. Its dual gives a computing device the power of universal choice. (Compare this to the complexity classes NP and co-NP [GJ79]). It is therefore natural to consider computing devices that have the power of both existential choice and universal choice. Such devices are called *alternating*. Alternation was studied in [CKS81] in the context of Turing machines and in [BL80, CKS81] for finite automata. The alternation formalisms in [BL80] and [CKS81] are different, though equivalent. We follow here the formalism of [BL80].

For a given set X, let $\mathcal{B}^+(X)$ be the set of positive Boolean formulas over X (i.e., Boolean formulas built from elements in X using $\wedge$ and $\vee$), where we also allow the formulas **true** and **false**. Let $Y \subseteq X$. We say that Y *satisfies* a formula $\theta \in \mathcal{B}^+(X)$ if the truth assignment that assigns *true* to the members of Y and assigns *false* to the members of $X - Y$ satisfes θ. For example, the sets $\{s_1, s_3\}$ and $\{s_1, s_4\}$ both satisfy the formula $(s_1 \vee s_2) \wedge (s_3 \vee s_4)$, while the set $\{s_1, s_2\}$ does not satisfy this formula.

Consider a nondeterministic automaton $A = (\Sigma, S, S^0, \rho, F)$. The transition function ρ maps a state $s \in S$ and an input symbol $a \in \Sigma$ to a set of states. Each element in this set is a possible nondeterministic choice for the automaton's next state. We can represent ρ using $\mathcal{B}^+(S)$; for example, $\rho(s, a) = \{s_1, s_2, s_3\}$ can be written as $\rho(s, a) = s_1 \vee s_2 \vee s_3$. In alternating automata, $\rho(s, a)$ can be an arbitrary formula from $\mathcal{B}^+(S)$. We can have, for instance, a transition

$$\delta(s, a) = (s_1 \wedge s_2) \vee (s_3 \wedge s_4),$$

meaning that the automaton accepts the word aw, where a is a symbol and w is a word, when it is in the state s, if it accepts the word w from both s_1 and s_2 or from both s_3 and

s_4. Thus, such a transition combines the features of existential choice (the disjunction in the formula) and universal choice (the conjunctions in the formula).

Formally, an *alternating automaton* is a tuple $A = (\Sigma, S, s^0, \rho, F)$, where Σ is a finite nonempty alphabet, S is a finite nonempty set of states, $s^0 \in S$ is the initial state (notice that we have a unique initial state), F is a set of accepting states, and $\rho : S \times \Sigma \rightarrow \mathcal{B}^+(S)$ is a transition function.

Because of the universal choice in alternating transitions, a run of an alternating automaton is a tree rather than a sequence. A *tree* is a (finite or infinite) connected directed graph, with one node designated as the *root* and denoted by ε, and in which every non-root node has a unique parent (s is the *parent* of t and t is a *child* of s if there is an edge from s to t) and the root ε has no parent. The *level* of a node x, denoted $|x|$, is its distance from the root ε; in particular, $|\varepsilon| = 0$. A *branch* $\beta = x_0, x_1, \ldots$ of a tree is a maximal sequence of nodes such that x_0 is the root ε and x_i is the parent of x_{i+1} for all $i > 0$. Note that β can be finite or infinite. A *Σ-labeled tree*, for a finite alphabet Σ, is a pair $(\tau, \mathcal{T})$, where τ is a tree and $\mathcal{T}$ is a mapping from $nodes(\tau)$ to Σ that assigns to every node of τ a label in Σ. We often refer to $\mathcal{T}$ as the labeled tree. A branch $\beta = x_0, x_1, \ldots$ of $\mathcal{T}$ defines an infinite word $\mathcal{T}(\beta) = \mathcal{T}(x_0), \mathcal{T}(x_1), \ldots$ consisting of the sequence of labels along the branch.

Formally, a run of A on a finite word $w = a_0, a_1, \ldots, a_{n-1}$ is a finite S-labeled tree r such that $r(\varepsilon) = s^0$ and the following holds:

> if $|x| = i < n$, $r(x) = s$, and $\rho(s, a_i) = \theta$, then x has k children $x_1, \ldots, x_k$, for some $k \leq |S|$, and $\{r(x_1), \ldots, r(x_k)\}$ satisfies θ.

For example, if $\rho(s^0, a_0)$ is $(s_1 \vee s_2) \wedge (s_3 \vee s_4)$, then the nodes of the run tree at level 1 include the label s_1 or the label s_2 and also include the label s_3 or the label s_4. Note that the depth of r (i.e., the maximal level of a node in r) is at most n, but not all branches need to reach such depth, since if $\rho(r(x), a_i) =$ **true**, then x does not need to have any children. On the other hand, if $|x| = i < n$ and $r(x) = s$, then we cannot have $\rho(s, a_i) =$ **false**, since **false** is not satisfiable.

The run tree r is *accepting* if all nodes at depth n are labeled by states in F. Thus, a branch in an accepting run has to hit the **true** transition or hit an accepting state after reading all the input word.

What is the relationship between alternating automata and nondeterministic automata? It turns out that just as nondeterministic automata have the same expressive power as deterministic automata but they are exponentially more succinct, alternating automata have the same expressive power as nondeterministic automata but they are exponentially more succinct.

We first show that alternating automata are at least as expressive and as succinct as nondeterministic automata.

Proposition 15. [BL80, CKS81, Lei81] *Let A be a nondeterministic automaton. Then there is an alternating automaton A_n such that $L(A_a) = L(A)$.*

Proof: Let $A = (\Sigma, S, S^0, \rho, F)$. Then $A_a = (\Sigma, S \cup \{s^0\}, s^0, \rho_a, F)$, where s^0 is a new state, and ρ_a is defined as follows, for $b \in \Sigma$ and $s \in S$:

– $\rho_a(s^0, b) = \bigvee_{t \in S^0, t' \in \rho(t,b)} t'$,

– $\rho_a(s, b) = \bigvee_{t \in \rho(s,b)} t$.

(We take an empty disjunction in the definition of ρ_a to be equivalent to **false**.) Essentially, the transitions of A are viewed as disjunctions in A_a. A special treatment is needed for the initial state, since we allow a set of initial states in nondeterministic automata, but only a single initial state in alternating automata. ▮

Note that A_a has essentially the same size as A; that is, the descriptions of A_a and A have the same length.

We now show that alternating automata are not more expressive than nondeterministic automata.

Proposition 16. [BL80, CKS81, Lei81] *Let A be an alternating automaton. Then there is a nondeterministic automaton A_n such that $L(A_n) = L(A)$.*

Proof: Let $A = (\Sigma, S, s^0, \rho, F)$. Then $A_n = (\Sigma, S_n, \{\{s^0\}\}, \rho_n, F_n)$, where $S_n = 2^S$, $F_n = 2^F$, and

$$\rho_n(T, a) = \{T' \mid T' \text{ satisfies } \bigwedge_{t \in T} \rho(t, a)\}.$$

(We take an empty conjunction in the definition of ρ_n to be equivalent to **true**; thus, $\emptyset \in \rho_n(\emptyset, a)$.)

