Journal of Computer and System Sciences 81 (2015) 1171–1193

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Synthesis of succinct systems ☆

John Fearnley [a], Doron Peled [b], Sven Schewe [a,*]

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

ARTICLE INFO

Article history:
Received 6 June 2013
Received in revised form 25 September 2014
Accepted 13 February 2015
Available online 4 March 2015

Keywords:
Succinct synthesis
CTL
LTL
Online Turing machine

ABSTRACT

Synthesis of correct by design systems from specifications has recently attracted a lot of attention. The theoretical results imply that this problem is highly intractable. For example, synthesizing a system is 2EXPTIME-complete for an LTL specification and EXPTIME-complete for CTL. An argument in favour of synthesis is that temporal specifications are highly compact, and the complexity reflects the large size of the system constructed. 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 slightly biased, in that the state space can be exponentially larger than the size of a reasonable implementation like a circuit or program. This raises the question if there exists a small bound on the circuits or programs. We show that small succinct model theorems depend on the collapse of complexity classes, e.g., of PSPACE and EXPTIME for CTL.

1. Introduction

Synthesis of reactive systems is a research direction inspired by Church's realisability problem [2]. It focuses on systems that receive a constant stream of inputs from an environment, and must, for each input, produce some output. Specifically, we are given a logical specification that dictates how the system must react to the inputs, and we must construct a system that satisfies the specification for all possible inputs that the environment could provide.

While the verification problem [3,8] (the validation or refutation of the correctness of such a system) has gained many algorithmic solutions and various successful tools, the synthesis problem [4,6,7,15,18] has had fewer results. One of the main problems is the complexity of the synthesis problem. A classical result by Pnueli and Rosner [16] shows that synthesis of a system from an LTL specification is 2EXPTIME-complete. It was later shown by Kupferman and Vardi that synthesis for CTL specifications is EXPTIME-complete [11]. A counter argument against the claim that synthesis has prohibitive high complexity is that the size of the system produced by the synthesis procedure is typically large. Some concrete examples [10] show that the size of the system synthesized may need to be doubly exponentially larger than the LTL specification. Thus, arguably, the high complexity of the synthesis problem arises due to the very compact nature of LTL specifications. This, in fact, shows that LTL specifications are a very compact representation of the requirements of a system, rather than simply a formalism that is intrinsically hard for synthesis.

☆ This work was supported by the Engineering and Physical Science Research Council through the grants EP/H046623/1 'Synthesis and Verification in Markov Game Structures' and EP/L011018/1 'Algorithms for Finding Approximate Nash Equilibria', by the Israel Science Foundation through grant 126-12 'Practical Synthesis of Control for Distributed Systems', and by a short visit grant 'Circuit Complexity for Synthesis' within the framework of the ESF activity 'Games for Design and Verification'. A preliminary version of this article appeared in ATVA 2012 [5].

* Corresponding author.
E-mail addresses: john.fearnley@liverpool.ac.uk (J. Fearnley), doron.peled@gmail.com (D. Peled), sven.schewe@liverpool.ac.uk (S. Schewe).

http://dx.doi.org/10.1016/j.jcss.2015.02.005
0022-0000/

As we are interested in the relationship between the specification and the synthesized system, we must pay careful attention 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 states and transitions. This is, to some extent, a biased measure, as other system representations, such as programs or 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. We ask the following question: *what is the size of the minimal system representation in terms of the specification?*

We look at specifications given in CTL, or LTL, and study the relative synthesized system complexity. We choose to represent our systems as online Turing machines with a bounded storage tape. This is because there exist straightforward translations between online Turing machines and the natural representations of a system, such as programs and circuits, with comparable representation size. The online Turing machine uses a read-only input tape to read the next input, a write-only output tape to write the corresponding output, and its storage tape to serve as the memory required to compute the corresponding output for the current input.

The binary-counter example mentioned above shows that there are instances in which an online Turing machine model of the specification 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 is a model of 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:

Every CTL formula has a small online Turing machine model (or no model at all) if, and only if, PSPACE = EXPTIME.

This result can be read in two ways. One point of view is that, since PSPACE is widely believed to be a proper subset of EXPTIME, the "only if" direction of our result implies that it is unlikely that every CTL formula has a small online Turing machine model. However, there is an opposing point of view. It is widely believed that finding a proof that PSPACE $\neq$ EXPTIME is an extremely difficult problem. The "if" direction of our result implies that, if we can find a family of CTL formulas that provably requires super-polynomial sized online Turing machine models, then we have provided a proof that PSPACE $\neq$ EXPTIME. If it is difficult to find a proof that PSPACE $\neq$ EXPTIME, then it must also be difficult to find such CTL formulas. This indicates that most CTL formulas, particularly those that are likely to arise in practice, might have small online Turing machine models.

The definition of a small online Turing machine does not give a fixed bound on the size of the machine. For example, it may be the case that all realisable CTL formulas ϕ have an online Turing machine model of size $|\phi|^{100}$. These models, while still being polynomial, are of little practical interest. To address this issue, we prove a second result:

If P = LOGSPACE, then every realisable CTL formula ϕ has an online Turing machine model of size $O(|\phi|^2 \cdot \log|\phi|)$.

Therefore, if we can find a family of CTL formulas that require online Turing machine models larger than $O(|\phi|^2 \cdot \log|\phi|)$, then we have shown that P $\neq$ LOGSPACE. This suggests that many of the CTL formulas that appear in practice might indeed have reasonably sized online Turing machine models.

We also study a second question. Using an online Turing machine raises the issue of the time needed to respond to an input. In principle, a polynomially-sized online Turing machine can take exponential time to respond to each input. A small model may, therefore, take exponential time in the size of the CTL formula to produce each output. This leads to the second question that we address in this paper: for CTL, does there always exist a small online Turing machine model that is *fast*? The model 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 a small and fast (or no) online Turing machine model 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 EXPTIME $= \Sigma_2^P$ [9], which means that the polynomial time hierarchy collapses to the second level, and that EXPTIME itself is contained in the polynomial time hierarchy. The polynomial time hierarchy is an infinite sequence of complexity classes that are contained in PSPACE. It is widely believed that the hierarchy is strict, that is, every level of the hierarchy is strictly contained in the next, and that the hierarchy itself is strictly contained in PSPACE. Therefore, EXPTIME $\subseteq$ P/poly seems very unlikely to be true.

Once again, this result can be read in two ways. Since many people believe that the polynomial hierarchy is strict, the "if" direction of our result implies that it is unlikely that all CTL formulas have small and fast models. On the other hand, the "only if" direction of the proof implies that finding a family of CTL formulas that do not have small and fast online Turing machine models is as hard as proving that EXPTIME is not contained in P/poly. As before, if finding a proof that EXPTIME $\not\subseteq$ P/poly is a difficult problem, then finding CTL formulas that do not have small and fast models must also be a difficult problem. This indicates that the CTL formulas that arise in practice might have small and fast models.

We also consider LTL specifications. Our definitions of "small" and "fast" must change when we consider LTL formulas, because LTL models may be exponentially larger than CTL models: while every CTL formula has a transition system model with exponentially many states, there are LTL formulas that require transition system models with doubly-exponentially many states. Thus, an online Turing machine is "small" with respect to an LTL formula if it uses exponential space with respect to the formula. Furthermore, an online Turing machine is "fast" if it responds to each input in exponential time.

This difference in definitions leads to some less elegant results. We show that if every LTL formula has a small online Turing machine model, then EXPSPACE = 2EXPTIME. In the opposite direction, we show that if PSPACE = EXPTIME, then every LTL formula has a small online Turing machine model. Hence, unlike for CTL, we fail to obtain an "if and only if" characterisation for the problem. When we consider small and fast models, we can only prove one direction: if EXPTIME $\subseteq$ P/poly, then every LTL formula has a small and fast online Turing machine model, but the techniques do not generate a meaningful result for the opposite direction. Since the CTL results are more elegant, we first describe how the results can be obtained for CTL, and then show how they can be adapted for the LTL setting.

