Synthesis of Succinct Systems

John Fearnley[1], Doron Peled[2], and Sven Schewe[1]

[1] Department of Computer Science, University of Liverpool, Liverpool, UK
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. Synthesis of correct by design systems from specification has recently attracted much attention. The theoretical results imply that this problem is highly intractable, e.g., synthesizing a system is 2EXPTIME-complete for an LTL specification, and EXPTIME-complete for a CTL specification. However, an argument against it is that the temporal specification is highly compact, and the complexity reflects the large size of the system constructed. In that respect, the complexity should, perhaps, be specified relative to the size of the minimal satisfying system. A careful observation reveals that the size of the system is presented in such arguments as the size of its state space. This view is a bit nonstandard, in the sense that the state space can be exponentially larger than the size of a reasonable implementation such as a circuit or a program. Although this alternative measure of the size of the synthesized system is more intuitive (e.g., this is the standard way model checking problems are measured), research on synthesis has so far stayed with measuring the system in terms of the explicit state space. This raises the question of whether or not there always exists a small system. In this paper, we show that this is the case if, and only if, PSPACE = EXPTIME.

1 Introduction

While automatic verification [4, 1] has gained many algorithmic solutions and various successful tools, automatic synthesis of correct-by-design [2, 3, 11] systems has had fewer results that have produced widely used tools. One of the main problems is the complexity of the synthesis problem. A classical result by Pnueli and Rosner [12] shows that synthesis of a system from an LTL specification is in 2EXPTIME-complete. It was later shown by Kupferman and Vardi that synthesis for CTL specifications is EXPTIME-complete [7]. A counter argument for this complexity difficulty is that the size of the system produced by the synthesis procedure is typically large. Some concrete examples [6] show that the size of the system synthesized may be doubly exponentially larger than the LTL specification. This, in fact, shows that LTL specification is quite a compact representation of a system, rather than simply a formalism that is intrinsically hard for synthesis.

As we are interested in the relationship between the specification and synthesized system, a question arises with respect to the nature of the system representation. The classical synthesis problem regards the system as a transition system with an explicit state space, and the size of this system is the number of transitions and states. This is, to some extent, a biased measurement, as systems (programs, circuits with memory) often have a much more concise representation: it is often possible to produce a circuit or program that is exponentially smaller than the corresponding transition system. For example, it is easy to produce a small program that implements an n bit binary counter, but a corresponding transition system requires 2^n distinct states to implement the same counter. Thus, we ask the question of *what is the size of the minimal system representation in terms of the specification?*

We look at CTL and LTL, and study the relative synthesized system complexity. We focus on CTL synthesis and then show results for LTL. We choose to represent our systems as online Turing machines with a bounded storage tape. This is because there exists straightforward translations between online Turing machines, and the natural representations of a system, such as programs and circuits. Moreover, these translations produce programs and circuits that have comparable size to the original online Turing machine.

Our binary-counter example showed that there are instances in which an online Turing machine model of a CTL formula is exponentially smaller than a transition system model of that formula. In this paper we ask: is this always the case? More precisely, for every CTL formula ϕ, does there always exist an online Turing machine $\mathcal{M}$ that models ϕ, where the amount of space required to describe $\mathcal{M}$ is polynomial in ϕ? We call machines with this property *small*. Our answer to this problem is the following theorem:

> Every CTL formula has small online Turing machine model if, and only if, PSPACE = EXPTIME.

Since it is widely believed that PSPACE $\neq$ EXPTIME, the "if" direction of this theorem implies that it is very unlikely that all CTL formulas have small online Turing machine models. On the other hand, since proving that PSPACE $\neq$ EXPTIME is very difficult, the "only if" direction of this theorem implies that it is also very difficult to find a family of CTL formulas that provably require super-polynomial sized models.

Since the online Turing machine has a storage tape, after receiving an input it may perform many intermediate computational steps to produce the corresponding output. In particular, if ϕ is a CTL formula, then a small model of ϕ may take exponential time in ϕ to produce each output. This leads to the second question that we address in this paper: for each CTL formula ϕ, does there always exist a small online Turing machine that is *fast*? The machine is fast if it always responds to each input in polynomial time. Again, our result is to link this question to an open problem in complexity theory:

> Every CTL formula has small online Turing machine model that is fast if, and only if, EXPTIME $\subseteq$ P/poly.

P/poly is the class of problems solvable by a polynomial-time Turing machine with an advice function that provides advice strings of polynomial size. It has been shown that if EXPTIME $\subseteq$ P/poly, then the polynomial hierarchy collapses [5]. Since our first result shows that it is already highly unlikely that there is always a small model, it is not too surprising that it is highly unlikely that there is always a small model that is fast. On the other hand, this result also says something stronger than our first result: we cannot even expect to find a family of CTL formulas that provably require either a super-polynomial sized model, or super-polynomial time to respond to an input.

2 Preliminaries

2.1 CTL Formulas

Given a finite set Π of atomic propositions, the syntax of a CTL formula is defined as follows:

$$\begin{aligned}\phi &::= p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi,\\ \psi &::= \mathcal{X}\phi \mid \phi\,\mathcal{U}\,\phi,\end{aligned}$$

where $p \in \Pi$. For each CTL formula ϕ we define $|\phi|$ to give the size of the parse tree for that formula.

Let $T = (V, E)$ be an infinite directed tree, with all edges pointing away from the root. Let $l : V \to 2^{\Pi}$ be a labelling function. The semantics of CTL are defined as follows. For each $v \in V$ we have:

- $v \models p$ if and only if $p \in l(v)$.
- $v \models \neg\phi$ if and only if $v \not\models \phi$.
- $v \models \phi \vee \psi$ if and only if either $v \models \phi$ or $v \models \psi$.
- $v \models A\psi$ if and only if for all paths π starting at v we have $\pi \models \psi$.
- $v \models E\psi$ if and only if there exists a path π starting at v with $\pi \models \psi$.

Let $\pi = v_1, v_2, \dots$ be an infinite path in T. We have:

- $\pi \models \mathcal{X}\psi$ if and only if $v_2 \models \psi$.
- $\pi \models \phi\,\mathcal{U}\,\psi$ if and only if there exists $i \in \mathbb{N}$ such that $v_i \models \psi$ and for all j in the range $1 \le j < i$ we have $v_j \models \phi$.

The pair (T, l), where T is a tree and l is a labelling function, is a model of ϕ if and only if $r \models \phi$, where $r \in V$ is the root of the tree. If (T, l) is a model of ϕ, then we write $T, l \models \phi$.

2.2 Abstract Transition Systems

An abstract transition system is a tuple $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l, \mathsf{start}, \mathsf{input})$. The set S is a finite set of states, and the state $\mathsf{start} \in S$ is a starting state. The set Σ_I gives an input alphabet, and the set Σ_O gives an output alphabet. The letter $\mathsf{input} \in \Sigma_I$ gives the initial input letter. The function $l : S \to \Sigma_I \times \Sigma_O$ maps each state to a pair of input and output letters. The transition function

$\tau : S \times \Sigma_I \to S$ gives the transitions for each state and each input letter. Let Π_I be a set of input variables, and let Π_O be a set of output variables. We are interested in transition systems in which the input alphabet is $\Sigma_I = 2^{\Pi_I}$, and the output alphabet is $\Sigma_O = 2^{\Pi_O}$.

We say that a transition system is *input preserving* if the labels accurately record the previous input. More formally, a transition system is input preserving if for every state $s \in S$, and every input letter σ_I, we have $l(\tau(s, \sigma_I)) = (\sigma_I, \sigma_O)$ for some output letter $\sigma_O \in \Sigma_I$. Moreover, we must have that $l(\mathsf{start}) = (\mathsf{input}, \sigma_O)$ for some letter $\sigma_O \in \Sigma_O$.

A sequence of states $\pi = s_1, s_2, s_3, \dots$ is an infinite path in the transition system if $s_1 = \mathsf{start}$, and if for each i there is a letter σ_I such that $\tau(s_i, \sigma_I) = s_{i+1}$. For each infinite path π, we define a word $\sigma(\pi) = l(s_1), l(s_2), l(s_3), \dots$, which gives the labels of the states seen along π.

Suppose that ϕ is a CTL formula that uses Π_I as a set *input* propositions, and Π_O as a set of *output* propositions. Let $\mathcal{T} = (S, 2^{\Pi_I}, 2^{\Pi_O}, \tau, l, \mathsf{start}, \mathsf{input})$ be an abstract transition system that uses sets of these propositions as input and output alphabets. Furthermore, let (T, l) be the infinite tree corresponding to the set of words $\sigma(\pi)$, over all infinite paths π. We say that $\mathcal{T}$ is a model of ϕ if $T, l \models \phi$. Given a CTL formula ϕ and an abstract transition system $\mathcal{T}$, the CTL model checking problem is to decide whether $\mathcal{T}$ is a model of ϕ.

Theorem 1 ([9]). *Given an abstract transition system $\mathcal{T}$ and an CTL-formula ϕ, the CTL model checking problem can be solved in space polynomial in $|\phi| \cdot \log |\mathcal{T}|$.*

Given a CTL formula ϕ, the CTL synthesis problem is to decide whether there exists an abstract transition system that is a model of ϕ. This problem is known to be EXPTIME-complete.