Intuitively, A_n guesses a run tree of A. At a given point of a run of A_n, it keeps in its memory a whole level of the run tree of A. As it reads the next input symbol, it guesses the next level of the run tree of A. ▮

The translation from alternating automata to nondeterministic automata involves an exponential blow-up. As shown in [BL80, CKS81, Lei81], this blow-up is unavoidable. For example, fix some $n \geq 1$, and let $\Sigma = \{a, b\}$. Let Γ_n be the set of all words that have two different symbols at distance n from each other. That is,

$$\Gamma_n = \{uavbw \mid u, w \in \Sigma^* \text{ and } v \in \Sigma^{n-1}\} \cup \{ubvaw \mid u, w \in \Sigma^* \text{ and } v \in \Sigma^{n-1}\}.$$

It is easy to see that Γ_n is accepted by the nondeterministic automaton $A = (\Sigma, \{p, q\} \cup \{1, \ldots, n\} \times \Sigma, \{p\}, \rho, \{q\})$, where $\rho(p, a) = \{p, \langle 1, a \rangle\}$, $\rho(p, b) = \{p, \langle 1, b \rangle\}$, $\rho(\langle a, i \rangle, x) = \{\langle a, i+1 \rangle\}$ and $\rho(\langle b, i \rangle, x) = \{\langle b, i+1 \rangle\}$ for $x \in \Sigma$ and $0 < i < n$, $\rho(\langle a, n \rangle, a) = \emptyset$, $\rho(\langle a, n \rangle, b) = \{q\}$, $\rho(\langle b, n \rangle, b) = \emptyset$, $\rho(\langle b, n \rangle, a) = \{q\}$, and $\rho(q, x) = \{q\}$ for $x \in \Sigma$. Intuitively, A guesses a position in the input word, reads the input symbol at that position, moves n positions to the right, and checks that it contains a different symbol. Note that A has $2n + 2$ states. By Propositions 15 and 17 (below), there is an alternating automaton A_a with $2n + 3$ states that accepts the complementary language $\overline{\Gamma_n} = \Sigma^* - \Gamma_n$.

Suppose that we have a nondeterministic automaton $A_{nd} = (\Sigma, S, S^0, \rho_{nd}, F)$ with fewer than 2^n states that accepts $\overline{\Gamma_n}$. Thus, A_n accepts all words ww, where $w \in \Sigma^n$. Let $s_w^0, \ldots, s_w^{2n}$ an accepting run of A_{nd} on ww. Since $|S| < 2^n$, there are two distinct word $u, v \in \Sigma^n$ such that $s_u^n = s_v^n$. Thus, $s_u^0, \ldots, s_u^n, s_v^{n+1}, \ldots, s_v^{2n}$ is an accepting run of A_{nd} on uv, but $uv \notin \overline{\Gamma_n}$ since it must have two different symbols at distance n from each other.

One advantage of alternating automata is that it is easy to complement them. We first need to define the *dual* operation on formulas in $\mathcal{B}^+(X)$. Intuitively, the dual $\overline{\theta}$ of a formula $\overline{\theta}$ is obtained from θ by switching $\vee$ and $\wedge$, and by switching **true** and **false**. For example, $\overline{x \vee (y \wedge z)} = x \wedge (y \vee z)$. (Note that we are considering formulas in $\mathcal{B}^+(X)$, so we cannot simply apply negation to these formulas.) Formally, we define the dual operation as follows:

– $\overline{x} = x$, for $x \in X$,
– $\overline{\textbf{true}} = \textbf{false}$,
– $\overline{\textbf{false}} = \textbf{true}$,
– $\overline{(\alpha \wedge \beta)} = (\overline{\alpha} \vee \overline{\beta})$, and
– $\overline{(\alpha \vee \beta)} = (\overline{\alpha} \wedge \overline{\beta})$.

Suppose now that we are given an alternating automaton $A = (\Sigma, S, s^0, \rho, F)$. Define $\overline{A} = (\Sigma, S, s^0, \overline{\rho}, S - F)$, where $\overline{\rho}(s, a) = \overline{\rho(s, a)}$ for all $s \in S$ and $a \in \Sigma$. That is, $\overline{\rho}$ is the dualized transition function.

Proposition 17. [BL80, CKS81, Lei81] *Let A be an alternating automaton. Then $L(\overline{A}) = \Sigma^* - L(A)$.*

By combining Propositions 11 and 16, we can obtain a nonemptiness test for alternating automata.

Proposition 18. [CKS81]

1. *The nonemptiness problem for alternating automata is decidable in exponential time.*
2. *The nonemptiness problem for alternating automata is PSPACE-complete.*

Proof: All that remains to be shown is the PSPACE-hardness of nonemptiness. Recall that PSPACE-hardness of nonuniversality was shown in Proposition 12 by a generic reduction. That is, there is a logarithmic-space algorithm that given a polynomial-space-bounded Turing machine M and a word w outputs an automaton $A_{M,w}$ such that M accepts w iff $A_{M,w}$ is nonuniversal. By Proposition 15, there is an alternating automaton A_a such that $L(A_a) = L(A_{M,w})$ and A_a has the same size as $A_{M,w}$. By Proposition 17, $L(\overline{A_a}) = \Sigma^* - L(A_a)$. Thus, $A_{M,w}$ is nonuniversal iff $\overline{A_a}$ is nonempty. ∎

2.6 Automata on Infinite Words - Alternation

We saw earlier that a nondeterministic automaton can be viewed both as an automaton on finite words and as an automaton on infinite words. Similarly, an alternating automaton can also be viewed as an automaton on infinite words, in which case it is called an *alternating Büchi automaton* [MS87].

Let $A = (\Sigma, S, s^0, \rho, F)$ be an alternating Büchi automaton. A run of A on an infinite word $w = a_0, a_1, \ldots$ is a (possibly infinite) S-labeled tree r such that $r(\varepsilon) = s^0$ and the following holds:

> if $|x| = i$, $r(x) = s$, and $\rho(s, a_i) = \theta$, then x has k children $x_1, \ldots, x_k$, for some $k \leq |S|$, and $\{r(x1), \ldots, r(xk)\}$ satisfies θ.

The run r is *accepting* if every infinite branch in r includes infinitely many labels in F. Note that the run can also have finite branches; if $|x| = i$, $r(x) = s$, and $\rho(s, a_i) = \mathbf{true}$, then x does not need to have any children.

We with alternating automata, alternating Büchi automata are as expressive as nondeterministic Büchi automata. We first show that alternating automata are at least as expressive and as succinct as nondeterministic automata. The proof of the following proposition is identical to the proof of Proposition 19.

Proposition 19. [MS87] *Let A be a nondeterministic Büchi automaton. Then there is an alternating Büchi automaton A_a such that $L_\omega(A_a) = L_\omega(A)$.*

As the reader may expect by now, alternating Büchi automata are not more expressive than nondeterministic Büchi automata. The proof of this fact, however, is more involved than the proof in the finite-word case.

Proposition 20. [MH84] *Let A be an alternating Büchi automaton. Then there is a nondeterministic Büchi automaton A_n such that $L_\omega(A_n) = L_\omega(A)$.*

Proof: As in the finite-word case, A_n guesses a run of A. At a given point of a run of A_n, it keeps in its memory a whole level of the run of A (which is a tree). As it reads the next input symbol, it guesses the next level of the run tree of A. The nondeterministic automaton, however, also has to keep information about occurrences of accepting states in order to make sure that every infinite branch hits accepting states infinitely often. To that end, A_n partitions every level of the run of A into two sets to distinguish between branches that hit F recently and branches that did not hit F recently.