Finally, we study the related problem of bounded synthesis for online Turing machines. This problem is: given a specification ϕ, and a bound $b \in \mathbb{N}$, does there exists an online Turing machine model of ϕ whose size is at most b? We show that this problem is PSPACE-complete for CTL and LTL specifications.

2. Preliminaries

2.1. LTL formulas

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

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

where $p \in \Pi$. For each LTL formula, we define $|\phi|$ to give the length of the formula. Throughout this paper, we will assume that our formulas ϕ use each proposition in Π at least once. Thus, we can assume that Π can be written down in $O(|\phi|)$ many bits.

Let $\sigma = \sigma_0, \sigma_1, \ldots$ be an infinite word where each symbol $\sigma_i \subseteq \Pi$. 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 $\sigma, i \models \psi$.
- $\sigma, i \models X\phi$ if and only if $\sigma, i+1 \models \phi$.
- $\sigma, i \models \phi U \psi$ 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 $\sigma \models \phi$.

2.2. CTL formulas

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

$$\phi ::= p \mid \neg\phi \mid \phi \vee \phi \mid A\psi \mid E\psi,$$

$$\psi ::= X\phi \mid \phi U \phi \mid G\phi,$$

where $p \in \Pi$. For each CTL formula ϕ we define $|\phi|$ to give the length of ϕ.

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

- $\pi \models X\psi$ if and only if $v_2, v_3, v_4, \ldots \models \psi$.
- $\pi \models \phi U \psi$ if and only if there exists $i \in \mathbb{N}$ such that $v_i, v_{i+1}, v_{i+2}, \ldots \models \psi$ and for all j in the range $1 \leq j < i$ we have $v_j, v_{j+1}, v_{j+2}, \ldots \models \phi$.
- $\pi \models G\phi$ if and only if $v_i, v_{i+1}, v_{i+2}, \ldots \models \phi$ for all i.

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 $T, l \models \phi$. For every CTL formula ϕ, we define $\mathrm{cl}(\phi)$ to give the set of all sub-formulas of ϕ.

2.3. Mealy machines

The synthesis problem is to construct a model that satisfies the given specification. We will give two possible formulations for a model. Traditionally, the synthesis problem asks us to construct a model in the form of a transition system. We will represent these transition systems as *Mealy machines*, which we will define in this section. In a later section we will give online Turing machines as an alternative, and potentially more succinct, model of a specification.

A Mealy machine 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 the starting state. The set Σ_I gives an input alphabet, and the set Σ_O gives an output alphabet. The transition function $\tau : S \times \Sigma_I \to S$ gives, for each state and input letter, an outgoing transition. The function $l : S \times \Sigma_I \to \Sigma_O$ is a labelling function, which assigns an output letter for each transition. The letter $\mathsf{input} \in \Sigma_I$ gives an initial input letter for the machine.

Suppose that ϕ is an LTL formula over the set Π of atomic propositions. Suppose that $\Pi = \Pi_I \cup \Pi_O$, where 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 a Mealy machine that uses sets of these propositions as input and output alphabets. A sequence of states $\pi = s_0, s_1, s_2, \ldots$ is an infinite path in $\mathcal{T}$ if $s_0 = \mathsf{start}$, and if, for each i, there is a letter $\sigma_i \in \Sigma_I$ such that $\tau(s_i, \sigma_i) = s_{i+1}$. We define ω_i to be the set of input and output propositions at position i in the path, that is we define $\omega_i = \sigma_i \cup l(s_i, \sigma_i)$, where we take $\sigma_0 = \mathsf{input}$. Then, for each infinite path π, we define the word $\sigma(\pi) = \omega_0, \omega_1, \omega_2 \ldots$ to give the sequence of letters along the path π.

We say that $\mathcal{T}$ 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 a Mealy machine $\mathcal{T}$, the LTL model checking problem is to decide whether $\mathcal{T}$ is a model of ϕ.

Theorem 1. *(See [19].) Given a Mealy machine $\mathcal{T}$ and an LTL-formula ϕ, the LTL model checking problem can be solved in $O((\log|\mathcal{T}| + |\phi|)^2)$ space.*

Given an LTL formula ϕ, defined over a set $\Pi = \Pi_I \cup \Pi_O$ of atomic propositions, the *LTL synthesis problem* is to decide whether there exists a Mealy machine that is a model of ϕ. This problem is known to be 2EXPTIME-complete.

Theorem 2. *(See [16].) The LTL synthesis problem is 2EXPTIME-complete.*

We can give analogous definitions for CTL formulas. Let ϕ be a CTL formula over the set $\Pi = \Pi_I \cup \Pi_O$ of atomic propositions, and suppose that $\mathcal{T} = (S, 2^{\Pi_I}, 2^{\Pi_O}, \tau, l, \mathsf{start}, \mathsf{input})$ is a Mealy machine. Let (T, l) be the infinite tree corresponding to the set of words $\sigma(\pi)$, over all possible infinite paths π. More precisely, (T, l) is the tree where every vertex has exactly Π_I many children, and every finite path starting at the root appears exactly once. We say that $\mathcal{T}$ is a model of ϕ if $T, l \models \phi$. Given a CTL formula ϕ and a Mealy machine $\mathcal{T}$, the CTL model checking problem is to decide whether $\mathcal{T}$ is a model of ϕ.

Theorem 3. *(See [13].) Given a Mealy machine $\mathcal{T}$ and a CTL-formula ϕ, the CTL model checking problem can be solved in space polynomial in $O(|\phi| \cdot \log^2(|\mathcal{T}| \cdot |\phi|))$.*

Given a CTL formula ϕ, the *CTL synthesis problem* is to decide whether there exists a Mealy machine that is a model of ϕ. This problem is known to be EXPTIME-complete.

Theorem 4. *(See [11].) The CTL synthesis problem is EXPTIME-complete.*

2.4. Online Turing machines

In this section, we define Online Turing machines as an alternative to Mealy machines. As we saw in the introduction, there are some CTL specifications that can be modelled more succinctly by online Turing machines, and the central question of this paper is whether all CTL specifications have this property.

2.4.1. Space bounded Turing machines with input and output

A deterministic c-space bounded Turing machine with input and output is a Turing machine with three tapes, called the *input*, *storage*, and *output* tapes. Formally, a deterministic c-space bounded Turing machine with input and output is 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 $\mathsf{input} \in \Sigma_I^\omega$ gives the contents of the input tape.

We denote the input, storage, and output tapes of the machine as $I = I_1, I_2, \ldots$, $T = T_1, T_2, \ldots T_c$, and $O = O_1, O_2, \ldots$, 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 *configuration* gives a complete description of the state of the machine in a given computation step. Formally, a configuration is a tuple (s, i, j, k, I, T, O) where $s \in S$, $i, k \in \mathbb{N}$, and j is in the range $1 \leq j \leq c$. The parameters I, T, and O are the three tapes: I and O are always infinite, while T always has length c.

For each $i \in \mathbb{N}$, and direction $d \in D$, we define a function that maps directions to head positions:

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

We also define a version for the storage tape, which does not move the head past the end of the tape:

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

The machine begins in the configuration $(\mathsf{start}, 0, 0, 0, \mathsf{input}, \mathsf{init}, \sqcup^\omega)$. We now describe one step of the machine. Suppose that the machine is in configuration (s, i, j, k, I, T, S), and that $\delta(s, I_i, T_j, O_k) = (s', (\sigma_I, d_1), (\sigma_T, d_2), (\sigma_O, d_3))$. Let $I' = I_1, I_2, \ldots I_{i-1}, \sigma_I, I_{i+1}, \ldots$ be the result of replacing the ith symbol in tape I with σ_I. Similarly, let T' be the result of replacing the jth symbol in T with σ_T, and let O' be the result of replacing the kth symbol in O with σ_O. The machine moves to the configuration $(s', \mathrm{Next}(i, d_1), \mathrm{Next}^c(j, d_2), \mathrm{Next}(k, d_3), I', T', O')$.