Theorem 2 ([7]). *The CTL synthesis problem is EXPTIME-complete.*

2.3 Tree Automata

Universal Co-Büchi tree automata will play a fundamental role in the proofs given in subsequent sections, because we will translate each CTL formula ϕ into a universal Co-Büchi tree automaton $\mathcal{U}(\phi)$. The automaton will accept transition systems, and the language of the tree automaton will be exactly the set of models accepted by ϕ. We will then use these automata to obtain our main results.

A *universal Co-Büchi tree automaton* is $\mathcal{A} = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$, where S denotes a finite set of states, Σ_I is a finite input alphabet, Σ_O is a finite output alphabet, $\mathsf{start} \in S$ is an initial state, δ is a transition function, $F \subseteq S$ is a set of final states, and $\mathsf{input} \in \Sigma_I$ is an initial input letter. The transition function $\delta : S \times (\Sigma_I \times \Sigma_O) \to 2^{S \times \Sigma_I}$ maps a state and an output letter to a set of pairs, where each pair consists of a successor states and an input letter.

The automaton runs on abstract transition systems that use Σ_I and Σ_O as their input and output alphabets. The acceptance mechanism is defined in terms of run graphs. A *run graph* of a universal Co-Büchi tree automaton $\mathcal{A} = (S_\mathcal{A}, \Sigma_I, \Sigma_O, \mathsf{start}_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A}, \mathsf{input})$ on an abstract Transition system $\mathcal{T} = (S_\mathcal{T}, \Sigma_I, \Sigma_O, \tau, l_\mathcal{T}, \mathsf{start}_\mathcal{T}, \mathsf{input})$ is defined to be a minimal directed graph $G = (V, E)$ that satisfies the following constraints:

- The vertices of G satisfy $V \subseteq S_\mathcal{A} \times S_\mathcal{T}$.
- The pair of initial states $(\mathsf{start}_\mathcal{A}, \mathsf{start}_\mathcal{T})$ is contained in V.
- Suppose that for a vertex $(q, t) \in V$, we have that $(q', \sigma_I) \in \delta(q, l_\mathcal{T}(t))$. An edge from (q, t) to $(q', \tau(t, \sigma_I))$ must be contained in E.

A run graph is *accepting* if every infinite path $v_1, v_2, v_3, \dots \in V^\omega$ contains only finitely many states in F. A transition system $\mathcal{T}$ is accepted by $\mathcal{A}$ if it has an accepting run graph. The set of transition systems accepted by $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. The automaton is empty if, and only if, its language is empty.

A universal Co-Büchi tree automaton is called a *safety* tree automaton if $F = \emptyset$. Therefore, for safety automata, we have that every run graph is accepting, and we drop the $F = \emptyset$ from the tuple defining the automaton. A universal Co-Büchi tree automaton is *deterministic* if $|\delta(s, (\sigma_I, \sigma_O))| = 1$, for all states s, input letters σ_I, and output letters σ_O.

2.4 Online Turing Machines

In this paper, we use *online* Turing machines as a formalisation of a reasonable implementation. An online Turing machine has three tapes: an infinite input tape, an infinite output tape, and a storage tape of bounded size. The input tape is read only, and the output tape is write only. Each time that a symbol is read from the input tape, the machine may spend time performing computation on the storage tape, before eventually writing a symbol to the output tape. A formal definition of this model can be found in Appendix A.

We can now define the synthesis problem for online Turing machines. Let ϕ be a CTL formula defined using Π_I and Π_O, as the sets of input, and output, propositions, respectively. We consider online Turing machines that use 2^{Π_I} as the input tape alphabet, and 2^{Π_O} as the output alphabet. Online Turing machines are required, after receiving an input symbol, to produce an output before the next input symbol can be read. Therefore, if we consider the set of all possible input words that could be placed on the input tape, then the set of possible outputs made by the online Turing machine forms a tree. If this tree is a model of ϕ, then we say that the online Turing machine is a model of ϕ.

Given a CTL formula ϕ, we say that an online Turing machine is a *small* model of ϕ if the machine can be described in space that is polynomial in ϕ. Note that a small model may still have a polynomially sized work tape, and therefore it may take an exponential number of steps to produce an output for a given input. We say that an online Turing machine is a *fast* model of ϕ if, for all inputs, it always takes a polynomial number of steps to produce an output.

3 Small Models Imply PSPACE = EXPTIME

Let ϕ be a CTL formula that has an input preserving model. In this section we show that, if there is always a small online Turing machine that models ϕ, then PSPACE = EXPTIME. Our approach to showing this result will be to guess a polynomially sized Turing machine $\mathcal{M}$, and then to use model checking to verify whether $\mathcal{M}$ is a model of ϕ. Since our assumption guarantees that we only need to guess polynomially sized online Turing machines, this gives a NPSPACE = PSPACE algorithm for solving the CTL synthesis problem. Our proof then follows from the fact that CTL synthesis is EXPTIME-complete.

To begin, we show how model checking can be applied to an online Turing machine. To do this, we unravel the machine to an abstract transition system.

Lemma 3. *Given an online Turing machine $\mathcal{M}$, and a CTL formula ϕ, there is an abstract transition system $\mathcal{T}(\mathcal{M})$ such that $\mathcal{M}$ is a model of ϕ if and only if $\mathcal{T}(\mathcal{M})$ is a model of ϕ.*

Obviously, the size of $\mathcal{T}(\mathcal{M})$ will be exponential in the size of $\mathcal{M}$. However, this is not a problem because there exists a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using only $O(|\mathcal{M}|)$ space.

Lemma 4. *There is a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using $O(|\mathcal{M}|)$ space.*

Since the model checking procedure given in Theorem 1 uses poly-logarithmic space, when it is applied to $\mathcal{T}(\mathcal{M})$, it will space polynomial in $|\mathcal{M}|$. Now, using standard techniques to compose space bounded Turing machines (see [10, Proposition 8.2], for example), we can compose the deterministic Turing machine given by Lemma 4 with the model checking procedure given in Theorem 1 to produce a deterministic Turing machine that uses polynomial space in $|\mathcal{M}|$. Hence, we have shown that each online Turing machine $\mathcal{M}$ can be model checked against ϕ in space polynomial in $|\mathcal{M}|$. This implies the following theorem.

Theorem 5. *Let ϕ be a satisfiable CTL formula. If there always exists an online Turing machine $\mathcal{M}$ that models ϕ, where $|\mathcal{M}|$ is polynomial in ϕ, then PSPACE = EXPTIME.*

4 PSPACE = EXPTIME Implies Small Models

In this section we show the opposite direction of the result given in Section 3. We show that if PSPACE = EXPTIME, then for every CTL formula ϕ that has an input preserving model, there exists a polynomially sized online Turing machine that is a model of ϕ. We start our proof of this result with a translation from CTL to universal Co-Büchi tree automata. In [7] it was shown that every CTL ϕ formula can be translated to an alternating Co-Büchi tree automaton whose language is the models of ϕ. It is relatively straightforward to translate this alternating tree automaton into a universal tree automaton.

One complication is that the input and output languages of the universal tree automaton are supersets of the propositions used to define ϕ. Let $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l_{\mathcal{T}}, \mathsf{start}, \mathsf{input})$ be an abstract transition system, where each $\sigma_I \in \Sigma_I$ contains some element $a \in 2^{\Pi_I}$ with $a \subseteq \sigma_I$, and each $\sigma_O \in \Sigma_O$ contains some element $a \in 2^{\Pi_O}$ with $a \subseteq \sigma_O$. We define $\mathcal{T} \restriction (\Pi_I, \Pi_O)$ to be the abstract transition system $\mathcal{T}' = (S, \Sigma_I, \Sigma_O, \tau, l_{\mathcal{T}'}, \mathsf{start}, \mathsf{input})$ where, if $l_{\mathcal{T}}(s) = (\sigma_I, \sigma_O)$, then we define $l_{\mathcal{T}'}(s) = (\sigma_I \cap 2^{\Pi_I}, \sigma_O \cap 2^{\Pi_O})$ for all $s \in S$. We have the following:

Lemma 6. *Let ϕ be a CTL formula, which is defined over the set 2^{Π_I} of input propositions, and the set 2^{Π_O} of output propositions. We can construct a universal Co-Büchi tree automaton $\mathcal{U}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$ such that:*

- *There is a model $\mathcal{T} \in \mathcal{L}(\mathcal{U}(\phi))$ if and only if $\mathcal{T} \restriction 2^{\Pi_O}$ is a model of ϕ.*
- *The size of the set S is polynomial in $|\phi|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|\phi|$.*
- *The transition function δ can be computed in time polynomial in $|\phi|$.*
- *The state start can be computed in polynomial time.*

The techniques used in [13] show how the automaton given by Lemma 6 can be translated into an equivalent deterministic safety tree automaton $\mathcal{F}(\phi)$. We use a slight modification of this reduction to ensure that $\mathcal{F}(\phi)$ accepts only the input preserving models of ϕ. The automaton has the following properties:

Lemma 7 ([13]). *Given the universal Co-Büchi tree automaton $\mathcal{U}(\phi)$, whose state space is $S_{\mathcal{U}}$, we can construct a deterministic safety tree automaton $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, \mathsf{input})$ such that:*

- *If $\mathcal{L}(\mathcal{U}(\phi))$ contains an input preserving abstract transition system, then $\mathcal{L}(\mathcal{F}(\phi))$ is not empty. Moreover, if $\mathcal{T}$ is in $\mathcal{L}(\mathcal{F}(\phi))$, then $\mathcal{T}$ is a model of ϕ.*
- *Each state in S can be stored in space polynomial in $|S_{\mathcal{U}}|$.*
- *The transition function δ can be computed in time polynomial in $|S_{\mathcal{U}}|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|S_{\mathcal{U}}|$.*
- *The state start can be computed in time polynomial in $|S_{\mathcal{U}}|$.*

We will use the safety automaton $\mathcal{F}(\phi)$ given by Lemma 7 to construct a polynomially sized model of ϕ. This may seem counter intuitive, because the number of states in $\mathcal{F}(\phi)$ may be exponential in ϕ. However, we do not need to build $\mathcal{F}(\phi)$. Instead our model will solve language emptiness problems for $\mathcal{F}(\phi)$.