Let $A = (\Sigma, S, s^0, \rho, F)$. Then $A_n = (\Sigma, S_n, S^0, \rho_n, F_n)$, where $S_n = 2^S \times 2^S$ (i.e., each state is a pair of sets of states of A), $S^0 = \{(\{s^0\}, \emptyset)\}$ (i.e., the single initial state is pair consisting of the singleton set $\{s^0\}$ and the empty set), $F_n = \{\emptyset\} \times 2^S$, and

– for $U \neq \emptyset$,

$$\begin{aligned}\rho_n((U,V),a) = \{(U',V') \mid \text{ there exist } X, Y \subseteq S \text{ such that}\\ X \text{ satisfies } \textstyle\bigwedge_{t\in U} \rho(t,a),\\ Y \text{ satisfies } \textstyle\bigwedge_{t\in V} \rho(t,a),\\ U' = X - F, \text{ and } V' = Y \cup (X \cap F)\},\end{aligned}$$

–

$$\begin{aligned}\rho_n((\emptyset,V),a) = \{(U',V') \mid \text{ there exists } Y \subseteq S \text{ such that}\\ Y \text{ satisfies } \textstyle\bigwedge_{t\in V} \rho(t,a),\\ U' = Y - F, \text{ and } V' = Y \cap F\}.\end{aligned}$$

The proof that this construction is correct requires a careful analysis of accepting runs of A. ▮

An important feature of this construction is that the blowup is exponential.

While complementation of alternating automata is easy (Proposition 17), this is not the case for alternating Büchi automata. Here we run into the same difficulty that we ran into in Section 2.2: not going infinitely often through accepting states is not the same as going infinitely often through non-accepting states. >From Propositions 7, 19 and 20.

it follows that alternating Büchi automata are closed under complement, but the precise complexity of complementation in this case is not known.

Finally, by combining Propositions 13 and 20, we can obtain a nonemptiness test for alternating Büchi automata.

Proposition 21.

1. *The nonemptiness problem for alternating Büchi automata is decidable in exponential time.*
2. *The nonemptiness problem for alternating Büchi automata is PSPACE-complete.*

Proof: All that remains to be shown is the PSPACE-hardness of nonemptiness. We show that the nonemptiness problem for alternating automata is reducible to the nonemptiness problem for alternating Büchi automata. Let $A = (\Sigma, S, s^0, \rho, F)$ be an alternating automaton. Consider the alternating Büchi automaton $A' = (\Sigma, S, s^0, \rho', \emptyset)$, where $\rho'(s,a) = \rho(s,a)$ for $s \in S - F$ and $a \in \Sigma$, and $\rho'(s,a) =$ **true** for $s \in F$ and $a \in \Sigma$.

We claim that $L(A) \neq \emptyset$ iff $L_\omega(A') \neq \emptyset$. Suppose first that $w \in L(A)$ for some $w \in \Sigma^*$. Then there is an accepting run r of A on w. But then r is also an accepting run of A' on wu for all $u \in \Sigma^\omega$, because $\rho'(s,a) =$ **true** for $s \in F$ and $a \in \Sigma$, so $wu \in L_\omega(A')$. Suppose, on the other hand, that $w \in L_\omega(A)$ for some $w \in \Sigma^\omega$. Then there is an accepting run r of A' on w. Since A' has no accepting state, r cannot have infinite branches, so by König's Lemma it must be finite. Thus, there is a finite prefix u of w such that r is an accepting run of A on u, so $u \in L(A)$. ▮

3 Linear Temporal Logic and Automata on Infinite Words

Formulas of *linear-time propositional temporal logic* (LTL) are built from a set $Prop$ of atomic propositions and are closed under the application of Boolean connectives, the unary temporal connective X (next), and the binary temporal connective U (until) [Pnu77, GPSS80]. LTL is interpreted over *computations*. A computation is a function $\pi : N \rightarrow 2^{Prop}$, which assigns truth values to the elements of $Prop$ at each time instant (natural number). For a computation π and a point $i \in \omega$, we have that:

– $\pi, i \models p$ for $p \in Prop$ iff $p \in \pi(i)$.
– $\pi, i \models \xi \wedge \psi$ iff $\pi, i \models \xi$ and $\pi, i \models \psi$.
– $\pi, i \models \neg\varphi$ iff not $\pi, i \models \varphi$
– $\pi, i \models X\varphi$ iff $\pi, i+1 \models \varphi$.
– $\pi, i \models \xi U \psi$ iff for some $j \geq i$, we have $\pi, j \models \psi$ and for all k, $i \leq k < j$, we have $\pi, k \models \xi$.

Thus, the formula $\mathbf{true} U \varphi$, abbreviated as $F\varphi$, says that φ holds *eventually*, and the formula $\neg F \neg \varphi$, abbreviated $G\varphi$, says that φ holds *henceforth*. For example, the formula $G(\neg \mathsf{request} \vee (\mathsf{request} U \mathsf{grant}))$ says that whenever a request is made it holds continuously until it is eventually granted. We will say that π *satisfies* a formula φ, denoted $\pi \models \varphi$, iff $\pi, 0 \models \varphi$.

Computations can also be viewed as infinite words over the alphabet 2^{Prop}. We shall see that the set of computations satisfying a given formula are exactly those accepted by some finite automaton on infinite words. This fact was proven first in [SPH84]. The proof there is by induction on structure of formulas. Unfortunately, certain inductive steps involve an exponential blow-up (e.g., negation corresponds to complementation, which we have seen to be exponential). As a result, the complexity of that translation is *nonelementary*, i.e., it may involve an unbounded stack of exponentials (that is, the complexity bound is of the form

$$2^{\cdot^{\cdot^{2^n}}},$$

where the height of the stack is n.)

The following theorem establishes a very simple translation between LTL and alternating Büchi automata.

Theorem 22. [MSS88, Var94] *Given an LTL formula φ, one can build an alternating Büchi automaton $A_\varphi = (\Sigma, S, s^0, \rho, F)$, where $\Sigma = 2^{Prop}$ and $|S|$ is in $O(|\varphi|)$, such that $L_\omega(A_\varphi)$ is exactly the set of computations satisfying the formula φ.*

Proof: The set S of states consists of all subformulas of φ and their negation (we identify the formula $\neg\neg\psi$ with ψ). The initial state s^0 is φ itself. The set F of accepting states consists of all formulas in S of the form $\neg(\xi U \psi)$. It remains to define the transition function ρ.

In this construction, we use a variation of the notion of dual that we used in Section 2.5. Here, the dual $\overline{\theta}$ of a formula is obtained from θ by switching $\vee$ and $\wedge$, by switching **true** and **false**, and, in addition, by negating subformulas in S, e.g., $\overline{\neg p \vee (q \wedge Xq)}$ is $p \wedge (\neg q \vee \neg Xq)$. More formally,

– $\overline{\xi} = \neg\xi$, for $\xi \in S$,
– $\overline{\textbf{true}} = \textbf{false}$,
– $\overline{\textbf{false}} = \textbf{true}$,
– $\overline{(\alpha \wedge \beta)} = (\overline{\alpha} \vee \overline{\beta})$, and
– $\overline{(\alpha \vee \beta)} = (\overline{\alpha} \wedge \overline{\beta})$.

We can now define ρ:

– $\rho(p, a) = \textbf{true}$ if $p \in a$,
– $\rho(p, a) = \textbf{false}$ if $p \notin a$,
– $\rho(\xi \wedge \psi, a) = \rho(\xi, a) \wedge \rho(\psi, a)$,
– $\rho(\neg\psi, a) = \overline{\rho(\psi, a)}$,
– $\rho(X\psi, a) = \psi$,
– $\rho(\xi U \psi, a) = \rho(\psi, a) \vee (\rho(\xi, a) \wedge \xi U \psi)$.

Note that $\rho(\psi, a)$ is defined by induction on the structure of ψ.