2.4.2. Online Turing machines

An *online* Turing machine is a space bounded Turing machine with input and output with additional restrictions. 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,
- it may not write to the input tape,
- it may not move the output tape head left, and
- it moves to an input state if, and only if, it moves the output tape head right.

Similarly, when the machine is in an input state:

- it is prohibited from moving the output tape head,
- it may not write to the input tape,
- it may not move the input tape head left, and
- it moves to an output state if, and only if, it moves the input tape head left.

We now formalise these conditions. Let (s, i, j, k, I, T, O) be a configuration, 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 $s \in S_O$, 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 $s' \in S_I$.

Similarly, 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 $s' \in S_I$, and if $d_1 = \rightarrow$ then $s' \in S_O$.

2.4.3. The synthesis problem for online Turing machines

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 tape, and the outputs made by the machine on the output tape. 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 *realisable* 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 realisable.

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), where every node has exactly Σ_I many children, and every finite path from the root appears exactly once. We say that $\mathcal{M}$ is a model of ϕ if $T, l \models \phi$.

2.4.4. Small and fast models

In this paper, we are concerned with online Turing machines that are either *small* or *small and fast*. We now formally define these two terms. For every online Turing machine $\mathcal{M}$, we define $|\mathcal{M}|$ to be size of the discrete control of $\mathcal{M}$ (that is, the action table) plus the length of the storage tape of $\mathcal{M}$.

Let $\mathcal{F}$ be a set of CTL formulas. We say that all formulas in $\mathcal{F}$ have *small* models if there exists a polynomial function $p : \mathbb{R} \to \mathbb{R}$ such that, for every formula $\phi \in \mathcal{F}$ there exists an online Turing machine $\mathcal{M}$ where $|\mathcal{M}|$ is at most $p(|\phi|)$.

Similarly, we say that all formulas in $\mathcal{F}$ have *small and fast* models if there exists two polynomial functions $p, q : \mathbb{R} \to \mathbb{R}$ such that, for every formula $\phi \in \mathcal{F}$ we have:

- $|\mathcal{M}|$ is at most $p(|\phi|)$.
- $\mathcal{M}$ responds to each input in at most $q(|\phi|)$ time.

2.5. The complexity class P/poly

Some of our results concern the complexity class P/poly. This is the class of problems that can be solved by a polynomial time Turing machine with and polynomially bounded *advice* function.

Formally, a P/poly algorithm consists of two parts: a polynomial-time Turing machine $\mathcal{T}$, and an advice function f. The advice function maps $\mathbb{N}$ to polynomial length advice strings. At the start of its computation, the polynomial-time Turing machine $\mathcal{T}$ is permitted to read $f(i)$, where i is the length of its input, and use the resulting advice string to aid in its computation. A problem lies in P/poly if there exists a machine $\mathcal{T}$ and advice function f that decide that problem. It is critical to note that the complexity of computing $f(i)$ is not considered: the machine $\mathcal{T}$ is given $f(i)$ for free.

2.6. Tree automata

Our results in Sections 6, 7, and 8 make heavy use of tree automata. In this section, we give the definitions that will be necessary for these results.

2.6.1. Transducers

Before we define the tree automata that we are interested in, we first define the objects that the automata will read. Our tree automata read finitely-branching infinite trees. In this paper we will represent those trees by *transducers*.

A transducer is a tuple $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l, \mathsf{start})$. The set S is a finite set of states, and the state $\mathsf{start} \in S$ is the starting state. The set Σ_I gives an input alphabet, and the set Σ_O gives an output alphabet. The transition function $\tau : S \times \Sigma_I \to S$ gives, for each state and input letter, a destination state. The function $l : S \to \Sigma_I \times \Sigma_O$ is a labelling function, which gives, for each state, an input letter, and an output letter.

As we can see, transducers are very similar to Mealy machines, but with an important difference: Mealy machines have labels assigned to their transitions, while transducers have labels assigned to their states. Moreover, each label in a transducer gives both an input and an output letter, while this is not necessary for Mealy machines, since each transition is already associated with an input letter.

Ultimately, we will consider tree automata that accept Mealy machines. The reason we must start with transducers is that results of Kupferman and Vardi [11], which form the starting point for our arguments, provide tree automata that accept transducers. We will bridge this gap in Section 5.4, where we show how to convert a tree automaton that accepts transducers into a tree automaton that accepts Mealy machines.

Finally, we can define whether a transducer $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \delta, l, \mathsf{start}, \mathsf{input})$ is a model of a CTL formula ϕ. Let T be the infinite unravelling of $\mathcal{T}$ from the state start into an infinite tree. Furthermore, let l' be the obvious extension of the labelling function l to T. We have that $\mathcal{T}$ is a model of ϕ if, and only if, $T, l \models \phi$. Similarly, $\mathcal{T}$ is a model of an LTL formula ϕ if, and only if, for every infinite path σ in $\mathcal{T}$, we have that $\sigma \models \phi$.

2.6.2. Tree automata

A tree automaton consists of a finite set of states S, an input alphabet Σ_I, an output alphabet Σ_O, an initial state $\mathsf{start} \in S$, an initial input letter input, and a transition function δ.

Before we define the format of the transition function, it is worth considering how a tree automaton reads a transducer. In each step, the automaton reads the output of the transducer, and then branches by creating copies of itself. Each branch can be represented by a pair (q, σ), where $q \in S$ is a state of the automaton, and σ is an input letter for the transducer. The meaning of the pair (q, σ) is that the automaton should create a copy of itself, where σ is sent as an input letter to the transducer, and the automaton moves to the state q. Each transition of the automaton will have many such pairs, and so the tree automaton will create many copies of itself.

Thus, the transition function of the tree automaton will specify the list of pairs that the automaton must branch into. However, the method that we use to give these pairs changes depending on the type of tree automaton that we want to define. For a *deterministic* tree automaton, it is sufficient to give a set of pairs $(q_0, \sigma_0), (q_1, \sigma_1), \ldots (q_k, \sigma_k)$, such that every pair has a distinct input letter, that is, if $\sigma_i \neq \sigma_j$ whenever $i \neq j$. This set can be given as a boolean formula: we can write it as $(q_0, \sigma_0) \wedge (q_1, \sigma_1) \wedge \cdots \wedge (q_k, \sigma_k)$, which just specifies that the automaton must pick all of the branches in the formula. Although this representation does not seem to be useful for deterministic automata, it will become useful for representing non-deterministic, universal, and eventually alternating tree automata.

A *non-deterministic* tree automaton can be seen in the same way as a non-deterministic word automaton: we give the automaton choices about which branches to make. For the sake of example, suppose that we only have two input letters σ_1 and σ_2. We could write down the following boolean formula to specify a non-deterministic transition: $(q_1, \sigma_1) \wedge (q_2, \sigma_2) \vee (q_3, \sigma_1) \wedge (q_4, \sigma_2)$. It can be seen that this transition simply offers the automaton the choice between two deterministic transitions. Either the automaton will branch according to $(q_1, \sigma_1) \wedge (q_2, \sigma_2)$, or the automaton will branch according to $(q_3, \sigma_1) \wedge (q_4, \sigma_2)$. In general, then, a non-deterministic transition is a formula that is a disjunction between a set of deterministic transition formulas.