For each state $s \in S$ in $\mathcal{F}(\phi)$, we define $\mathcal{F}^s(\phi)$ to be the automaton $\mathcal{F}(\phi)$ with starting state s. The *emptiness problem* takes a CTL formula ϕ and a state of $\mathcal{F}(\phi)$, and requires us to decide whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$. Note that the input has polynomial size in $|\phi|$. To solve this problem, we just construct $\mathcal{F}^s(\phi)$. Since $\mathcal{F}^s(\phi)$ can have at most exponentially many states in $|\phi|$, and the language emptiness problem for safety automata can be solved in polynomial time, we have that our emptiness problem lies in EXPTIME.

Lemma 8. *For every CTL formula ϕ, and every state s in $\mathcal{F}(\phi)$ we can decide whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$ in exponential time.*

Our key observation is that, under the assumption that PSPACE = EXPTIME, Lemma 8 implies that there must exist an algorithm for the emptiness problem that uses polynomial space. We will use this fact to construct $\mathcal{M}(\phi)$, which is a polynomially sized online Turing machine that models ϕ.

Let ϕ be a CTL formula that uses Π_I and Π_O as the set of input and output propositions, and suppose that ϕ has an input enabled model. Let $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F)$. The machine $\mathcal{M}(\phi)$ always maintains a current state $s \in S$. Lemma 7 implies that this can be stored in polynomial space. The machine $\mathcal{M}(\phi)$ begins by setting the current state $s = \mathsf{start}$. By Lemma 7 this can be done in polynomial time, and hence polynomial space.

Every time that $\mathcal{M}(\phi)$ reads a new input letter $\sigma_I \in 2^{\Pi_I}$ from the input tape, the following procedure is executed. The machine loops through each possible element of $\sigma_O \in \Sigma_O$ and checks whether there is a pair $(s', \sigma'_I) \in \delta(s, (\sigma''_I, \sigma_O))$ such that $\sigma'_I \cap 2^{\Pi_I} = \sigma_I$ (recall from Lemma 6 that each letter of Σ_I contains an element of 2^{Π_I} as a subset) and $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$. When an output symbol σ_O and state s' with this property are found, then the machine outputs $\sigma_O \cap 2^{\Pi_O}$, moves to the state s', and reads the next input letter.

The fact that a suitable pair σ_O and s' always exists can be proved by a simple inductive argument, which starts with the fact that $\mathcal{L}(\mathcal{F}^{\mathsf{start}}(\phi)) \neq \emptyset$, and uses the fact that we always pick a successor that satisfies the non-emptiness check. Moreover, it can be seen that $\mathcal{M}(\phi)$ is in fact simulating some abstract transition system $\mathcal{T} \restriction (\Pi_I, \Pi_O)$ where $\mathcal{T} \in \mathcal{L}(\mathcal{F}(\phi))$. Therefore, by Lemma 6, we have that $\mathcal{M}(\phi)$ is a model of ϕ.

The important part of our proof is that, if PSPACE = EXPTIME, then this procedure can be performed in polynomial space. Since each letter in Σ_O can be stored in polynomial space, we can iterate through all letters in Σ_O while using only polynomial space. By Lemma 8, the check $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$ can be performed in exponential time, and hence, using our assumption that PSPACE = EXPTIME, there must exist a polynomial space algorithm that performs this check. Therefore, we have constructed an online Turing machine that uses polynomial space and models ϕ. We have proved the following theorem.

Theorem 9. *Let ϕ be a CTL formula that has an input preserving model. If PSPACE = EXPTIME then there is an online Turing machine $\mathcal{M}$ that models ϕ, where $|\mathcal{M}|$ is polynomial in ϕ.*

Theorem 9 is not constructive. However, if a polynomially sized online Turing machine that models ϕ exists, then we can, always find it in PSPACE by guessing the machine, and then model checking it.

5 Small And Fast Models Imply EXPTIME $\subseteq$ P/poly

In this section we show that, if all satisfiable CTL formulas have a polynomially sized model that responds to all inputs within polynomial time, then EXPTIME

$\subseteq$ P/poly. This first step is to show the following property. Let $\mathcal{A}_b$ be an universal[1] alternating Turing machine, whose storage tape is of length b. It is possible to construct a CTL formula ϕ_b such that encodes the following specification: (1) the first input given by the environment encodes the initial tape configuration of $\mathcal{A}_b$, and (2) given this initial tape configuration, the first output given the system must correctly decide whether $\mathcal{A}_b$ halts.

Lemma 10. *There is a family of satisfiable CTL formulas ϕ_b, where all models of ϕ_b are required, in their first output, to solve the halting problem for $\mathcal{A}_b$.*

We now show how to construct a polynomial time Turing machine, with a polynomially bounded advice function, that solves the halting problem for an alternating Turing machine with polynomial space. Specifically, we will solve the halting problem for the alternating Turing machine $\mathcal{A}$, which is $\mathcal{A}_b$ where the tape is unbounded. We begin by constructing the advice function f. This function maps natural numbers to advice strings, and our algorithm is permitted to use the advice string $f(b)$, where b is the length of our input. In our case, the length of each input is the length of the initial tape of $\mathcal{A}$.

If all CTL formulas have a polynomially sized model that responds to all inputs in polynomial time, then Lemma 10 implies that, for each b, there exists an polynomial size online Turing machine, which responds to all inputs in polynomial time, and which solves the halting problem for $\mathcal{A}_b$ in its first output. Thus, we can construct an advice function f, such that $f(b)$ gives the online Turing machine that solves the halting problem for $\mathcal{A}_b$. Our assumptions imply that every advice string has polynomial length in $\mathcal{A}_b$.

Given the advice function f, we give a polynomial time algorithm for solving the halting problem for $\mathcal{A}$. For each input, we find b, which is the length of the storage tape of $\mathcal{A}$, and we obtain the online Turing machine $f(b)$. We then give $\mathcal{A}$ as the input to $f(b)$, and simulate $f(b)$ until it produces its first output. We then output the answer given by $f(b)$. By our assumptions, this can take at most polynomial time. Thus, we have shown that the halting problem for $\mathcal{A}$ lies in P/poly. Since this problem is complete for APSPACE, and since APSPACE = EXPTIME, we have shown the following theorem.

Theorem 11. *If every satisfiable CTL formula ϕ has a polynomial size model that responds to all inputs after a polynomial amount of time, then EXPTIME $\subseteq$ P/poly.*

6 EXPTIME $\subseteq$ P/poly Implies Small And Fast Models

Let ϕ be a CTL formula that has an input preserving model. In this section we show that if EXPTIME $\subseteq$ P/poly, then there always exists an polynomially sized online Turing machine that is a model of ϕ, and that responds to every input within a polynomial number of steps.

[1] here, the word "universal" means an alternating Turing machine that is capable of simulating all alternating Turing machines.

The proof of this result closely follows the proof given in Section 4. In Lemma 8 we showed that the emptiness problem can be solved in exponential time. In this proof, we will use a slightly harder problem. The inputs to our problem will be a CTL formula ϕ, a state s and input letter $\sigma_I \in 2^{\Pi_I}$ of $\mathcal{F}(\phi)$, an integer n, and a bit string w of length n. Given these inputs, the *successor emptiness problem* is to determine whether there is a letter $\sigma_O \in \Sigma_O$ such that the first n bits of σ_O are w, and σ_O satisfies the following properties: there exists $(s', \sigma'_I) \in \delta(s, (\sigma''_I, \sigma_O))$ such that $\sigma'_I \cap \Pi_I = \sigma_I$ and $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$. Once again, Lemma 7 implies that the input size of this problem is polynomial in $|\phi|$. Moreover, the problem can be solved in exponential time by cycling through all possible letters in Σ_O and applying the exponential time algorithm of Lemma 8.