Consider now a run r of A_φ. It is easy to see that r can have two types of infinite branches. Each infinite branch is labeled from some point on by a formula of the form $\xi U \psi$ or by a formula of the form $\neg(\xi U \psi)$. Since $\rho(\neg(\xi U \psi), a) = \overline{\rho(\psi, a)} \wedge (\overline{\rho(\xi, a)} \vee \neg(\xi U \psi))$, an infinite branch labeled from some point by $\neg(\xi U \psi)$ ensures that $\xi U \psi$ indeed fails at that point, since ψ fails from that point on. On the other hand, an infinite branch labeled from some point by $\xi U \psi$ does not ensure that $\xi U \psi$ holds at that point, since it does not ensure that ψ eventually holds. Thus, while we should allow infinite branches labeled by $\neg(\xi U \psi)$, we should not allow infinite branches labeled by $\xi U \psi$. This is why we defined F to consists of all formulas in S of the form $\neg(\xi U \psi)$. ∎

Example 1. Consider the formula $\varphi = (X\neg p)Uq$. The alternating Büchi automaton associated with φ is $A_\varphi = (2^{\{p,q\}}, \{\varphi, \neg\varphi, X\neg p, \neg X\neg p, \neg p, p, q, \neg q\}, \varphi, \rho, \{\neg\varphi\})$, where ρ is described in the following table.

s	$\rho(s, \{p, q\})$	$\rho(s, \{p\})$	$\rho(s, \{q\})$	$\rho(s, \emptyset)$
φ	**true**	$\neg p \wedge \varphi$	**true**	$\neg p \wedge \varphi$
$\neg\varphi$	**false**	$p \vee \neg\varphi$	**false**	$p \vee \neg\varphi$
$X\neg p$	$\neg p$	$\neg p$	$\neg p$	$\neg p$
$\neg X\neg p$	p	p	p	p
$\neg p$	**false**	**false**	**true**	**true**
p	**true**	**true**	**false**	**false**
q	**true**	**false**	**true**	**false**
$\neg q$	**false**	**true**	**false**	**true**

In the state φ, if q does not hold in the present state, then A_φ requires both $X\neg p$ to be satisfied in the present state (that is, $\neg p$ has to be satisfied in next state), and φ to be satisfied in the next state. As $\varphi \notin F$, A_φ should eventually reach a state that satisfies q. Note that many of the states, e.g., the subformulas $X\neg p$ and q, are not *reachable*; i.e., they do not appear in any run of A_φ. ∎

By applying Proposition 20, we now get:

Corollary 23. [VW94] *Given an LTL formula φ, one can build a Büchi automaton $A_\varphi = (\Sigma, S, S^0, \rho, F)$, where $\Sigma = 2^{Prop}$ and $|S|$ is in $2^{O(|\varphi|)}$, such that $L_\omega(A_\varphi)$ is exactly the set of computations satisfying the formula φ.*

The proof of Corollary 23 in [VW94] is direct and does not go through alternating Büchi automata. The advantage of the proof here is that it separates the logic from the combinatorics. Theorem 22 handles the logic, while Proposition 20 handles the combinatorics.

Example 2. Consider the formula $\varphi = FGp$, which requires p to hold from some point on. The Büchi automaton associated with φ is $A_\varphi = (2^{\{p\}}, \{0, 1\}, \{0\}, \rho, \{1\})$, where ρ is described in the following table.

s	$\rho(s, \{p\})$	$\rho(s, \emptyset)$
0	{0,1}	{0}
1	{1}	$\emptyset$

The automaton A_φ can stay forever in the state 0. Upon reading p, however, A_φ can choose to go to the state 1. Once A_φ has made that transition, it has to keep reading p, otherwise it rejects. Note that A_φ has to make the transition to the state 1 at *some* point, since the state 0 is not accepting. Thus, A_φ accepts precisely when p holds from some point on. ∎

4 Applications

4.1 Satisfiability

An LTL formula φ is *satisfiable* if there is some computation π such that $\pi \models \varphi$. An unsatisfiable formula is uninteresting as a specification, so unsatisfiability most likely indicates an erroneous specification. The *satisfiability problem* for LTL is to decide, given an LTL formula φ, whether φ is satisfiable.

Theorem 24. [SC85] *The satisfiability problem for LTL is PSPACE-complete.*

Proof: By Corollary 23, given an LTL formula φ, we can construct a Büchi automaton A_φ, whose size is exponential in the length of φ, that accepts precisely the computations that satisfy φ. Thus, φ is satisfiable iff A_φ is nonempty. This reduces the satisfiability problem to the nonemptiness problem. Since nonemptiness of Büchi automata can be tested in nondeterministic logarithmic space (Proposition 13) and since A_φ is of exponential size, we get a polynomial-space algorithm (again, the algorithm constructs A_φ "on-the-fly").

To prove PSPACE-hardness, it can be shown that any PSPACE-hard problem can be reduced to the satisfiability problem. That is, there is a logarithmic-space algorithm that given a polynomial-space-bounded Turing machine M and a word w outputs an LTL formula $\varphi_{M,w}$ such that M accepts w iff $\varphi_{M,w}$ is satisfiable. ▮

An LTL formula φ is *valid* if for every computation π we have that $\pi \models \varphi$. A valid formula is also uninteresting as a specification. The *validity problem* for LTL is to decide, given an LTL formula φ, whether φ is valid. It is easy to see that φ is valid iff $\neg\varphi$ is not satisfiable. Thus, the validity problem for LTL is also PSPACE-complete.

4.2 Verification

We focus here on *finite-state* programs, i.e., programs in which the variables range over finite domains. The significance of this class follows from the fact that a significant number of the communication and synchronization protocols studied in the literature are in essence finite-state programs [Liu89, Rud87]. Since each state is characterized by a finite amount of information, this information can be described by certain *atomic propositions*. This means that a finite-state program can be specified using *propositional* temporal logic. Thus, we assume that we are given a finite-state program and an LTL formula that specifies the legal computations of the program. The problem is to check whether all computations of the program are legal. Before going further, let us define these notions more precisely.

A finite-state program over a set $Prop$ of atomic propositions is a structure of the form $P = (W, w_0, R, V)$, where W is a finite set of states, $w_0 \in W$ is the initial state, $R \subseteq W^2$ is a total accessibility relation, and $V : W \rightarrow 2^{Prop}$ assigns truth values to propositions in $Prop$ for each state in W. The intuition is that W describes all the states that the program could be in (where a state includes the content of the memory, registers, buffers, location counter, etc.), R describes all the possible transitions between states (allowing for nondeterminism), and V relates the states to the propositions (e.g., it tells us in what states the proposition `request` is true). The assumption that R is total (i.e., that every state has a child) is for technical convenience. We can view a terminated execution as repeating forever its last state.

Let $\mathbf{u}$ be an infinite sequence $u_0, u_1 \ldots$ of states in W such that $u_0 = w_0$, and $u_i R u_{i+1}$ for all $i \geq 0$. Then the sequence $V(u_0), V(u_1) \ldots$ is a *computation* of P. We say that P *satisfies* an LTL formula φ if *all* computations of P satisfy φ. The *verification problem* is to check whether P satisfies φ.

The complexity of the verification problem can be measured in three different ways. First, one can fix the specification φ and measure the complexity with respect to the size of the program. We call this measure the *program-complexity* measure. More precisely, the program complexity of the verification problem is the complexity of the sets $\{P \mid P \text{ satisfies } \varphi\}$ for a fixed φ. Secondly, one can fix the program P and measure the complexity with respect to the size of the specification. We call this measure the *specification-complexity* measure. More precisely, the specification complexity of the verification problem is the complexity of the sets $\{\varphi \mid P \text{ satisfies } \varphi\}$ for a fixed P. Finally, the complexity in the combined size of the program and the specification is the *combined complexity*.