A *universal* tree automaton does not allow non-deterministic choice, but it does allow multiple branches to have the same input letter. An example of a universal transition would be $(q_1, \sigma_1) \wedge (q_2, \sigma_1) \wedge (q_3, \sigma_2)$. Note that this formula does not specify a deterministic transition function, because we have two branches that use the input letter σ_1. In general, a universal transition is a formula that is a conjunction between branches. Unlike deterministic transition formulas, we do not restrict the set of branches that may appear in the conjunction.

Finally, we arrive at *alternating* tree automata. These automata mix non-deterministic and universal transitions. That is, an alternating transition is an unrestricted formula over branches constructed with $\wedge$ and $\vee$. We now formally define the transition function δ for alternating tree automata. A positive boolean formula is a formula constructed using only a set of atoms A, `true`, `false`, $\wedge$ and $\vee$. Let $\mathbb{B}^+(S \times \Sigma_I)$ denote the set of positive boolean formulas over the set of atoms $S \times \Sigma_I$. The transition function has the format $\delta : S \times (\Sigma_I \times \Sigma_O) \to \mathbb{B}^+(S \times \Sigma_I)$. Note that the input to δ takes a state $q \in S$, and a pair (σ_I, σ_O) of input and output letters from $\Sigma_I \times \Sigma_O$. Recall that each transition of the tree automaton reads the output from the transducer. The labelling function of the transducer is defined so that each state outputs an element of $\Sigma_I \times \Sigma_O$. Therefore, our tree automaton must read an element from $\Sigma_I \times \Sigma_O$.

Note that, in both universal and deterministic tree automata, the transition function δ gives formulas that only use $\wedge$, and do not use $\vee$. Thus, we can view these transitions as subsets of $S \times \Sigma_I$, and we will use the notation $(s', \sigma_I') \in \delta(s, (\sigma_I, \sigma_O))$ as a shorthand to denote that (s', σ_I') appears in the formula specified by $\delta(s, \sigma_I)$. Furthermore, for deterministic tree automata, we can view the transition as a partial function, which maps input letters to states. In this case, we will use $\delta(s, (\sigma_I, \sigma_O))(\sigma_I') = s'$ as a shorthand, to indicate that $\delta(s, (\sigma_I, \sigma_O))$ branches to s' on input σ_I'.

2.6.3. Acceptance conditions

We now define a *run* of a tree automaton $\mathcal{A} = (S_\mathcal{A}, \Sigma_I, \Sigma_O, \mathsf{start}_\mathcal{A}, \delta_\mathcal{A}, \mathsf{input})$ on a transducer $\mathcal{T} = (S_\mathcal{T}, \Sigma_I, \Sigma_O, \tau, l, \mathsf{start}_\mathcal{T})$. A run is represented by a finite graph called a *run graph*. A run graph R is a directed graph (V, E), where $V \subseteq S_\mathcal{A} \times S_\mathcal{T}$, which satisfies the following conditions:

- The vertex $(\mathsf{start}_\mathcal{A}, \mathsf{start}_\mathcal{T})$ is contained in the run graph.
- For every node (q, t) in V, there is a set $A = \{(q_1, \sigma_1), (q_2, \sigma_2), \ldots, (q_k, \sigma_k)\}$ such that:
  - A is a satisfying assignment of $\delta(q, l(t))$.
  - For each i in the range $1 \leq i \leq k$, we have that there is an edge from (q, t) to $(q_i, \tau(t, \sigma_i))$.
  - There are no other edges leaving (q, t).

Note that, since we allow our formulas to contain `true`, we may have that $A = \emptyset$, and therefore the run graph may contain sinks. Also, since we allow our formulas to contain `false`, it is possible that $\delta(q, l(t))$ may be unsatisfiable. Our definitions imply that no run graph may contain a vertex (q, t) where $\delta(q, l(t))$ is unsatisfiable.

We can now define the acceptance conditions for our tree automata. In this paper, we are interested in two acceptance conditions: the Co-Büchi acceptance condition, and the safety acceptance condition. For the Co-Büchi condition, the automaton comes equipped with a set of *final* states $F \subseteq S_{\mathcal{A}}$. A run graph R is accepted if, for every infinite path $(q_0, t_0), (q_1, t_1), \ldots$ in R, there are only finitely many positions i with $q_i \in F$. A tree automaton $\mathcal{A}$ accepts a transducer $\mathcal{T}$ if, and only if, there exists an accepting run graph R of $\mathcal{A}$ on $\mathcal{T}$. We will write down Co-Büchi tree automata as tuples $(S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$, where F is the set of final states.

In word automata, a safety condition would come with a set of forbidden states F, and would require that the forbidden state is encountered during a run. In tree automata, however, we can specify a safety condition without giving a forbidden set of states, because we can use unsatisfiable transitions to give the same effect. For example, if we wish to specify that the state q is forbidden, then we can set $\delta(q, \sigma) = \texttt{false}$ for every letter $\sigma \in \Sigma_I \times \Sigma_O$. Thus, we can define the safety acceptance condition: a tree automaton $\mathcal{A}$ accepts a transducer $\mathcal{T}$ if, and only if, there exists a run graph of $\mathcal{A}$ on $\mathcal{T}$.

3. Small models imply PSPACE = EXPTIME

In this section we show that, if every satisfiable CTL formula ϕ has a small online Turing machine model, then PSPACE = EXPTIME. The proof of this result can be outlined as follows: we guess a polynomially sized online Turing machine $\mathcal{M}$, and then use model checking to verify whether $\mathcal{M}$ is a model of ϕ. Our assumption guarantees that we only need to consider polynomially sized online Turing machines. This gives a NPSPACE = PSPACE algorithm for solving the CTL synthesis problem, and the result 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 first show how an online Turing machine $\mathcal{M}$ can be unravelled to produce an equivalent Mealy machine $\mathcal{T}(\mathcal{M})$.

Lemma 5. *Let ϕ be a CTL formula, and let $\mathcal{M}$ be an online Turing machine. There exists a Mealy machine $\mathcal{T}(\mathcal{M})$, where $\mathcal{T}(\mathcal{M})$ is a model of ϕ if, and only if, $\mathcal{M}$ is a model of ϕ.*

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. In this proof, we will construct the Mealy machine $\mathcal{T}(\mathcal{M}) = (S_{\mathcal{T}}, \Sigma_I, \Sigma_O, \tau_{\mathcal{T}}, l_{\mathcal{T}}, \mathsf{start}_{\mathcal{T}}, \mathsf{input})$.

Each state of $\mathcal{T}(\mathcal{M})$ will represent a configuration of $\mathcal{M}$. To simulate an online Turing machine, we need the following information:

- The current state s of the Turing machine.
- The current symbol σ_I at the input tape head.
- The current contents of the storage tape $T = \langle T_1, T_2, \ldots, T_c \rangle$.
- The current position j of the storage tape head.

Note that we do not need to know the symbol at the output tape head, because by definition, whenever a symbol is written to the output tape, the tape head must be moved to the right. Therefore, there will always be a blank symbol at the output tape head. However, for our reduction to work, we must remember the *previous* symbol written to the output tape. This is necessary to ensure correctness in the case where $\mathcal{M}$ runs forever without reading an input symbol.

Formally, we define $S_{\mathcal{T}} = S_N \cup \{\mathsf{fail}\}$, where $S_N = S \times \Sigma_I \times \mathbb{N}_{\leq c} \times \Sigma_O \times (\Sigma_T)^c$ to be the states used in $\mathcal{T}$. The state fail is used to indicate that the online Turing machine runs forever without reading an input. The starting state of $\mathcal{T}(\mathcal{M})$ is defined to be $\mathsf{start}_{\mathcal{T}} = (\mathsf{start}, \mathsf{input}_1, 1, \emptyset, \mathsf{init})$, which corresponds to the initial configuration of $\mathcal{M}$.