If the CTL formula ϕ is fixed, then Lemma 7 implies that all other input parameters have bounded size. For a fixed formula ϕ, let $(\phi, s, \sigma_I, n, w)$ be the input of the successor emptiness problem that requires the longest representation. We pad the representation of all other inputs so that they have the same length as $(\phi, s, \sigma_I, n, w)$. Note that our algorithm for solving the successor emptiness problem still runs in exponential time, even if the inputs are padded.

Now, we can use our assumption of EXPTIME $\subseteq$ P/poly to obtain a polynomial time Turing machine with advice function f that solves the emptiness problem. Our padding ensures that we have that we have, for each CTL formula ϕ, a unique advice string in f that can be used to solve the successor emptiness problem. Therefore, by appending this advice string to the storage tape of the machine, we construct a polynomial time Turing machine that solves the successor emptiness problem. Hence, we have shown the following lemma.

Lemma 12. *If EXPTIME $\subseteq$ P/poly then, for each CTL formula ϕ, there is a polynomial time Turing machine that solves the successor emptiness problem.*

The construct of the online Turing machine that models ϕ is then the same as the one that was provided in Section 4, except we use the polynomial time Turing machine from Lemma 12 to solve the successor emptiness problem in each step. More precisely, we use binary search to find the output letter σ_O in each step. Since the size of σ_O is polynomial in $|\phi|$, this can obviously be achieved in polynomial time. Moreover, our online Turing machine still obviously uses only polynomial space. Thus, we have the main result of this section.

Theorem 13. *Let ϕ be a CTL formula that has an input preserving model. If EXPTIME $\subseteq$ P/poly then there is a polynomially sized online Turing machine $\mathcal{M}$ that models ϕ that responds to every input after a polynomial number of steps.*

7 LTL and Automata

In this section, we prove similar results for LTL. For LTL, our claims go through the intermediate automata that are usually used in LTL model checking and synthesis. That is, we turn a given specification ϕ into a Büchi word automaton for $\neg\varphi$. This automaton is a simply a universal Co-Büchi automaton.

Theorem 14. *[8] Given an LTL formula ϕ, we can construct a universal Co-Büchi automaton $\mathcal{U}_\phi$ with $2^{O(|\phi|)}$ states that accepts a transition system $\mathcal{T}$ if, and only if, $\mathcal{T}$ satisfies ϕ.*

As LTL is a trace language, this automaton is essentially a word automaton (the dual of the Büchi automaton that recognises the paths that do not satisfy ϕ): the states send into each direction are the same, and the transition function is therefore polynomial in the states of the automaton. As the arguments in Sections 4 and 6 go through a polynomial time reduction from CTL to universal Co-Büchi automata (Lemma 7), we can reuse the arguments from these sections, bearing in mind that model checking if an online Turing machine $\mathcal{M}$ is accepted by $\mathcal{U}$ can be done in in $O\big((\log|\mathcal{U}| + \log|\mathcal{T}(\mathcal{M})|)^2\big)$, using the reduction from [15] Theorem 3.2 (note that the language of $\mathcal{U}$ is the complement of the language of the same automaton read as a nondeterministic reachability automaton, where blocking translates to immediate acceptance and vice versa) to the emptiness problem of nondeterministic Büchi word automata [14].

Theorem 15. *Let $\mathcal{U}$ be a universal Co-Büchi automaton that accepts an input preserving transition system.*

1. *If PSPACE = EXPTIME then there is an online Turing machine $\mathcal{M}$ in the language of $\mathcal{U}$, where $|\mathcal{M}|$ is polynomial in the states and a representation of the transition function of $\mathcal{U}$.*
2. *If EXPTIME $\subseteq$ P/poly then there is a polynomially sized online Turing machine $\mathcal{M}$ that is accepted by $\mathcal{U}$, which responds to every input after a polynomial number of steps.*

For the other direction, we use universal safety word automata to show that the result holds for all languages in the middle. For these automata, we retrace the reduction from universal alternating Turing machines: we again interpret the first environment input as the initial configuration of a this Turing machine and henceforth use the environment to resolve the nondeterminism, the only difference is that our target language now is a universal safety word automaton rather than a CTL formula. The translations are moved to an appendix.

Theorem 16. *Let $\mathcal{U}$ be a universal safety word automaton that accepts all traces of an input preserving transition system.*

1. *If there is always an online Turing machine $\mathcal{M}$ in the language of $\mathcal{U}$, where $|\mathcal{M}|$ is polynomial in the states of $\mathcal{U}$, then PSPACE=EXPTIME.*
2. *If there is always an online Turing machine $\mathcal{M}$ in the language of $\mathcal{U}$, where $|\mathcal{M}|$ is polynomial in the states of $\mathcal{U}$ that responds to every input after polynomially many steps, then EXPTIME $\subseteq$ P/poly.*

It may look surprising that we use the intermediate automata instead of the logic. The reason for this is that it provides inroads for other logics. In fact, it would now be simple to prove similar results for the modal μ-calculus and its alternation free fragment, or to start with ACTL* instead of starting with LTL.

8 Conclusions

In our first set of proofs, we have linked the existence of small models with the equivalence of PSPACE and EXPTIME. In our second set of proofs we have linked the existence of small and fast models to the question of whether EXPTIME $\subseteq$ P/poly.

There are two ways to interpret these results. A pessimistic viewpoint is that it is extremely unlikely that EXPTIME = PSPACE or that EXPTIME $\subseteq$ P/poly. Our results therefore indicate that it is also unlikely that all CTL formulas have either small, or small and fast, models. On the other hand, an optimistic viewpoint is that finding a proof of PSAPCE $\neq$ EXPTIME, or a proof of EXPTIME $\not\subseteq$ P/poly, is considered to difficult. Under this point of view, our results indicate that it must also be very difficult to find a family of CTL formulas that provably do not have small, or small and fast, models.

References

1. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proc. of CAV*, pages 359–364, 2002.
2. R. Ehlers. Unbeast: Symbolic bounded synthesis. In *Proc. of TACAS*, pages 272–275, 2011.
3. E. Filiot, N. Jin, and J.-F. Raskin. An antichain algorithm for LTL realizability. In *Proc of CAV*, pages 263–277, 2009.
4. G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
5. R. Karp and R. Lipton. Turing machines that take advice. *Enseign. Math*, 28(2):191–209, 1982.
6. O. Kupferman and M. Y. Vardi. Freedom, weakness, and determinism: From linear-time to branching-time. In *Proc. LICS*, June 1995.
7. O. Kupferman and M. Y. Vardi. Synthesis with incomplete informatio. In *Proc. of ICTL*, pages 91–106, July 1997.
8. O. Kupferman and M. Y. Vardi. Safraless decision procedures. In *Proc. of FOCS*, pages 531–540, 2005.
9. O. Kupferman, M. Y. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. *Journal of the ACM*, 47(2):312–360, March 2000.
10. C. Papadimitriou. *Computational Complexity.* Addison Wesley Pub. Co., 1994.
11. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *Proc. of VMCAI*, pages 364–380, 2006.
12. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of POPL*, pages 179–190, 1989.
13. S. Schewe and B. Finkbeiner. Bounded synthesis. In *Proc. of ATVA*, pages 474–488, 2007.
14. 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.
15. M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. of LICS*, pages 322–331. Cambridge, UK, 1986.

A Online Turing Machines

In this section we formally define an online Turing machine. This takes two steps: we first define Turing machines with input and output, and we then introduce additional restrictions to ensure that each input is followed by exactly one output. Once this has been done, we will formally define the synthesis problem for Online Turing machines.

A.1 Space Bounded Turing Machines with Input and Output

A deterministic space bounded Turing machine with input and output is a three-tape Turing machine defined by a tuple $(S, \Sigma_I, \Sigma_T, \Sigma_O, \delta, \mathsf{start}, c, \mathsf{init}, \mathsf{input})$. The set S is a finite set of states, and the state $\mathsf{start} \in S$ is a starting state. The sets Σ_I, Σ_T, and Σ_O give the *alphabet symbols* for the input, storage, and output tapes. We require that there is a *blank* symbol $\sqcup$, such that $\sqcup$ is contained in Σ_I, Σ_T, and Σ_O. The function δ is a *transition function* which maps elements of $S \times \Sigma_I \times \Sigma_T \times \Sigma_O$ to elements of $S \times (\Sigma_I \times D) \times (\Sigma_T \times D) \times (\Sigma_O \times D)$, where $D = \{\leftarrow, -, \rightarrow\}$ is the set of *directions*. The number $c \in \mathbb{N}$ gives the *space bound* for the machine, and the sequence $\mathsf{init} \in (\Sigma_T)^c$ gives the initial contents of the storage tape, and the letter input gives the initial input symbol. We define the size $|\mathcal{M}|$ of a space bounded Turing machine $\mathcal{M}$ to be amount of space used by the tuple $(S, \Sigma_I, \Sigma_T, \Sigma_O, \delta, \mathsf{start}, c, \mathsf{init})$.