Let C be a complexity class. We say that the program complexity of the verification problem is in C if $\{P \mid P \text{ satisfies } \varphi\} \in C$ for any formula φ. We say that the program complexity of the verification problem is hard for C if $\{P \mid P \text{ satisfies } \varphi\}$ is hard for C for some formula φ. We say that the program complexity of the verification problem is complete for C if it is in C and is hard for C. Similarly, we say that the specification complexity of the verification problem is in C if $\{\varphi \mid P \text{ satisfies } \varphi\} \in C$ for any program P, we say that the specification complexity of the verification problem is hard for C if $\{\varphi \mid P \text{ satisfies } \varphi\}$ is hard for C for some program P, and we say that the specification complexity of the verification problem is complete for C if it is in C and is hard for C.

We now describe the automata-theoretic approach to the verification problem. A finite-state program $P = (W, w_0, R, V)$ can be viewed as a Büchi automaton $A_P = (\Sigma, W, \{w_0\}, \rho, W)$, where $\Sigma = 2^{Prop}$ and $s' \in \rho(s, a)$ iff $(s, s') \in R$ and $a = V(s)$. As this automaton has a set of accepting states equal to the whole set of states, any infinite run of the automaton is accepting. Thus, $L_\omega(A_P)$ is the set of computations of P.

Hence, for a finite-state program P and an LTL formula φ, the verification problem is to verify that all infinite words accepted by the automaton A_P satisfy the formula φ. By Corollary 23, we know that we can build a Büchi automaton A_φ that accepts exactly the computations satisfying the formula φ. The verification problem thus reduces to the automata-theoretic problem of checking that all computations accepted by the automaton A_P are also accepted by the automaton A_φ, that is $L_\omega(A_P) \subseteq L_\omega(A_\varphi)$. Equivalently, we need to check that the automaton that accepts $L_\omega(A_P) \cap L_\omega(\overline{A_\varphi})$ is empty, where

$$L_\omega(\overline{A_\varphi}) = \overline{L_\omega(A_\varphi)} = \Sigma^\omega - L_\omega(A_\varphi).$$

First, note that, by Corollary 23, $L_\omega(\overline{A_\varphi}) = L_\omega(A_{\neg\varphi})$ and the automaton $A_{\neg\varphi}$ has $2^{O(|\varphi|)}$ states. (A straightforward approach, starting with the automaton A_φ and then using Proposition 7 to complement it, would result in a doubly exponential blow-up.) To get the intersection of the two automata, we use Proposition 6. Consequently, we can build an automaton for $L_\omega(A_P) \cap L_\omega(A_{\neg\varphi})$ having $|W| \cdot 2^{O(|\varphi|)}$ states. We need to check this automaton for emptiness. Using Proposition 13, we get the following results.

Theorem 25. [LP85, SC85, VW86]

1. *The program complexity of the verification problem is complete for NLOGSPACE.*
2. *The specification complexity of the verification problem is complete for PSPACE.*
3. *Checking whether a finite-state program P satisfies an LTL formula φ can be done in time $O(|P| \cdot 2^{O(|\varphi|)})$ or in space $O((|\varphi| + \log |P|)^2)$.*

We note that a time upper bound that is polynomial in the size of the program and exponential in the size of the specification is considered here to be reasonable, since the specification is usually rather short [LP85]. For a practical verification algorithm that is based on the automata-theoretic approach see [CVWY92].

4.3 Synthesis

In the previous section we dealt with *verification*: we are given a finite-state program and an LTL specification and we have to verify that the program meets the specification. A frequent criticism against this approach, however, is that verification is done *after* significant resources have already been invested in the development of the program. Since programs invariably contain errors, verification simply becomes part of the debugging process. The critics argue that the desired goal is to use the specification in the program development process in order to guarantee the design of correct programs. This is called *program synthesis*. It turns out that to solve the program-synthesis problem we need to use automata on infinite trees.

Rabin Tree Automata Rabin tree automata run on infinite labeled trees with a uniform branching degree (recall the definition of labeled trees in Section 2.5). The (infinite) *k-ary tree* τ_k is the set $\{1, \ldots, k\}^*$, i.e., the set of all finite sequences over $\{1, \ldots, k\}$. The elements of τ_k are the nodes of the tree. If x and xi are nodes of τ_k, then there is an edge from x to xi, i.e., x is the parent of xi and xi is the child of x. The empty sequence ε is the root of τ_k. A branch $\beta = x_0, x_1, \ldots$ of τ_k is an infinite sequence of nodes such that $x_0 = \varepsilon$, and x_i is the parent of x_{i+1} for all $i \geq 0$. A *Σ-labeled k-ary tree $\mathcal{T}$*, for a finite alphabet Σ, is a mapping $\mathcal{T} : \tau_k \to \Sigma$ that assigns to every node a label. We often refer to *labeled trees* as *trees*; the intention will be clear from the context. A branch $\beta = x_0, x_1, \ldots$ of $\mathcal{T}$ defines an infinite word $\mathcal{T}(\beta) = \mathcal{T}(x_0), \mathcal{T}(x_1), \ldots$ consisting of the sequence of labels along the branch.

A k-ary *Rabin tree automaton* A is a tuple $(\Sigma, S, S^0, \rho, G)$, where Σ is a finite alphabet, S is a finite set of states, $S^0 \subseteq S$ is a set of initial states, $G \subseteq 2^S \times 2^S$ is a Rabin condition, and $\rho : S \times \Sigma \to 2^{S^k}$ is a transition function. The automaton A takes as input Σ-labeled k-ary trees. Note that $\rho(s, a)$ is a set of k-tuples for each state s and symbol a. Intuitively, when the automaton is in state s and it is reading a node x, it nondeterministically chooses a k-tuple $\langle s_1, \ldots, s_k \rangle$ in $\rho(s, \mathcal{T}(x))$ and then makes k copies of itself and moves to the node xi in the state s_i for $i = 1, \ldots, k$. A *run* $r : \tau_k \to S$ of A on a Σ-labeled k-ary tree $\mathcal{T}$ is an S-labeled k-ary tree such that the root is labeled by an initial state and the transitions obey the transition function ρ; that is, $r(\varepsilon) \in S^0$, and for each node x we have $\langle r(x1), \ldots, r(xk) \rangle \in \rho(r(x), \mathcal{T}(x))$. The run is *accepting* if $r(\beta)$ satisfies G for every branch $\beta = x_0, x_1, \ldots$ of τ_k. That is, for every branch $\beta = x_0, x_1, \ldots$, there is some pair $(L, U) \in G$ such that $r(x_i) \in L$ for infinitely many i's, but $r(x_i) \in U$ for only finitely many i's. Note that different branches might be satisfied by different pairs in G. The *language* of A, denoted $L_\omega(A)$, is the set of trees accepted by A. It is easy to see that Rabin automata on infinite words are essentially 1-ary Rabin tree automata.

The *nonemptiness problem* for Rabin tree automata is to decide, given a Rabin tree automaton A, whether $L_\omega(A)$ is nonempty. Unlike the nonemptiness problem for automata on finite and infinite words, the nonemptiness problem for tree automata is highly nontrivial. It was shown to be decidable in [Rab69], but the algorithm there had *nonelementary* time complexity; i.e., its time complexity could not be bounded by any fixed stack of exponential functions. Later on, elementary algorithms were described in [HR72, Rab72]. The algorithm in [HR72] runs in doubly exponential time and the algorithm in [Rab72] runs in exponential time. Several years later, in [Eme85, VS85], it was shown that the nonemptiness problem for Rabin tree automata is in NP. Finally, in [EJ88], it was shown that the problem is NP-complete.