We now define the transition function $\tau_{\mathcal{T}}$, and the labelling function $l_{\mathcal{T}}$. Let $z = (s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \ldots, T_c \rangle)$ be a state from the set S_N. Recall that there is always a blank symbol at the output tape head. Therefore, to find the next configuration, we must consider the transition:

$$\delta(s, \sigma_I, T_j, \emptyset) = (s', (\sigma_I', d_1), (\sigma_T', d_2), (\sigma_O', d_3)).$$

We will use this transition to classify the states $z \in S_N$. If z has $d_1 = \rightarrow$, then we say that z is an *input configuration*, because $\mathcal{M}$ reads a new input at z. Similarly, if a has $d_3 = \rightarrow$, then we say that z is an *output configuration*, because $\mathcal{M}$ writes an output at z. For technical convenience, we also consider the starting state $\mathsf{start}_{\mathcal{T}}$ to be an input state.

The transition function $\tau_{\mathcal{T}}$ gives transitions between input configurations: if z is an input configuration, then $\tau_{\mathcal{T}}(z, \sigma_I)$ gives the next input configuration z' that $\mathcal{M}$ visits after reading σ_I. Note that, while moving from z to z', $\mathcal{M}$ will see exactly one output configuration z''. The labelling function $l_{\mathcal{T}}(z, \sigma_I)$ gives the output that is produced at z''.

We now formally define $\tau_{\mathcal{T}}$. We begin with a helper function $\mathrm{Succ}(z)$, which, for each configuration z, gives the next configuration z'. Let $z = (s, \sigma_I, j, \sigma_O, T = \langle T_1, T_2, \ldots, T_c \rangle)$ be a configuration and suppose that:

$$\delta(s, \sigma_I, T_j, \emptyset) = (s', (\sigma_I', d_1), (\sigma_T', d_2), (\sigma_O', d_3)).$$

Let T' be the tape T with the jth symbol replaced with σ_T'. If $d_3 = -$ then we define $\mathrm{Succ}(z)$ to be

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

On the other hand, if $d_3 =\rightarrow$ then we define $\mathrm{Succ}(z)$ 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.

Next, we define a helper function $\mathrm{Inp}(z, \sigma'_I)$. This function gives, for every configuration $z = (s, \sigma_I, j, \sigma_O, T)$, the configuration $z' = (s, \sigma'_I, j, \sigma_O, T)$. This function just replaces the current input symbol with σ'_I, and it will be used whenever $\mathcal{M}$ moves the input tape head to the right.

For every input configuration z, and every input letter σ_I, we define $\tau_{\mathcal{T}}(z, \sigma_I)$ as follows:

- We first find $z' = \mathrm{Inp}(\mathrm{Succ}(z), \sigma_I)$. That is, we find the next state from z, and we assume that σ_I appeared on the input tape when we moved to z'.
- Let π be the path that starts at z', and follows the function Succ until another input state is reached. This path may be infinite, because $\mathcal{M}$ may never visit another input state after visiting z'.
- If π ends at a state z'', then we define $\tau(z, \sigma_I) = z''$. Furthermore, if σ_O is the output letter stored in z'', then we define $l_{\mathcal{T}}(z, \sigma_I) = \sigma_O$.
- If π is infinite, then we define $\tau_{\mathcal{T}}(a, \sigma_I) = \mathsf{fail}$. Note that π can visit at most one output state. There are two cases for determining the labelling function.
  - If π visits the output state z'', then we define $l_{\mathcal{T}}(a, \sigma_I) = \sigma_O$, where σ_O is the output produced at z''.
  - If π never visits an output state, then we define $l_{\mathcal{T}}(a, \sigma_I) = \emptyset$.

We define fail to be an absorbing state that produces no output. That is, for all input letters σ_I, we define $\tau(\mathsf{fail}, \sigma_I) = \mathsf{fail}$, and we define $l_{\mathcal{T}}(\mathsf{fail}, \sigma_I) = \emptyset$. Finally, for the sake of completeness, for every configuration z that is not an input configuration, and every input letter σ_I, we define $\tau(z, \sigma_I) = \mathsf{fail}$ and $l_{\mathcal{T}}(z, \sigma_I) = \emptyset$. Note that this definition is irrelevant, since only input configurations, and fail, can be reached from $\mathsf{start}_{\mathcal{T}}$.

We now argue that this reduction is correct. We must show that, when $\mathcal{M}$ and $\mathcal{T}(\mathcal{M})$ are both given the same sequence of inputs, that they both produce the same sequence of outputs. To prove that this is the case, we first describe the behaviour of $\mathcal{T}(\mathcal{M})$ as the sequence:

$$s_0 \sigma_I^0 \sigma_O^0 s_1 \sigma_I^1 \sigma_O^1 s_2 \sigma_I^2 \sigma_O^2 s_3 \sigma_I^3 \sigma_O^3 s_4 \ldots,$$

where

- $s_0 = \mathsf{start}_{\mathcal{T}}$ is the initial state of $\mathcal{T}(\mathcal{M})$ and
- $\sigma_O^i = l_{\mathcal{T}}(s_i, \sigma_I^i)$ and $s_{i+1} = \tau_{\mathcal{T}}(s_i, \sigma_I^i)$ hold for all $i \in \omega$.

We then describe the behaviour of $\mathcal{M}$ on $\sigma_I^0 \sigma_I^1 \sigma_I^2 \sigma_I^3 \ldots$ as the sequence

$$s'_0 \sigma_I^0 \sigma_O^{0\prime} s'_1 \sigma_I^1 \sigma_O^{1\prime} s'_2 \sigma_I^2 \sigma_O^{2\prime} s'_3 \sigma_I^3 \sigma_O^{3\prime} \ldots,$$

which is either infinite and contains no output letter $\perp$ ($\forall i \in \omega.\ \sigma_O^i \neq \perp$), or contains exactly one output letter $\perp = \sigma_O^k$ (and $\forall i < k.\ \sigma_O^i \neq \perp$). In the letter case, the length of the sequence is $3k+3$, and this $\perp$ symbol ends the sequence. For this sequence:

- $s'_0 = \mathsf{start}$ is the initial configuration of $\mathcal{M}$,
- $\sigma_O^0 \sigma_O^1 \sigma_O^2 \sigma_O^3 \ldots$ is the output sequence produced by $\mathcal{M}$ on $\sigma_I^0 \sigma_I^1 \sigma_I^2 \sigma_I^3 \ldots$, and
- s'_i is the configuration reached by $\mathcal{M}$ after reading the i-th input letter, σ_I^{i-1}.

We can now use an inductive argument to show that the behaviour of $\mathcal{M}$ is an initial sequence of the behaviour of $\mathcal{T}(\mathcal{M})$ for a given input sequence. The base case of the induction is simple: $s_0 = s'_0$ holds by definition. For the inductive step, if we have shown the claim up to some s_i (i.e., for the first $3i+1$ positions of the sequences), we observe that σ_I^i occurs in both sequences. Then we must argue that both $\mathcal{M}$ and $\mathcal{T}(\mathcal{M})$ output the same symbol. This holds by construction: if $\mathcal{M}$ outputs σ_O^i, then $l_{\mathcal{T}}(s_i, \sigma_I^i) = \sigma_O^i$. On the other hand, if $\mathcal{M}$ runs forever without outputting another symbol, then $\mathcal{T}(\mathcal{M})$ will transition to fail, and will never output any further symbols. Finally, we must argue that if s'_{i+1} is the last state visited by $\mathcal{M}$ before the next input is read, then s_{i+1} will be the state in S_N that corresponds to s'_{i+1}. Again this holds by construction, because the path π used to define $\tau(s_i, \sigma_I)$ only ends when an input state is reached. Thus, we have that $\mathcal{M}$ is a model of ϕ if and only if $\mathcal{T}(\mathcal{M})$ is a model of ϕ. □

The size of $\mathcal{T}(\mathcal{M})$ is exponential in the size of $\mathcal{M}$, because the number of storage tape configurations of $\mathcal{M}$ grows exponentially with the length of the tape. Since our goal is to model check $\mathcal{M}$ in polynomial space, with respect to $\mathcal{M}$, we should not construct $\mathcal{T}(\mathcal{M})$ directly. Instead, we will use a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using

only $O(|\mathcal{M}|)$ space. It is important to note that here, space usage measures that amount of extra space used by the Turing machine, and does not include the space used to store the output.