The machine has three tapes $I = I_1, I_2, \ldots,$ $T = T_1, T_2, \ldots T_c$, and $O = O_1, O_2, \ldots,$ which we call the *input*, *storage*, and *output* tapes, respectively. Note that, while the input and output tapes are infinite, the storage tape contains exactly c positions. For all $i \in \mathbb{N}$ we have $I_i \in \Sigma_I$, and $O_i \in \Sigma_O$. For all i in the range $1 \leq i \leq c$ we have that $T_i \in \Sigma_T$. The tapes are initialized as follows: the input tape I contains an infinitely long *input word*, where the first letter $I_1 = \mathsf{input}$. The output tape O contains an infinite sequences of blank symbols, and the storage tape T contains the initial storage word init.

A *position* gives the current state of the machine, along with the position of the three tape heads. Formally, a position is a tuple of the form (s, i, j, k), where $s \in S$, $i, k \in \mathbb{N}$, and j is in the range $1 \leq j \leq c$. For each $i \in \mathbb{N}$, and direction $d \in D$, we define:

$$\mathrm{Next}(i, d) = \begin{cases} i - 1 & \text{if } d = \leftarrow \text{ and } j > 0, \\ i + 1 & \text{if } d = \rightarrow, \\ i & \text{otherwise.} \end{cases}$$

We also define:

$$\mathrm{Next}^c(i, d) = \begin{cases} i - 1 & \text{if } d = \leftarrow \text{ and } j > 0, \\ i + 1 & \text{if } d = \rightarrow \text{ and } j < c, \\ i & \text{otherwise.} \end{cases}$$

The machine begins in the position $(\mathsf{start}, 0, 0, 0)$. We now describe one step of the machine. Suppose that the machine is in position (s, i, j, k), and that

$\delta(s, I_i, T_j, O_k) = (s', (\sigma_I, d_1), (\sigma_T, d_2), (\sigma_O, d_3))$. First the symbols σ_I, σ_T, and σ_O are written to I_i, T_j, and O_k, respectively. Then, the machine moves to the position $(s', \mathrm{Next}(i, d_1), \mathrm{Next}^c(j, d_2), \mathrm{Next}(k, d_3))$, and the process repeats.

A.2 Online Turing Machines

An *online* Turing machine is a Turing machine with input and output that has additional restrictions on the transition function δ. We wish to ensure the following property: the machine may only read the symbol at position i on the input tape after it has written a symbol to position $i-1$ on the output tape. Moreover, once a symbol has been written to the output tape, we require that it can never be changed. Thus, the machine must determine the first i symbols of the output before the $i+1$th symbol of the input can be read.

To this end, we partition the set S into the set S_I of *input states*, and the set S_O of *output states*, and we require that $\mathsf{start} \in S_O$. While the machine is in an output state, it is prohibited from moving the input tape head, or from moving the output tape head left. Furthermore, the machine moves from an output state to an input state only when a symbol is written to the output tape. Similarly, when the machine is in an input state it is prohibited from moving the output tape head, and the machine only moves from an input state to an output state when the input tape head is moved right. Finally, the input tape is read-only. This means that in every state, the machine is prohibited from overwriting the symbols on the input tape.

Formally, let (s, i, j, k) be a position, and suppose that $\delta(s, I_i, T_j, O_k) = (s', (\sigma_I, d_1), (\sigma_T, d_2), (\sigma_O, d_3))$. If $s \in S_I$, then we require:

- The direction d_1 is either $-$ or $\rightarrow$, and the direction d_3 is $-$.
- The symbol $\sigma_I = I_i$, and the symbol $\sigma_O = O_k$.
- If $d_1 = -$ then we require that $s' \in S_I$, and if $d_1 = \rightarrow$ then $s' \in S_O$.

Similarly, if $s \in S_O$, then we require:

- The direction d_1 is $-$, and the direction d_3 is either $-$ or $\rightarrow$.
- The symbol $\sigma_1 = I_i$, and if $d_3 = -$ then $\sigma_3 = O_k$.
- If $d_3 = -$ then $s' \in S_O$, and if $d_3 = \rightarrow$ then $s' \in S_I$.

A.3 The Synthesis Problem

We are interested in online Turing machines with input alphabet $\Sigma_I = 2^{\Pi_I}$ and output alphabet $\Sigma_O = 2^{\Pi_O}$, where Π_I is a set of input variables, and Π_O is a set of output variables. The sets Π_I and Π_O are required to be disjoint. We will use $\emptyset$ as the blank symbol for the input and output tapes.

Let $\mathcal{M}$ be an online Turing machine. For each input word σ that can be placed on the input tape, the machine produces an output word on the output tape. This output word is either an infinite sequence of outputs made by the machine, or a finite sequence of outputs, followed by an infinite sequence of blanks. The second case arises when the machine runs forever while producing only a finite number of outputs.

We define $\mathcal{M}(\sigma)$ to be the combination of the inputs given to the machine on the input variables, and the outputs made by the machine on the output variables. Formally, let $I = I_0, I_1, \ldots$, and $O = O_0, O_1, \ldots$ be the contents of the input and output tapes after the machine has been allowed to run for an infinite number of steps on the input word σ. We define $\mathcal{M}(\sigma) = \sigma_0, \sigma_1, \ldots$, where $\sigma_i = I_i \cup O_i$.

Let ϕ be an LTL formula that uses $\Pi_I \cup \Pi_O$ as the set of atomic propositions. We say that an online Turing machine $\mathcal{M}$ is a model of ϕ if $\mathcal{M}(\sigma) \models \phi$ for all input words σ. We say that ϕ is *realizable* if there exists an online Turing machine $\mathcal{M}$ that satisfies ϕ. The LTL synthesis problem for the formula ϕ is to decide whether ϕ is realizable.

On the other hand, let ϕ be a CTL formula that uses $\Pi_I \cup \Pi_O$ as the set of atomic propositions. Note that, since an online Turing machine cannot read the ith input letter until it has produced $i-1$ outputs, if σ and σ' are two input words that agree on the first i letters, then $\mathcal{M}(\sigma)$ and $\mathcal{M}(\sigma')$ must agree on the first i letters. Note also that, since the initial input letter is fixed, all of these words must agree on the first letter. Hence, the set of words $\{\mathcal{M}(\sigma) \ : \ \sigma \in (\Sigma_I)^\omega\}$ must form an infinite directed labelled tree (T, l). We say that $\mathcal{M}$ is a model of ϕ if $T, l \models \phi$.

B Proof of Lemma 6

To prove this Lemma, we first invoke the result of [7] to argue that CTL formulas can be translated to *alternating* Co-Büchi tree automata, and then argue that these automata can be translated into universal Co-Büchi tree automata. Therefore, we proceed by first giving definitions for alternating tree automata, and then providing a proof for Lemma 6.

B.1 Alternating Tree Automata

We now define alternating Co-Büchi tree automata. An *alternating Co-Büchi tree automaton* is a tuple $\mathcal{A} = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$, where S denotes a finite set of states, Σ_I is a finite input alphabet, Σ_O is a finite output alphabet, $\mathsf{start} \in S$ is an initial state, δ is a transition function, $F \subseteq S$ is a set of final states, and input is an initial input letter. The transition function $\delta : S \times (\Sigma_I \times \Sigma_O) \rightarrow \mathbb{B}^+(S \times \Sigma_I)$ maps a state and an output letter to a boolean formula that is built from elements of $S \times \Sigma_I$, conjunction $\wedge$, disjunction $\vee$, *true*, and *false*. Universal Co-Büchi tree automata correspond to alternating Co-Büchi tree automata in which all formulas given by δ are conjunctions.

The automaton runs on abstract transition systems that use Σ_I and Σ_O as their input and output alphabets. The acceptance mechanism is defined in terms of run graphs. A *run graph* of an alternating Co-Büchi tree automaton $\mathcal{A} = (S_\mathcal{A}, \Sigma_I, \Sigma_O, \mathsf{start}_\mathcal{A}, \delta_\mathcal{A}, F_\mathcal{A}, \mathsf{input})$ on an abstract Transition system $\mathcal{T} =$

$(S_{\mathcal{T}}, \Sigma_I, \Sigma_O, \tau, l_{\mathcal{T}}, \mathsf{start}_{\mathcal{T}}, \mathsf{input}_{\mathcal{T}}, \mathsf{input})$ is defined to be a minimal directed graph $G = (V, E)$ that satisfies the following constraints:

- The vertices of G satisfy $V \subseteq S_{\mathcal{A}} \times S_{\mathcal{T}}$.
- The pair of initial states $(\mathsf{start}_{\mathcal{A}}, \mathsf{start}_{\mathcal{T}})$ is contained in V.
- For each vertex $(q, t) \in V$, the set

$$\Big\{ \big(q', \sigma_I\big) \in S \times \Sigma_I \mid \Big((q,t), \big(q', \tau(t, \sigma_I)\big) \Big) \in E \Big\}$$

  is a satisfying assignment of $\delta(q, l_{\mathcal{T}}(t))$.