There are two relevant size parameters for Rabin tree automata. The first is the *transition size*, which is size of the transition function (i.e., the sum of the sizes of the sets $|\rho(s, a)|$ for $s \in S$ and $a \in \Sigma$); the transition size clearly takes into account the the number of states in S. The second is the number of pairs in the acceptance condition G. For our application here we need a complexity analysis of the nonemptiness problem that takes into account separately the two parameters.

Proposition 26. [EJ88, PR89] *For Rabin tree automata with transition size m and n pairs, the nonemptiness problem can be solved in time $(mn)^{O(n)}$.*

In other words, the nonemptiness problem for Rabin tree automata can be solved in time that is exponential in the number of pairs but polynomial in the transition size. As we will see, this distinction is quite significant.

Realizability The classical approach to program synthesis is to extract a program from a proof that the specification is satisfiable. In [EC82, MW84], it is shown how to extract programs from (finite representations of) models of the specification. In the late 1980s, several researchers realized that the classical approach is well suited to *closed* systems, but not to *open* systems [Dil89, PR89, ALW89]. In open systems the program interacts with the environment; such programs are called *reactive* programs [HP85]. A correct reactive program should be able to handle arbitrary actions of the environment. If one applies the techniques of [EC82, MW84] to reactive programs, one obtains programs that can handle only certain actions of the environment. In [PR89, ALW89, Dil89], it is argued that the right way to approach synthesis of reactive programs is to consider the situation as an *infinite game* between the environment and the program.

We are given a finite set W of states and a valuation $V : W \rightarrow 2^{Prop}$. The intuition is that W describes all the *observable* states that the system can be in. (We will see later why the emphasis here on observability.) A *behavior* r over W is an infinite word over the alphabet W. The intended meaning is that the behavior $w_0, w_1, \ldots$ describes a sequence of states that the system goes through, where the transition from w_{i-1} to w_i was caused by the environment when i is odd and by the program when i is even. That is, the program makes the first move (into the first state), the environment responds with the second move, the program counters with the third move, and so on. We associate with r the *computation* $V(r) = V(w_0), V(w_1), \ldots$, and say that r satisfies an LTL formula φ if $V(r)$ satisfies φ. The goal of the program is to satisfy the specification φ in the face of every possible move by the environment. The program has no control over the environment moves; it only controls its own moves. Thus, the situation can be viewed as an infinite game between the environment and the program, where the goal of the program is to satisfy the specification φ. Infinite games were introduced in [GS53] and they are of fundamental importance in descriptive set theory [Mos80].

Histories are finite words in W^*. The history of a run $r = w_0, w_1, \ldots$ at the even point $i \geq 0$, denoted $hist(r, i)$, is the finite word $w_1, w_3, \ldots, w_{i-1}$ consisting of all states moved to by the environment; the history is the empty sequence ε for $i = 0$. A *program* is a function $f : W^* \rightarrow W$ from histories to states. The idea is that if the program is scheduled at a point at which the history is h, then the program will cause a change into the state $f(h)$. This captures the intuition that the program acts in reaction to the environment's actions. A behavior r over W is a *run* of the program f if $s_i = f(hist(r, i))$ for all even i. That is, all the state transitions caused by the program are consistent with the program f. A program f *satisfies* the specification φ if every run of f over W satisfies φ. Thus, a correct program can be then viewed as a winning strategy in the game against the environment. We say that φ is *realizable* with respect to W and V if there is a program f that satisfies φ, in which case we say that f *realizes* φ. (In the sequel, we often omit explicit mention of W and V when it is clear from the context.) It turns out that satisfiability of φ is not sufficient to guarantee realizability of φ.

Example 3. Consider the case where $Prop = \{p\}$, $W = \{0, 1\}$, $V(0) = \emptyset$, and $V(1) = \{p\}$. Consider the formula Gp. This formula requires that p always be true, and it is clearly satisfiable. There is no way, however, for the program to enforce this requirement, since the environment can always moves to the state 0, making p false. Thus, Gp is not realizable. On the other hand, the formula GFp, which requires p to hold infinitely often, is realizable; in fact, it is realized by the simple program that maps every history to the state 1. This shows that realizability is a stronger requirement than satisfiability. ▮

Consider now the specification φ. By Corollary 23, we can build a Büchi automaton $A_\varphi = (\Sigma, S, S^0, \rho, F)$, where $\Sigma = 2^{Prop}$ and $|S|$ is in $2^{O(|\varphi|)}$, such that $L_\omega(A_\varphi)$ is exactly the set of computations satisfying the formula φ. Thus, given a state set W and a valuation $V : W \rightarrow 2^{Prop}$, we can also construct a Büchi automaton $A'_\varphi = (W, S, S^0, \rho', F)$ such that $L_\omega(A'_\varphi)$ is exactly the set of behaviors satisfying the formula φ, by simply taking $\rho'(s, w) = \rho(s, V(w))$. It follows that we can assume without loss of generality that the winning condition for the game between the environment and the program is expressed by a Büchi automaton A: the program f *wins* the game if every run of f is accepted by A. We thus say that the program f realizes a Büchi automaton A if all its runs are accepted by A. We also say then that A is realizable.

It turns out that the realizability problem for Büchi automata is essentially the *solvability problem* described in [Chu63]. (The winning condition in [Chu63] is expressed in S1S, the monadic second-order theory of one successor function, but it is known [Büc62] that S1S sentences can be translated to Büchi automata.) The solvability problem was studied in [BL69, Rab72]. It is shown in [Rab72] that this problem can be solved by using Rabin tree automata.

Consider a program $f : W^* \rightarrow W$. Suppose without loss of generality that $W = \{1, \ldots, k\}$, for some $k > 0$. The program f can be represented by a W-labeled k-ary tree $\mathcal{T}_f$. Consider a node $x = i_0 i_1 \ldots i_m$, where $1 \leq i_j \leq k$ for $j = 0, \ldots, m$. We note that x is a history in W^*, and define $\mathcal{T}_f(x) = f(x)$. Conversely, a W-labeled k-ary tree $\mathcal{T}$ defines a program $f_{\mathcal{T}}$. Consider a history $h = i_0 i_1 \ldots i_m$, where $1 \leq i_j \leq k$ for $j = 0, \ldots, m$. We note that h is a node of τ_k, and define $f_{\mathcal{T}}(h) = \mathcal{T}(h)$. Thus, W-labeled k-ary trees can be viewed as programs.

It is not hard to see that the runs of f correspond to the branches of $\mathcal{T}_f$. Let $\beta = x_0, x_1, \ldots$ be a branch, where $x_0 = \varepsilon$ and $x_j = x_{j-1} i_{j-1}$ for $j > 0$. Then $r = \mathcal{T}(x_0), i_0, \mathcal{T}(x_1), i_1, \mathcal{T}(x_2), \ldots$ is a run of f, denoted $r(\beta)$. Conversely, if $r = i_0, i_1, \ldots$ is a run of f, then $\mathcal{T}_f$ contains a branch $\beta(r) = x_0, x_1, \ldots$, where $x_0 = \varepsilon$, $x_j = x_{j-1} i_{2j+1}$, and $\mathcal{T}(x_j) = i_{2j}$ for $j \geq 0$. One way to visualize this is to think of the edge from the parent x to its child xi as labeled by i. Thus, the run $r(\beta)$ is the sequence of edge and node labels along β.