Lemma 6. *There is a deterministic Turing machine that outputs $\mathcal{T}(\mathcal{M})$, while using $O(|\mathcal{M}|)$ space.*

Proof. We start with $\mathcal{M} = (S, \Sigma_I, \Sigma_T, \Sigma_O, \delta, \text{start}, c, \text{init}, \text{input})$, and our task is to output the Mealy machine $\mathcal{T}(\mathcal{M}) = (S_\mathcal{T}, \Sigma_I, \Sigma_O, \tau_\mathcal{T}, l_\mathcal{T}, \text{start}_\mathcal{T}, \text{input})$ that is defined in Lemma 5.

We first argue that each state in $S_\mathcal{T}$ can be stored in $O(|\mathcal{M}|)$ space. Recall that each configuration is a tuple of the form $(s, \sigma_I, j, \sigma_O, T)$. Obviously the parameters s, j, σ_I, and σ_O, can be stored in $O(\mathcal{M})$ space. Moreover, since the description of $\mathcal{M}$ contains the initial tape 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 the set of configurations, and the state fail, we have that each state of $\mathcal{T}(\mathcal{M})$ can be stored in $O(|\mathcal{M}|)$ space.

Now we can give an algorithm that outputs $\mathcal{T}(\mathcal{M})$ in $O(|\mathcal{M}|)$ space.

- To output $S_\mathcal{T}$, we loop through each configuration z, and output it. We also output the state fail.
- We output the state $\text{start}_\mathcal{T} = (\text{start}, \text{input}, 1, \emptyset, \text{init})$.
- To output the transitions $\tau_\mathcal{T}$, and the labels $l_\mathcal{T}$, we do the following. We maintain a counter n, which is initially set so that $n = 0$. We loop through each configuration z, and each input letter σ_I, and do the following.
  - We compute $z' = \text{Inp}(\text{Succ}(z), \sigma_I)$.
  - We repeatedly iterate $z' = \text{Succ}(z')$, and we increment n each time we do so. If z' is an output state, then we remember the letter that was written to the output state.
  - If we eventually find a z' that is an input state, then we output the transition $\tau_\mathcal{T}(z, \sigma_I)$ and label $l_\mathcal{T}(z, \sigma_I)$ according to Lemma 5.
  - If we reach a point where $n > |S_\mathcal{T}|$, then we can conclude that $\mathcal{M}$ runs forever without reading the next input, and we output the appropriate values for $\tau_\mathcal{T}(z, \sigma_I)$ and label $l_\mathcal{T}(z, \sigma_I)$.

Note that, since each configuration z can be stored in $O(|\mathcal{M}|)$ space, the counter n can be stored in $\log(2^{O(|\mathcal{M}|)})$ space. All other variables can clearly be stored in $O(|\mathcal{M}|)$ space. Therefore, the total amount of space used by this algorithm is $O(|\mathcal{M}|)$. □

We now argue that $\mathcal{M}$ can be model checked in polynomial space. To do this, we use the fact that Theorem 3 gives a model checking procedure that uses polylogarithmic space. Thus, when it is applied to $\mathcal{T}(\mathcal{M})$, it will use space polynomial in $|\mathcal{M}|$. Now, using standard techniques to compose space bounded Turing machines (see [14, Proposition 8.2], for example), we can compose the deterministic Turing machine given by Lemma 6 with the model checking procedure given in Theorem 3 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 $|\mathcal{M}|$. The next theorem gives the main result of this section.

Theorem 7. *If every realisable CTL formula ϕ has an online Turing machine $\mathcal{M}$ model, where $|\mathcal{M}|$ is polynomial in ϕ, then PSPACE = EXPTIME.*

Proof. Let ϕ be a CTL formula. Our assumption implies that there is some polynomial function f such that, if ϕ has an online Turing machine model, then it has an online Turing machine model $\mathcal{M}$, with $|\mathcal{M}| \leq f(|\phi|)$.

Thus, we can solve the CTL synthesis problem with the following algorithm. We first non-deterministically guess an online Turing machine $\mathcal{M}$ with $|\mathcal{M}| \leq f(|\phi|)$. Then we model check against the input formula ϕ, using the composition of the Turing machine given by Lemma 6 and the Turing machine given by Theorem 3. 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, we have that we can model check $\mathcal{M}$ in polynomial space, with respect to $|\mathcal{M}|$. Since $|\mathcal{M}| \leq f(|\phi|)$, we have produced an algorithm for solving the CTL synthesis problem in NPSPACE.

To obtain the conclusion that PSPACE = EXPTIME, it suffices to note that NPSPACE = PSPACE, and that the CTL synthesis problem is EXPTIME-complete. □

4. Small and fast models imply EXPTIME ⊆ P/poly

In this section we show that, if all realisable CTL formulas have a polynomially sized model that responds to all inputs within polynomial time, then EXPTIME ⊆ P/poly.

An alternating Turing machine is a Turing machine that has both existential and universal states (see [1] for a formal definition). Let $\mathcal{A}_b$ be an alternating Turing machine with a tape of length b that is written in binary. Since the machine is alternating, its state space S is split into $S_\forall$, which is the set of universal states, and $S_\exists$, which is the set of existential states. The first step of our proof is to construct a CTL formula ϕ_b, such that all models of ϕ_b are forced to solve the halting

problem for $\mathcal{A}_b$. More precisely, the first input from the environment will give an initial tape configuration for $\mathcal{A}_b$, and the first output from the model must correctly determine whether $\mathcal{A}_b$ terminates from that initial tape. In the next lemma, we show that such a formula can be constructed.

Lemma 8. *For every alternating Turing machine $\mathcal{A}_b$, there exists a realisable CTL formula ϕ_b, defined over a set of propositions $\Pi = \Pi_I \cup \Pi_O$, with the following properties:*

- *The first input from the environment will encode an initial tape for $\mathcal{A}_b$.*
- *Every model of ϕ_b must, in response to this input, correctly determine whether $\mathcal{A}_b$ halts from that initial tape.*

Proof. The formula will use a set of input propositions Π_I such that $|\Pi_I| = b$. This therefore gives us enough input propositions to encode a tape configuration of $\mathcal{A}_b$. The set of output propositions Π_O will allow us to encode a configuration of $\mathcal{A}_b$. More precisely, we will have the following output propositions:

- b propositions to encode the current contents of the tape,
- $\lceil \log_2(b) \rceil$ propositions to encode the current position of the tape head, and
- $\lceil \log_2(|Q|) \rceil$ propositions to encode q, which is the current state of the machine.

Our goal is to simulate $\mathcal{A}_b$. The environment, which is responsible for providing the input in each step, will perform the following tasks in our simulation. In the first step, the environment will provide an initial tape configuration for $\mathcal{A}_b$. In each subsequent step, the environment will resolve the non-determinism, which means that it will choose the specific existential or universal successor for the current configuration. In response to these inputs, our CTL formula will require that the model should faithfully simulate $\mathcal{A}_b$. That is, it should start from the specified initial tape, and then always follow the existential and universal choices made by the environment. It is not difficult to write down a CTL formula that specifies these requirements.