A run graph is *accepting* if every infinite path $v_1, v_2, v_3, \dots \in V^\omega$ contains only finitely many final states. A transition system $\mathcal{T}$ is accepted by $\mathcal{A}$ if it has an accepting run graph. The set of transition systems accepted by an automaton $\mathcal{A}$ is called its *language* $\mathcal{L}(\mathcal{A})$. An automaton is empty if, and only if, its language is empty.

The acceptance of a transition system can also be viewed as the outcome of a game, where player *accept* chooses, for a pair $(q, t) \in S_{\mathcal{A}} \times S_{\mathcal{T}}$, a set of atoms that satisfies $\delta(q, l_{\mathcal{T}}(t))$. Player *reject* then chooses one of these atoms, and then moves to the corresponding state. The transition system is accepted if, and only if, player *accept* has a strategy that ensures that all paths visit F a finite number of times.

B.2 Translating CTL to Universal Co-Büchi tree automata

The reason we are interested in alternating Co-Büchi tree automata is that there exists a translation from CTL formulas to alternating Co-Büchi tree automata.

Lemma 17. *[7] For every CTL formula ϕ there is an alternating Co-Büchi tree automaton $\mathcal{A} = (S, \Sigma_I, \sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$ such that:*

- *$\mathcal{L}(\mathcal{A})$ is the set of abstract transition systems that model ϕ.*
- *We have that $|S|$ and $|\delta|$ have size polynomial in $|\phi|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|\phi|$.*
- *The starting state can be computed in polynomial time.*

We now show that $\mathcal{A} = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta_{\mathcal{A}}, F, \mathsf{input})$, which is the alternating Co-Büchi tree automaton given by Lemma 17, can be translated into a universal Co-Büchi automaton $\mathcal{U} = (S, \Sigma_I, \Sigma'_O, \mathsf{start}, \delta_{\mathcal{U}}, F, \mathsf{input})$.

Let $\mathcal{T} \in \mathcal{L}(\mathcal{A})$ be an abstract transition system that is accepted by $\mathcal{A}$, and let $G = (V, E)$ be the run graph of $\mathcal{T}$ on $\mathcal{A}$. Note that, by the definition of a run graphs, for each state $(q, t) \in V$ we use exactly one satisfying assignment to generate the outgoing edges from (q, t). The idea behind this proof is to use the output symbols of $\mathcal{T}$ to store this satisfying assignment.

Formally, we define the extended alphabet $\Sigma'_O := \Sigma_O \cup (S \to 2^{S \times \Sigma_I})$. Each output letter of the abstract transition system contains an actual output letter

$\sigma_O \in \Sigma_O$, along with $|S|$ lists of satisfying assignments. Since S has size polynomial in ϕ, and each element of Σ_I has size in $O(|\phi|)$, each element of Σ'_O can be stored in space polynomial in $O(|\phi|)$.

Let $q \in S$ be a state of $\mathcal{A}$, let $(\sigma_I, \sigma_O) \in \Sigma_I \times \Sigma_O$ be a pair of input and output letters, and let $\gamma : S \to 2^{S \times \Sigma_I}$). If $\gamma(q)$ is a satisfying assignment of $\delta(q, (\sigma_I, \sigma_O))$, then we add the transition $\delta_{\mathcal{U}}(q, (\sigma_I, (\sigma_O, \gamma))) = \gamma(s)$. Although the function $\delta_{\mathcal{U}}$ may be exponential, we can compute, for a given q, (σ_I, σ_O), and γ, whether $\delta_{\mathcal{U}}(q, (\sigma_O, \gamma))$ is a transition in polynomial time. This is because δ has polynomial size in $|\phi|$, and checking whether $\gamma(s)$ is a satisfying assignment can easily be done in polynomial time.

Note that, for each abstract transition system $\mathcal{T} \in \mathcal{L}(\mathcal{U})$, we can use labels of each state to argue that there must be a corresponding run graph of $\mathcal{T} \upharpoonright 2^{\Pi}$ on $\mathcal{A}$. On the other hand, if $\mathcal{T}$ has a run graph on $\mathcal{A}$, then we can easily use the satisfying assignments used in this run graph to construct an abstract transition system $\mathcal{T}' \in \mathcal{L}(\mathcal{U})$ with $\mathcal{T}' \upharpoonright 2^{\Pi} = \mathcal{T}$. Therefore, we have that there exists a $\mathcal{T} \in \mathcal{L}(\mathcal{A})$ if and only if $\mathcal{T} \upharpoonright 2^{\Pi}$ is a model of ϕ. This completes the proof of Lemma 6.

C Proof of Lemma 7

From [13] we have the following lemma.

Lemma 18 ([13]). *Given the universal Co-Büchi tree automaton $\mathcal{U}(\phi)$, we can construct a deterministic safety tree automaton $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta)$ such that:*

- *We have $\mathcal{L}(\mathcal{U}(\phi)) = \mathcal{L}(\mathcal{F}(\phi))$.*
- *Each state in S can be stored in space polynomial in $|\phi|$.*
- *The transition function δ can be computed in time polynomial in $|\phi|$.*
- *Each letter in Σ_I and Σ_O can be stored in space polynomial in $|\phi|$.*
- *The state start can be computed in polynomial time.*

The only remaining step is to ensure that $\mathcal{L}(\mathcal{F}(\phi))$ contains only input preserving models. To do this, we construct a second deterministic safety tree automaton $\mathcal{F}'(\phi) = (S', \Sigma_I, \Sigma_O, \mathsf{start}', \delta', \mathsf{input})$. The set of states $S' = \Sigma_I \times S$, and the starting state $\mathsf{start}' = (\mathsf{input}, \mathsf{start})$. Let $(\sigma_I, s) \in S'$ be a state, and let (σ'_I, σ'_O) be a pair of input and output letters. We define:

$$\delta'((\sigma_I, s), (\sigma'_I, \sigma'_O)) = \begin{cases} \emptyset & \text{if } \sigma_I \neq \sigma'_I, \\ \{(\sigma_I, s), \sigma_I) \ : \ (s, \sigma_I) \in \delta((\sigma_I, s), (\sigma'_I, \sigma'_O)) & \text{otherwise.} \end{cases}$$

This construction simply eliminates all transitions of $\mathcal{F}(\phi)$ that are not input preserving, and appends the letter σ_I to the state for all transitions that are input preserving. Thus, if $L(\mathcal{F})$ contains an input preserving abstract transition system $\mathcal{T}$, then $\mathcal{T} \in \mathcal{L}(\mathcal{F}'(\phi))$. This completes the proof of Lemma 7.

D Proof of Lemma 3

Proof. Suppose that $\mathcal{M} = (S, \Sigma_I, \Sigma_T, \Sigma_O, \delta, \mathsf{start}, c, \mathsf{init}, \mathsf{input})$, and let (S_I, S_O) be the partition of S into the input and output states. We will assume that $\Sigma_I = 2^{\Pi_I}$, and that $\Sigma_O = 2^{\Pi_O}$, for some sets Π_I and Π_O of input and output propositions.

We define $\mathcal{T}(\mathcal{M}) = (S_\mathcal{T}, \Sigma_I, \Sigma_O, \tau_\mathcal{T}, l_\mathcal{T}, \mathsf{start}_\mathcal{T}, \mathsf{input})$ as follows. The state space is defined to be the union of three sets. Firstly we have the normal states $S_N = S \times \Sigma_I \times \mathbb{N}_{\leq c} \times \Sigma_O \times (\Sigma_T)^c$, , where $\mathbb{N}_{\leq c} = 1, 2, \dots, c$ which represent the computational states that the Turing machine can be in. We also have a special failure state fail, which will be used to indicate that the Turing machine runs forever without writing an output symbol. Finally, for each state $a \in S_N$ we have a special failure state fail_a, which will be used in the case where the Turing machine reads an input, writes an output, and then runs forever without reading another input symbol. The state fail_a will be used to hold the final output of the machine, before we move to fail. Therefore, we define $S_\mathcal{T} = S_N \cup \mathsf{fail} \cup \{\mathsf{fail}_a : a \in S_N\}$.

All states $a \in S_N$ are tuples of the form $(s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \dots, T_c\rangle)$, where s is a state of the Turing machine, j is the current position of the storage tape head, σ_I is the symbol at the head of the input tape, σ_O is the last symbol written to the output tape, and T is the current state of the storage tape. Since we know that $I_1 = \mathsf{input}$ and $O_1 = \emptyset$ in the initial state of the machine, the starting state of the abstract transition system will be $\mathsf{start}_\mathcal{T} = (\mathsf{start}, \mathsf{input}, 1, \emptyset, \mathsf{init})$.

We now define $\tau_\mathcal{T}$. Let $a = (s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \dots, T_c\rangle)$ be a normal state. Note that the definition of an online Turing machine ensures that there is always a blank symbol at the head of the output tape. Therefore, suppose that:

$$\delta(s, \sigma_I, T_j, \emptyset) = (s', (\sigma'_I, d_1), (\sigma'_T, d_2), (\sigma'_O, d_3)).$$

If a state a has $d_1 =\rightarrow$, then we say that a is an *input state*, and if a has $d_3 =\rightarrow$, then we say that a is an *output state*. The transition function $\tau_\mathcal{T}$ will only define transitions for input states and for $\mathsf{start}_\mathcal{T}$, as these will be the only states that are reachable from the starting state of the transition system. For all other states s we define $\tau_\mathcal{T}(s, \sigma_I) = \mathsf{fail}$ for all input letters $\sigma_I \in \Sigma_I$.