We thus refer to the behaviors $r(\beta)$ for branches β of a W-labeled k-ary tree $\mathcal{T}$ as the *runs of* $\mathcal{T}$, and we say that $\mathcal{T}$ *realizes* a Büchi automaton A if all the runs of $\mathcal{T}$ are accepted by A. We have thus obtained the following:

Proposition 27. *A program f realizes a Büchi automaton A iff the tree $\mathcal{T}_f$ realizes A.*

We have thus reduced the realizability problem for LTL specifications to an automata-theoretic problem: given a Büchi automaton A, decide if there is a tree $\mathcal{T}$ that realizes A. Our next step is to reduce this problem to the nonemptiness problem for Rabin tree automata. We will construct a Rabin automaton B that accepts precisely the trees that realize A. Thus, $L_\omega(B) \neq \emptyset$ iff there is a tree that realizes A.

Theorem 28. *Given a Büchi automaton A with n states over an alphabet $W = \{1, \ldots, k\}$, we can construct a k-ary Rabin tree automaton B with transition size $k 2^{O(n \log n)}$ and $O(n)$ pairs such that $L_\omega(B)$ is precisely the set of trees that realize A.*

Proof: Consider an input tree $\mathcal{T}$. The Rabin tree automaton B needs to verify that for every branch β of $\mathcal{T}$ we have that $r(\beta) \in L_\omega(A)$. Thus, B needs to "run A in parallel" on all branches of $\mathcal{T}$. We first need to deal with the fact that the labels in $\mathcal{T}$ contain information only about the actions of f (while the information on the actions of the environment is implicit in the edges). Suppose that $A = (W, S, S^0, \rho, F)$. We first define a Büchi automaton A' that emulates A by reading pairs of input symbols at a time. Let $A' = (W^2, S \times \{0, 1\}, S^0 \times \{0\}, \rho', S \times \{1\})$, where

$$\begin{aligned} \rho'(s, \langle a, b \rangle) = \{ & \langle t, 0 \rangle \mid t \in \rho(s', b) - F \text{ for some } s' \in \rho(s, a) - F\} \cup \\ & \{\langle t, 1 \rangle \mid t \in \rho(s', b) \text{ for some } s' \in \rho(s, a) \cap F\} \cup \\ & \{\langle t, 1 \rangle \mid t \in \rho(s', b) \cap F \text{ for some } s' \in \rho(s, a)\}. \end{aligned}$$

Intuitively, ρ' applies two transitions of A while remembering whether either transition visited F. Note that this construction doubles the number of states. It is easy to prove the following claim:
Claim: A' accepts the infinite word $\langle w_0, w_1\rangle, \langle w_2, w_3\rangle, \ldots$ over the alphabet W^2 iff A accepts the infinite word $w_0, w_1, w_2, w_3, \ldots$ over W.

In order to be able to run A' in parallel on all branches, we apply Proposition 10 to A' and obtain a deterministic Rabin automaton A_d such that $L_\omega(A_d) = L_\omega(A')$. As commented in Section 2.2, A_d has $2^{O(n \log n)}$ states and $O(n)$ pairs. Let $A_d = (W^2, Q, \{q^0\}, \delta, G)$.

We can now construct a Rabin tree automaton that "runs A_d in parallel" on all branches of $\mathcal{T}$. Let $B = (W, Q, \{q^0\}, \delta', G)$, where δ' is defined as follows:

$$\delta'(q, a) = \delta(q, \langle a, 1\rangle) \times \cdots \delta(q, \langle a, k\rangle).$$

Intuitively, B emulates A_d by feeding it pairs consisting of a node label and an edge label. Note that if $\delta(q, \langle a, i\rangle) = \emptyset$ for some $1 \leq i \leq k$, then $\delta'(q, a) = \emptyset$.
Claim: $L_\omega(B)$ is precisely the set of trees that realize A.

It remains to analyze the size of A. It is clear that it has $2^{O(n \log n)}$ states and $O(n)$ pairs. Since it is deterministic, its transition size is $k2^{O(n \log n)}$. ∎

We can now apply Proposition 26 to solve the realizability problem.

Theorem 29. [ALW89, PR89] *The realizability problem for Büchi automata can be solved in exponential time.*

Proof: By Theorem 28, given a Büchi automaton A with n states over an alphabet $W = \{1, \ldots, k\}$, we can construct a k-ary Rabin tree automaton B with transition size $k2^{O(n \log n)}$ and and $O(n)$ pairs such that $L_\omega(B)$ is precisely the set of trees that realize A. By Proposition 26, we can test the nonemptiness of B in time $k^{O(n)}2^{O(n^2 \log n)}$. ∎

Corollary 30. [PR89] *The realizability problem for LTL can be solved in doubly exponential time.*

Proof: By Corollary 23, given an LTL formula φ, one can build a Büchi automaton A_φ with $2^{O(|\varphi|)}$ states such that $L_\omega(A_\varphi)$ is exactly the set of computations satisfying the formula φ. By combining this with the bound of Theorem 29, we get a time bound of $k^{2^{O(|\varphi|)}}$. ∎

In [PR89], it is shown that the doubly exponential time bound of Corollary 30 is essentially optimal. Thus, while the realizability problem for LTL is decidable, in the worst case it can be highly intractable.

Example 4. Consider again the situation where $Prop = \{p\}$, $W = \{0, 1\}$, $V(0) = \emptyset$, and $V(1) = \{p\}$. Let φ be the formula Gp. We have $A_\varphi = (W, \{1\}, \{1\}, \rho, W)$, where $\rho(1, 1) = \{1\}$, and all other transitions are empty (e.g., $\rho(1, 0) = \emptyset$, etc.). Note that A_φ is deterministic. We can emulate A_φ by an automaton that reads pairs of symbols: $A'_\varphi = (W^2, W \times \{0, 1\}, \{\langle 1, 0\rangle\}, \rho', W \times \{1\})$, where $\rho(\langle 1, 0\rangle, \langle 1, 1\rangle) = \{\langle 1, 1\rangle\}$, and all other transitions are empty. Finally, we construct the Rabin tree automaton

$B = (W, W \times \{0,1\}, \langle 1,0\rangle, \delta', (L,U))$, where $\delta'(s,a)$ is empty for all states s and symbol a. Clearly, $L_\omega(B) = \emptyset$, which implies that Gp is not realizable. ▮

We note that Corollary 30 only tells us how to decide whether an LTL formula is realizable or not. It is shown in [PR89], however, that the algorithm of Proposition 26 can provide more than just a "yes/no" answer. When the Rabin automaton B is nonempty, the algorithm returns a finite representation of an infinite tree accepted by B. It turns out that this representation can be converted into a program f that realizes the specification. It even turns out that this program is a *finite-state* program. This means that there are a finite set N, a function $g : W^* \rightarrow N$, a function $\alpha_1 : N \times W \rightarrow N$, and a function $\alpha_2 : N \rightarrow W$ such that for all $h \in W^*$ and $w \in W$ we have:

– $f(h) = \alpha_2(g(h))$
– $g(hw) = \alpha_1(g(h), w)$

Thus, instead of remembering the history h (which requires an unbounded memory), the program needs only to remember $g(h)$. It performs its action $\alpha_1(h)$ and, when it sees the environment's action w, it updates its memory to $\alpha_2(g(h), w)$. Note that this "memory" is *internal* to the program and is not pertinent to the specification. This is in contrast to the observable states in W that are pertinent to the specification.

Acknowledgements

I am grateful to Oystein Haugen, Orna Kupferman, and Faron Moller for their many comments on earlier drafts of this paper.