In addition to the above requirements, we also want our model to predict whether $\mathcal{A}_b$ halts. To achieve this we add the following output propositions:

- Let $C = |S| \cdot 2^b \cdot b$ be the total number of configurations that $\mathcal{A}_b$ can be in. If the machine halts, then it halts within C steps. We add $\log(C+2)$ propositions to encode a counter for the number of simulated steps. This counter is not a ring counter, but will stop to count once it reaches its maximum $C+2$.
- We add a proposition h, and we will require that h correctly predicts whether $\mathcal{A}_b$ halts from the current configuration.

The counter can easily be enforced by a CTL formula. To implement h, we will add the following constraints to our CTL formula:

- If the counter has reached its maximum value, then h must be false.
- Else, if s is an accepting state, then h must be true.
- Else, if s is an existential state, then $h \leftrightarrow EXh$.
- If s is a universal state, then $h \leftrightarrow AXh$.

These conditions ensure that, prior to acceptance or reaching the maximal counter value, whenever the machine is in an existential state, there must be at least one successor state from which $\mathcal{A}_b$ halts before the maximal counter value is reached, and whenever the machine is in a universal state, $\mathcal{A}_b$ must halt from all successors before the maximal counter value is reached. This completes the specification of ϕ_b.

We now argue that ϕ_b has the claimed properties. Namely, that it is realisable, and that any model of ϕ must, in its first output, correctly predict whether $\mathcal{A}_b$ halts on the given input tape. We begin by showing the later property. This can be proved by induction on the number of steps i left before either $\mathcal{A}_b$ reaches an accepting state, or the counter reaches its maximum. In the base case, where $i = 0$, the formula ϕ_b directly enforces that h should correctly determine whether $\mathcal{A}_b$ halts.

For the inductive step, suppose that h correctly determines whether $\mathcal{A}_b$ halts when there are at most i steps remaining. Now suppose that the machine is at a state s where there are at most $i+1$ steps remaining. If s is an existential state, then the formula $h \leftrightarrow EXh$ ensures that h will be true if and only if the existential non-determinism can be resolved so that the machine will halt in at most i steps. If s is a universal state, then the formula $h \leftrightarrow AXh$ ensures that h will be true if and only if all possible ways of resolving the universal non-determinism lead to the machine halting in at most i steps. This completes the proof of the inductive step. Now, having shown that the property holds for all i, we observe that in the initial state, the machine is at most C steps away from either terminating or reaching the maximum counter value. Therefore, for any model of ϕ_b, the value of h in the first output must correctly determine whether $\mathcal{A}_b$ halts.

Finally, we must argue that ϕ_b is realisable. If we ignore the proposition h, then ϕ_b is obviously realisable, since it only requires that a deterministic step of a Turing machine, which is chosen by the environment, is executed correctly, and that the counter is updated correctly. It is easy to build a model that satisfies these properties. If we now add back in the

proposition h, then the formula remains realisable: since $\mathcal{A}_b$ is a space bounded machine, we can simply determine from each state whether $\mathcal{A}_b$ will halt or not and output this information in h. □

Now, if every realisable CTL formula has an online Turing machine model that is both small and fast, then ϕ_b must have an online Turing machine model $\mathcal{M}$ that is both small and fast. In the next lemma, we show how the existence of $\mathcal{M}$ allows us to construct a polynomial time algorithm for the halting problem of $\mathcal{A}_b$.

Lemma 9. *If every CTL formula has an online Turing machine model that is small and fast, then there is a polynomial-size polynomial-time Turing machine that decides the halting problem for $\mathcal{A}_b$.*

Proof. Let $\mathcal{M}$ be a model of ϕ_b that is both small and fast. Suppose that we want to decide whether $\mathcal{A}_b$ halts on the input word I. The polynomial-size polynomial-time Turing machine $\mathcal{T}$ does the following:

- It begins by giving I to $\mathcal{M}$ as the first input letter.
- It then proceeds by simulating $\mathcal{M}$ until the first output letter is produced.
- Finally, it reads the value of h from the output letter, and then outputs it as the answer to the halting problem for $\mathcal{A}_b$.

By Lemma 8, we know that h correctly predicts whether $\mathcal{A}_b$ halts on I. Therefore, this algorithm is correct. Since $\mathcal{M}$ is both small and fast, we have that $\mathcal{T}$ is a polynomial time Turing machine. □

We now use Lemma 9 to prove the main result of this section: if every CTL formula has a small and fast model, then EXPTIME ⊆ P/poly. We will do this by showing that there is a P/poly algorithm for solving an EXPTIME-hard problem.

We begin by defining our EXPTIME-hard problem. We define HALT-IN-SPACE as follows. Let $\mathcal{U}$ be a universal[1] alternating Turing machine. We assume that $\mathcal{U}$ uses space polynomial in the amount of space used by the machine that is simulated. The inputs to our problem are:

- An input word I for $\mathcal{U}$.
- A sequence of blank symbols B, where $|B|$ is polynomial in $|I|$.

Given these inputs, HALT-IN-SPACE requires us to decide whether $\mathcal{U}$ halts when it is restricted to use a tape of size $|I| + |B|$. Since B can only ever add a polynomial amount of extra space, it is apparent that this problem is APSPACE-hard, and therefore EXPTIME-hard.

Lemma 10. *HALT-IN-SPACE is EXPTIME-hard.*

Proof. We prove this fact by reduction from the halting problem for an alternating polynomial space Turing machine, which we will denote as APSPACE-HALT. This is an APSPACE-complete problem. There are two inputs to APSPACE-HALT:

- a polynomially-space bounded alternating Turing machine $\mathcal{T}$, and
- an input word $I_{\mathcal{T}}$ for $\mathcal{T}$.

Since $\mathcal{T}$ is polynomially space bounded, we can assume that it comes equipped with a function $s : \mathbb{N} \to \mathbb{N}$, where $s(i)$ gives the total amount of space used for an input of length i. Since computing s can be done by evaluating a polynomial, we know that $s(i)$ can be computed in polynomial time.

We now provide a polynomial time reduction to HALT-IN-SPACE. Since $\mathcal{U}$ is a universal Turing machine, there must exist an input I for $\mathcal{U}$ such that $\mathcal{U}$ accepts I if and only if $\mathcal{T}$ accepts $I_{\mathcal{T}}$. We fix I for the rest of the proof. Since $\mathcal{U}$ uses polynomially more space than the machine that it simulates, we can assume that it comes equipped with a function $t : \mathbb{N} \to \mathbb{N}$, where $t(i)$ gives the amount of space used by $\mathcal{U}$ while simulating a machine that uses i space. Again, since computing $t(i)$ can be done by evaluating a polynomial, we have that $t(i)$ can be computed in polynomial time. Therefore, to complete our reduction, we construct B to be a sequence of blanks of length $t(s(|I|))$, and then solve HALT-IN-SPACE for I and B. □

The final step is to use the polynomial time Turing machine from Lemma 9 to construct a P/poly algorithm for solving HALT-IN-SPACE. This then shows that an EXPTIME-hard problem lies in P/poly, which gives the main result of this section.

Theorem 11. *If every realisable CTL formula has a polynomially sized online Turing machine model that responds to all inputs in polynomial time, then EXPTIME ⊆ P/poly.*

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

Proof. We provide a P/poly algorithm for HALT-IN-SPACE. Recall that a P/poly algorithm consists of a polynomial time Turing machine $\mathcal{T}$ that is equipped with an advice function f. We begin by defining the advice function. Let $\mathcal{U}_b$ be the machine $\mathcal{U}$, when it is restricted to only use the first b symbols on its tape. By Lemma 9, there exists a polynomial-size polynomial-time deterministic Turing machine $\mathcal{T}_b$ that solves the halting problem for $\mathcal{U}_b$. We define the advice function f so that, for each integer b, we have that $f(b)$ gives the description of $\mathcal{T}_b$. Since $\mathcal{T}_b$ can be described in polynomial space, the advice function f gives polynomial size advice strings.