Before we define $\tau_\mathcal{T}$, we first define a helper function Succ. This function will allow us to find the transitions between input states. Let $a = (s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \dots, T_c\rangle)$ be a normal state, and let T' be the tape T with the jth symbol replaced with σ'_T. If $d_3 = -$ then we define $\mathrm{Succ}(a)$ to be

$$a' = (s', \sigma_I, \mathrm{Next}(j, d_2), \sigma_O, T').$$

On the other hand, if and $d_3 =\rightarrow$ then we define $\mathrm{Succ}(a)$ to be

$$a' = (s', \sigma_I, \mathrm{Next}(j, d_2), \sigma'_O, T').$$

Note that this definition correctly remembers the last symbol that was written to the output tape.

For each input state $a = (s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \dots, T_c \rangle)$ and each input letter σ_I, let a' be $\text{Succ}(a)$, where the input letter is replaced by Σ_I. We define $\pi(a, \sigma_I)$ be the path that starts at a' and follows $\text{Succ}(a)$ until another input state is reached. Note that this path may be infinite if the Turing machine runs forever without requesting an input. If $\pi(a, \sigma_I)$ ends at a state a'', then we define $\tau(a, \sigma_I) = a''$. On the other hand, if $\pi(a, \sigma_I)$ is an infinite path, then we have two cases to consider. If $\pi(a, \sigma_I)$ never visits an output state, then we define $\tau_{\mathcal{T}}(a, \sigma_I) = \mathsf{fail}$. On the other hand, if $\pi(a, \sigma_I)$ does visit an output state a'', then we define $\tau_{\mathcal{T}}(a, \sigma_I) = \text{Succ}(a'')$. This is because $\text{Succ}(a'')$ is the first state that correctly remembers the output made at a''. For each state $a \in S_N$ we define $\tau_{\mathcal{T}}(\mathsf{fail}_a, \sigma_I) = \mathsf{fail}$, for all input letters $\sigma_I \in \Sigma_I$. We also define $\tau_{\mathcal{T}}(\mathsf{fail}, \sigma_I) = \mathsf{fail}$ for all input letters $\sigma_I \in \Sigma_I$.

Finally, we define the $l_{\mathcal{T}}$. For each normal state $a = (s, \sigma_I, j, \sigma_O, T)$ we define $l_{\mathcal{T}}(a) = (\sigma_I, \sigma_O)$. Note that this implies that the transition system is input preserving, because at an input state, the parameter σ_I must contain the input that corresponds to the last output. For each state fail_a with $a = (s, j, \sigma_O, T)$ we define $l_{\mathcal{T}}(\mathsf{fail}_a) = \sigma_O$. Finally, we define $l_{\mathcal{T}}(\mathsf{fail}) = \emptyset$, which is the blank symbol for the output tape of an online Turing machine.

To see that this reduction is correct, note that, in an online Turing machine, exactly one output is written to the output tape for each input that is read from the input tape. Therefore, for every state a, our transition function $l_{\mathcal{T}}(a, \sigma_I)$ correctly moves to a state $a' = (s, j, \sigma_O, T)$, where σ_O is the output given by the online Turing machine for the input σ_I. Moreover, if the Turing machine runs for an infinite number of steps while producing only a finite number of outputs, then $\mathcal{T}$ will correctly produce an infinite sequence of blank symbols, and it also correctly outputs the final output symbol. Therefore, we have that $\mathcal{T}(\mathcal{M})$ is a model of ϕ if and only if $\mathcal{M}$ is a model of ϕ. □

E Proof of Lemma 4

Proof. Given the online Turing machine $\mathcal{M} = (S, \Sigma_I, \Sigma_T, \Sigma_O, \delta, \mathsf{start}, c, \mathsf{init})$, our task is to output $\mathcal{T}(\mathcal{M}) = (S_{\mathcal{T}}, \Sigma_{\mathcal{T}}, \tau_{\mathcal{T}}, l_{\mathcal{T}}, \mathsf{start}_{\mathcal{T}})$. Recall that each normal state $s \in S_N$ is a tuple of the form (s, j, σ_O, T). Obviously the parameters s, j, and σ_O, can be stored in $O(\mathcal{M})$ space. Moreover, since the description of $\mathcal{M}$ contains start, which is a tape of length c, the tape T can also be stored in $O(|\mathcal{M}|)$ space. Since the state space of $S_{\mathcal{T}}$ consists of S_N, fail, and fail_a for each $a \in S_N$, we have that each state of $\mathcal{T}(\mathcal{M})$ can be stored in $O(|\mathcal{M}|)$ space.

Our algorithm for outputting $\mathcal{T}(\mathcal{M})$ is as follows. We begin by outputting $\mathsf{start}_{\mathcal{T}}$. Then we output the state fail along with the outgoing transitions from fail. We then cycle through each normal state $s \in S_N$, and output s and fail_s, along with $l_{\mathcal{T}}(s)$ and $l_{\mathcal{T}}(\mathsf{fail}_s)$. We also output the outgoing transitions from fail_s. All of these operations can obviously be done in $O(|\mathcal{M}|)$ space.

Finally, we must argue, for each normal state $s \in S_N$, that the outgoing transition from s can be computed in $O(|\mathcal{M}|)$ space. To compute $\tau_{\mathcal{T}}(s, \sigma_I)$ for some input letter $\sigma_I \in \Sigma_I$, we iteratively follow the function $\text{Succ}(s, \sigma_I)$ until we find an input state. If, while iterating $\mathrm{Succ}(s, \sigma_I)$, we encounter an output state s', then we remember it. If we find an input state s', then we output $\tau_{\mathcal{T}}(s, \sigma_I) = s'$. On the other hand, we may never find an input state. Therefore, we also maintain a counter, which counts the number of times that Succ has been followed. If the counter reaches $|S_N|$, then we know that the online Turing machine must run forever without reading its next input. In this case, if an output state s' has been remembered, then we output $\tau_{\mathcal{T}}(s, \sigma_I) = \mathsf{fail}_{\mathrm{Succ}(s', \sigma_I)}$. Otherwise, we output $\tau_{\mathcal{T}}(s, \sigma_I) = \mathsf{fail}$.

To implement this procedure, we must remember at most 3 states, one for s, one for the current state, and one for the output state that must be remembered. We must also maintain a counter that uses $\log(|S_N|)$ bits, and $|S_N| \in 2^{O(n)}$. Therefore, this procedure can be implemented in $O(|\mathcal{M}|)$ space. □

F Proof of Theorem 5

Proof. We show that, under the assumption that there is always a model of size c, the CTL synthesis problem can be solved in polynomial space. Since Theorem 2 implies that CTL synthesis is EXPTIME-complete, we will therefore prove that PSPACE = EXPTIME.

The algorithm is as follows. We first non-deterministically guess an online Turing machine $\mathcal{M}$ with with $|\mathcal{M}| \in O(\mathrm{poly}(|\phi|))$. Then we model check against the input formula ϕ, using the Turing machine given by Lemma 4 and the Turing machine given by Theorem 1. Since the output of the first Turing machine has size $2^{O(|\mathcal{M}|)}$, we have that the second Turing machine uses $O(|\mathcal{M}|)$ space. Using standard techniques to compose space bounded Turing machines (see [10, Proposition 8.2], for example), we obtain a Turing machine that solves the CTL synthesis problem in NPSPACE = PSPACE. □

G Proof of Lemma 8

Proof. From the formula ϕ we can construct $\mathcal{F}^s(\phi) = (S, \Sigma_I, \Sigma_O, s, \delta, F)$ in exponential time by doing the following: first we loop through each possible state in S and output it. Since Lemma 7 guarantees that each state can be stored in space polynomial in ϕ, this procedure can take at most exponential time in ϕ. Then, we loop through each member of $S \times \Sigma_O$. Again, since Lemma 7 implies that each member of $S \times \Sigma_O$ can be written in polynomial space, and therefore this procedure takes at most exponential time.

So far we have shown that the states and transitions of $\mathcal{F}^s(\phi)$ can be constructed in exponential time, while using exponential space. We call a state $s' \in S$ *rejecting* if $\delta(s', \sigma_O) = \emptyset$ for all output letters $\sigma_O \in \Sigma_O$. It is not difficult to see that $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$ if, and only if, all possible paths from s lead to a rejecting state. Thus, we can solve the emptiness problem by solving a simple reachability query on the automaton that we have constructed. Since reachability can be solved in polynomial time, and the description of our automaton uses exponential space, this reachability query can be answered in exponential time. □

H Proof of Lemma 10

Proof. We first define the set of output propositions Π_O that will be used by our CTL formula. Our intention is that each letter $\sigma_O \in \Sigma_O = 2^{\Pi_O}$ should encode a configuration of an alternating Turing machine with a storage tape of length b. This the storage tape itself can be represented using $b' = b \cdot \log_2 |\Sigma_T|$ atomic propositions $p_1, \cdot, p_{b'}$. We also use b atomic propositions $t_1, \dots, t_b$ to encode the position of the tape head: the propositions t_i is true if and only if the tape head is at position i of the tape. We use $l = \log_2(|Q|)$ atomic propositions, where Q is the set of states in our Turing machine, to encode q, which is the current state in the configuration. We also use $l + b' + \log_2 |b|$ atomic propositions to encode a counter c, which will count the number of steps that have been executed. Finally, and most importantly, we include one proposition h, and we will require that h accurately predicts whether the alternating Turing machine will eventually halt from the current configuration. We will use $\Sigma_I = \Sigma_O$.