References

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. In *Proc. 16th Int. Colloquium on Automata, Languages and Programming*, volume 372, pages 1–17. Lecture Notes in Computer Science, Springer-Verlag, July 1989.

[BL69] J.R. Büchi and L.HG. Landweber. Solving sequential conditions by finite-state strategies. *Trans. AMS*, 138:295–311, 1969.

[BL80] J.A. Brzozowski and E. Leiss. Finite automata, and sequential networks. *Theoretical Computer Science*, 10:19–35, 1980.

[Büc62] J.R. Büchi. On a decision method in restricted second order arithmetic. In *Proc. Internat. Congr. Logic, Method and Philos. Sci. 1960*, pages 1–12, Stanford, 1962. Stanford University Press.

[Cho74] Y. Choueka. Theories of automata on ω-tapes: A simplified approach. *J. Computer and System Sciences*, 8:117–141, 1974.

[Chu63] A. Church. Logic, arithmetics, and automata. In *Proc. International Congress of Mathematicians, 1962*, pages 23–35. institut Mittag-Leffler, 1963.

[CKS81] A.K. Chandra, D.C. Kozen, and L.J. Stockmeyer. Alternation. *Journal of the Association for Computing Machinery*, 28(1):114–133, 1981.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[CVWY92] C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[Dil89] D.L. Dill. *Trace theory for automatic hierarchical verification of speed independent circuits*. MIT Press, 1989.

[EC82] E.A. Emerson and E.M. Clarke. Using branching time logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[EH86] 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.

[EJ88] E.A. Emerson and C. Jutla. The complexity of tree automata and logics of programs. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 328–337, White Plains, October 1988.

[EJ89] E.A. Emerson and C. Jutla. On simultaneously determinizing and complementing ω-automata. In *Proceedings of the 4th IEEE Symposium on Logic in Computer Science*, pages 333–342, 1989.

[EL85a] E.A. Emerson and C.-L. Lei. Modalities for model checking: Branching time logic strikes back. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 84–96, New Orleans, January 1985.

[EL85b] E.A. Emerson and C.-L. Lei. Temporal model checking under generalized fairness constraints. In *Proc. 18th Hawaii International Conference on System Sciences*, pages 277–288, Hawaii, 1985.

[Eme85] E.A. Emerson. Automata, tableaux, and temporal logics. In *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 79–87. Springer-Verlag, Berlin, 1985.

[GJ79] M. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. Freeman and Co., San Francisco, 1979.

[GPSS80] D. Gabbay, A. Pnueli, S. Shelah, and J. Stavi. On the temporal analysis of fairness. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 163–173, January 1980.

[GS53] D. Gale and F. M. Stewart. Infinite games of perfect information. *Ann. Math. Studies*, 28:245–266, 1953.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In K. Apt, editor, *Logics and Models of Concurrent Systems*, volume F-13 of *NATO Advanced Summer Institutes*, pages 477–498. Springer-Verlag, 1985.

[HR72] R. Hossley and C.W. Rackoff. The emptiness problem for automata on infinite trees. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 121–124, 1972.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, New York, 1979.

[Jon75] N.D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11:68–75, 1975.

[Kur94] Robert P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, Princeton, New Jersey, 1994.

[Lam80] L. Lamport. Sometimes is sometimes "not never" - on the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, January 1980.

[Lei81] Leiss. Succinct representation of regular languages by boolean automata. *Theoretical Computer Science*, 13:323–330, 1981.

[Liu89] M.T. Liu. Protocol engineering. *Advances in Computing*, 29:79–195, 1989.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[LPZ85] O. Lichtenstein, A. Pnueli, and L. Zuck. The glory of the past. In *Logics of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 196–218, Brooklyn, 1985. Springer-Verlag, Berlin.

[McN66] R. McNaughton. Testing and generating infinite sequences by a finite automaton. *Information and Control*, 9:521–530, 1966.

[MF71] A.R. Meyer and M.J. Fischer. Economy of description by automata, grammars, and formal systems. In *Proc. 12th IEEE Symp. on Switching and Automata Theory*, pages 188–191, 1971.

[MH84] S. Miyano and T. Hayashi. Alternating finite automata on ω-words. *Theoretical Computer Science*, 32:321–330, 1984.

[Mic88] M. Michel. Complementation is more difficult with automata on infinite words. CNET, Paris, 1988.

[Mos80] Y.N. Moschovakis. *Descriptive Set Theory*. North Holland, 1980.

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

[MS72] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential time. In *Proc. 13th IEEE Symp. on Switching and Automata Theory*, pages 125–129, 1972.

[MS87] D.E. Muller and P.E. Schupp. Alternating automata on infinite trees. *Theoretical Computer Science*, 54,:267–276, 1987.

[MSS88] D. E. Muller, A. Saoudi, and P. E. Schupp. Weak alternating automata give a simple explanation of why most temporal and dynamic logics are decidable in exponential time. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science*, pages 422–427, Edinburgh, July 1988.

[MW84] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 6(1):68–93, January 1984.

[OL82] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4(3):455–495, July 1982.

[Pei85] R. Peikert. ω-regular languages and propositional temporal logic. Technical Report 85-01, ETH, 1985.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symposium on Foundation of Computer Science*, pages 46–57, 1977.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the Sixteenth ACM Symposium on Principles of Programming Languages*, Austin, Januery 1989.

[Rab69] M.O. Rabin. Decidability of second order theories and automata on infinite trees. *Transaction of the AMS*, 141:1–35, 1969.

[Rab72] M.O. Rabin. Automata on infinite objects and Church's problem. In *Regional Conf. Ser. Math., 13*, Providence, Rhode Island, 1972. AMS.

[RS59] M.O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. of Research and Development*, 3:115–125, 1959.

[Rud87] H. Rudin. Network protocols and tools to help produce them. *Annual Review of Computer Science*, 2:291–316, 1987.

[Saf88] S. Safra. On the complexity of omega-automata. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, pages 319–327, White Plains, October 1988.

[Sav70] W.J. Savitch. Relationship between nondeterministic and deterministic tape complexities. *J. on Computer and System Sciences*, 4:177–192, 1970.

[SC85] A.P. Sistla and E.M. Clarke. The complexity of propositional linear temporal logic. *Journal of the Association for Computing Machinery*, 32:733–749, 1985.

[Sis83] A.P. Sistla. *Theoretical issues in the design and analysis of distributed systems*. PhD thesis, Harvard University, 1983.

[SPH84] R. Sherman, A. Pnueli, and D. Harel. Is the interesting part of process logic uninteresting: a translation from PL to PDL. *SIAM J. on Computing*, 13(4):825–839, 1984.

[SVW87] A.P. Sistla, M.Y. Vardi, and P. Wolper. The complementation problem for Büchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

[Tho90] W. Thomas. Automata on infinite objects. *Handbook of theoretical computer science*, pages 165–191, 1990.

[Var94] M.Y. Vardi. Nontraditional applications of automata theory. In *Theoretical Aspects of Computer Software, Proc. Int. Symposium (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 575–597. Springer-Verlag, Berlin, 1994.

[VS85] M.Y. Vardi and L. Stockmeyer. Improved upper and lower bounds for modal logics of programs. In *Proc 17th ACM Symp. on Theory of Computing*, pages 240–251, 1985.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

[VW94] M.Y. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, 1994.

[Wol82] P. Wolper. *Synthesis of Communicating Processes from Temporal Logic Specifications*. PhD thesis, Stanford University, 1982.

[WVS83] P. Wolper, M.Y. Vardi, and A.P. Sistla. Reasoning about infinite computation paths. In *Proc. 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, Tucson, 1983.