We now specify the CTL formula ϕ_b. The first input symbol will be interpreted as the initial state of our alternating Turing machine $\mathcal{A}$. Since our machine is an alternating machine, the transition function between configurations is not deterministic. Instead, in each step there is either a universal or nondeterministic choice that must be made. We will allow the environment to resolve these decisions. Since Σ_I contains enough letters to encode every possible configuration of $\mathcal{A}$, there are obviously more than enough letters in Σ_I to perform this task. Once the nondeterminism or universality has been resolved, the formula requires the model to output the next configuration of the alternating Turing machine. In other words, the environment will pick a specific branch of the computation of $\mathcal{A}$, and therefore all models of ϕ must be capable of producing all possible computation branches of $\mathcal{A}$. It is not difficult to produce a CTL formula that encodes these requirements.

However, we have one final requirement that must be enforced: that the proposition h correctly predicts whether the current configuration eventually halts. This can be achieved by adding the following requirements to our CTL formula.

- If q is an accepting state, then h must be true.
- If c has reached its maximum value, then h must be false.
- If q is non-accepting and c has not reached its maximum value then:
  - If q is an existential state, then $h \leftrightarrow E\mathcal{X}h$.
  - If q is a universal state, then $h \leftrightarrow A\mathcal{X}h$.

Therefore, we have constructed a CTL formula ϕ such that, for every model of ϕ, the first output from the model must solve the halting problem for a b-space bounded alternating Turing machine. □

I LTL and Automata

I.1 LTL Formulas

Given a finite set Π of atomic propositions, the syntax of an LTL formula is defined as follows:

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid \mathcal{X}\phi \mid \phi \,\mathcal{U}\, \phi,$$

where $p \in \Pi$. For each LTL formula, we define $|\phi|$ to give the size of the formula, which is the size of the parse tree for that formula.

Let $\sigma = \sigma_0, \sigma_1, \ldots$ be an infinite word where each symbol $\sigma_i \in \Pi$. For each $i \in \mathbb{N}$, we define the semantics of an LTL formula ϕ as follows:

- $\sigma, i \models p$ if and only if $p \in \sigma_i$.
- $\sigma, i \models \neg\phi$ if and only if $\sigma, i \not\models \phi$.
- $\sigma, i \models \phi \vee \psi$ if and only if either $\sigma, i \models \phi$ or $\sigma, i \models \psi$.
- $\sigma, i \models \mathcal{X}\phi$ if and only if $\sigma, i+1 \models \phi$.
- $\sigma, i \models \phi \,\mathcal{U}\, \psi$ if and only if there exists $n \geq i$ such that $\sigma, n \models \psi$ and for all j in the range $i \leq j < n$ we have $\sigma, j \models \phi$.

A word σ is a model of an LTL formula ϕ if and only if $\sigma, 0 \models \phi$. If σ is a model of ϕ, then we write $\sigma \models \phi$.

Let ϕ be an LTL formula that uses $\Pi_I \cup \Pi_O$ as a set of atomic propositions. We say that the abstract transition system is a model of ϕ if $\sigma(\pi) \models \phi$ for every infinite path π that begins at the starting state. Given an LTL formula ϕ and an abstract transition system $\mathcal{T}$, the LTL model checking problem is to decide whether $\mathcal{T}$ is a model of ϕ.

Theorem 19 ([15]). *Given an abstract transition system $\mathcal{T}$ and an LTL-formula ϕ, the LTL model checking problem can be solved in $O((\log|\mathcal{T}| + |\phi|)^2)$ space.*

The LTL synthesis problem is defined in the same way as the CTL synthesis problem: given an LTL formula ϕ, we must decide whether there exists an abstract transition system that is a model of ϕ.

Theorem 20 ([12]). *The LTL synthesis problem is 2EXPTIME-complete.*

I.2 Proofs for Theorem 16

Lemma 21. *Let $\mathcal{U}$ be a realisable universal safety word automaton. If there always exists an online Turing machine $\mathcal{M}$ that realises $\mathcal{U}$ while taking only $O(poly(|\mathcal{U}|))$ time between reading two input letters, then EXPTIME $\subset$ P/poly.*

Proof. As in the proof of Lemma 10, we use a reduction from the halting problem of a universal space bounded alternating Turing machine.

We first define the set of output propositions Π_O that will be used by our universal Co-Büchi automaton. Our intention is again that each letter $\sigma_O \in \Sigma_O = 2^{\Pi_O}$ should encode a configuration of an alternating Turing machine with a storage tape of length b. This the storage tape itself can be represented using $b' = b \cdot \log_2 |\Sigma_T|$ atomic propositions $p_1, \cdots, p_{b'}$. We also use b atomic propositions $t_1, \ldots, t_b$ to encode the position of the tape head: the propositions t_i is true if and only if the tape head is at position i of the tape. We use $l = \log_2(|Q|)$ atomic propositions, where Q is the set of states in our Turing machine, to encode q, which is the current state in the configuration. We also use $l + b' + \log_2 |b|$ atomic propositions to encode a counter c, which will count the number of steps that have been executed. Finally, and most importantly, we include one proposition h, and we will require that h accurately predicts whether the alternating Turing machine will eventually halt from the current configuration. Different to the reduction from CTL, we also have to include a way to resolve existential choices in the model. We therefore also include atomic propositions that refer to the directions that serve as witnesses for the fact that h is *true* for existential states or *false* for universal states. We refer to this successor as the *witness successor*.

We now specify the universal safety automaton $\mathcal{U}$. Since $\mathcal{A}$ is an alternating Turing machine, the transition function between configurations is not deterministic. Instead, in each step there is either a universal or an existential choice that must be made. We will allow the environment to resolve these decisions. Since Σ_I contains enough letters to encode every possible configuration of $\mathcal{A}$, there are obviously more than enough letters in Σ_I to perform this task.

To check the correctness of these transitions, the universal safety automaton would, for each transition, have a corresponding input letter. It would send, for each cell of the tape of $\mathcal{A}$, for the finite control of $\mathcal{A}$, and for the position the read/write head should be in after the transition, a state to all successors. This state would not only contain this first input symbol will be interpreted as the initial state of our alternating Turing machine $\mathcal{A}$. While it sends the obligations to all successors, they are only interpreted on the single successor where the input read (and hence represented in the label) is σ. (Reading a different input leads to immediate acceptance.)

Once the existential and universal decisions of $\mathcal{A}$ have been resolved, the formula requires the model to output the next configuration of the alternating Turing machine. In other words, the environment will pick a specific branch of the computation of $\mathcal{A}$, and therefore all transition systems in the language of $\mathcal{U}$ must be capable of producing all possible computation branches of $\mathcal{A}$. It is not difficult to produce a universal safety automaton that encodes these requirements.

However, we have one final requirement that must be enforced: that, in the first step of the computation, the proposition h correctly predicts whether the current configuration eventually halts. This can be achieved by adding the following requirements to our universal safety automaton.

- If q is an accepting state, then h must be true.
- If q is non-accepting and c has reached its maximum value, then h must be false.
- If q is non-accepting and c has not reached its maximum value then:
  - If q is an existential state and h is true, then h must be true for the witness successor.

- If q is an existential state and h is false, then h must be false for all successors.
- If q is an existential state and h is true, then h must be true for all successors.
- If q is an universal state and h is false, then h must be false for the witness successor.

Therefore, we have constructed a universal safety word automaton $\mathcal{U}$ such that, for every model of $\mathcal{U}$, the first output from the model must solve the halting problem for a b-space bounded alternating Turing machine. $\square$

Theorem 22. *Let $\mathcal{U}$ be a realisable universal safety word automaton. If there always exists an online Turing machine $\mathcal{M}$, with $|\mathcal{M}| \in O(poly(|\mathcal{U}|))$, that is accepted by $\mathcal{U}$, then PSPACE = EXPTIME.*

Proof. A proof that the synthesis problem for these automata is EXPTIME complete is contained in the proof of the previous lemma.

Model checking if $\mathcal{M}$ is accepted by $\mathcal{U}$ can be done in space $O\big((\log|\mathcal{U}| + \log|\mathcal{T}(\mathcal{M})|)^2\big)$, using the reduction from [15] Theorem 3.2 (note that the language of $\mathcal{U}$ is the complement of the language of the same automaton read as a nondeterministic reachability automaton, where blocking translates to immediate acceptance and vice versa) to the emptiness problem of nondeterministic Büchi word automata [14]. $\square$