The second step is to give a polynomial-time algorithm that uses the advice string from f to solve HALT-IN-SPACE. Let I and B be an instance of HALT-IN-SPACE. The algorithm begins by obtaining $\mathcal{T}_{i+b} = f(|I| + |B|)$ from the advice function. It then simulates $\mathcal{T}_{i+b}$ on the input word I, and outputs the answer computed by $\mathcal{T}_{i+b}$. By construction, this algorithm takes polynomial time, and correctly solves HALT-IN-SPACE. Therefore, we have shown that an EXPTIME-hard problem lies in P/poly. □

5. Tree automata

Our results in Sections 6, 7, and 8 depend on tree automata. In particular, Kupferman and Vardi provide a translation from each CTL formula to an alternating Co-Büchi tree automaton [11]. For our results, we need to turn this automaton into a deterministic safety tree automaton. In this section, we give a satisfiability preserving reduction for this task.

5.1. Translating CTL formulas to alternating tree automata

The reason we are interested in tree automata is that Kupferman and Vardi have shown that every CTL formula ϕ can be translated to an alternating tree automaton $\mathcal{A}(\phi)$ [11]. In this section, we give a brief overview of their construction, because some of our results depend on it. The construction that we give here is a simplification of theirs, because their construction is equipped to handle CTL synthesis with imperfect information, which is not necessary for our results.

They consider CTL formulas that are given in a certain positive normal form, where only atoms can be negated. In particular, they rewrite their CTL formulas into the following grammar:

$$\phi ::= p \mid \neg p \mid \phi \vee \phi \mid \phi \wedge \phi \mid A\psi \mid E\psi,$$
$$\psi ::= X\phi \mid \phi U \phi \mid \phi R \phi \mid F\phi \mid G\phi,$$

where p is an atomic proposition. Then, they define the alternating Co-Büchi tree automaton $\mathcal{A}(\phi) = (S, \Sigma_I, \Sigma_O, \mathsf{start}, \delta, F, \mathsf{input})$ as follows.

- They set $S = s_0 \cup \mathrm{cl}(\phi)$, where $\mathrm{cl}(\phi)$ denotes the set of sub-formulas in ϕ.
- They set $\Sigma_I = 2^{\Pi_I}$ and $\Sigma_O = 2^{\Pi_I \cup \Pi_O}$, where Π_I is the set of input propositions of ϕ, and Π_O is the set of output propositions of ϕ.
- They set $\mathsf{start} = s_0$.
- The transition function δ is then defined as follows.
  - For every atomic proposition p, and every output letter $\sigma_O \in \Sigma_O$ they set:

$$\delta(p, \sigma_O) = \begin{cases} \mathtt{true} & \text{if } p \in \sigma_O, \\ \mathtt{false} & \text{if } p \notin \sigma_O, \end{cases} \quad \delta(\neg p, \sigma_O) = \begin{cases} \mathtt{false} & \text{if } p \in \sigma_O, \\ \mathtt{true} & \text{if } p \notin \sigma_O. \end{cases}$$

  - $\delta(\phi_1 \vee \phi_2, \sigma_O) = \delta(\phi_1, \sigma_O) \vee \delta(\phi_2, \sigma_O)$.
  - $\delta(\phi_1 \wedge \phi_2, \sigma_O) = \delta(\phi_1, \sigma_O) \wedge \delta(\phi_2, \sigma_O)$.
  - $\delta(EX\phi_1, \sigma_O) = \bigvee_{\sigma_I \in \Sigma_I} (\phi_1, \sigma_I)$.
  - $\delta(AX\phi_1, \sigma_O) = \bigwedge_{\sigma_I \in \Sigma_I} (\phi_1, \sigma_I)$.
  - $\delta(E\phi_1 U \phi_2, \sigma_O) = \delta(\phi_2, \sigma_O) \vee \big(\delta(\phi_1, \sigma_O) \wedge \bigvee_{\sigma_I \in \Sigma_I} (E\phi_1 U \phi_2, \sigma_I)\big)$.
  - $\delta(E\phi_1 R \phi_2, \sigma_O) = \delta(\phi_2, \sigma_O) \wedge \big(\delta(\phi_1, \sigma_O) \vee \bigvee_{\sigma_I \in \Sigma_I} (E\phi_1 R \phi_2, \sigma_I)\big)$.
  - $\delta(A\phi_1 U \phi_2, \sigma_O) = \delta(\phi_2, \sigma_O) \vee \big(\delta(\phi_1, \sigma_O) \wedge \bigwedge_{\sigma_I \in \Sigma_I} (A\phi_1 U \phi_2, \sigma_I)\big)$.
  - $\delta(A\phi_1 R \phi_2, \sigma_O) = \delta(\phi_2, \sigma_O) \wedge \big(\delta(\phi_1, \sigma_O) \vee \bigwedge_{\sigma_I \in \Sigma_I} (A\phi_1 R \phi_2, \sigma_I)\big)$.
  - $\delta(EG\phi_1, \sigma_I) = \delta(\phi_1, \sigma_I) \wedge \bigvee_{\sigma_I \in \Sigma_I} (EG\phi_1)$.
  - $\delta(EF\phi_1, \sigma_I) = \delta(\phi_1, \sigma_I) \vee \bigvee_{\sigma_I \in \Sigma_I} (EF\phi_1)$.
  - $\delta(AG\phi_1, \sigma_I) = \delta(\phi_1, \sigma_I) \wedge \bigwedge_{\sigma_I \in \Sigma_I} (AG\phi_1)$.
  - $\delta(AF\phi_1, \sigma_I) = \delta(\phi_1, \sigma_I) \vee \bigwedge_{\sigma_I \in \Sigma_I} (AF\phi_1)$.
- Their set of final states F contains all sub-formulas of the form $E\phi R\phi$, $A\phi R\phi$, $EG\phi$, and $AG\phi$.

Our results will depend on the fact that δ can be written down in $O(|\phi|^2)$ bits. At first glance, this may not appear to be true, because the formulas used by δ use operations of the form $\bigwedge_{\sigma_I \in \Sigma_I} (\phi', \sigma_I)$ and $\bigvee_{\sigma_I \in \Sigma_I} (\phi', \sigma_I)$, for some $\phi' \in \mathrm{cl}(\phi)$. Since $|\Sigma_I|$ and $|\Sigma_O|$ are exponential in ϕ, a naive approach would not allow us to represent δ in $O(|\phi|^2)$ space. However, if we assign two special symbols to $\bigwedge_{\sigma_I \in \Sigma_I}$ and $\bigvee_{\sigma_I \in \Sigma_I}$, then we can indeed represent δ in $O(|\phi|^2)$ many bits. Another

important property is that, for each state $s \in S$, and output letter $\sigma_O \in \Sigma_O$, the transition formula $\delta(s, \sigma_O)$ can be written down in $O(|\phi|)$ space.

Lemma 12. *(See [11, Theorem 4.7].) For every CTL formula ϕ there is an alternating Co-Büchi tree automaton $\mathcal{A}(\phi) = (S, \Sigma_I, \Sigma_O, \text{start}, \delta, F, \text{input})$ such that:*

- *A transducer $\mathcal{T} \in \mathcal{L}(\mathcal{A}(\phi))$ if and only if $\mathcal{T}$ is a model of ϕ.*
- *S contains $O(|\phi|)$ many states.*
- *δ can be represented in $O(|\phi|^2)$ space.*
- *Each transition formula $\delta(s, \sigma_O)$ can be represented in $O(|\phi|)$ space.*
- *The starting state* start *can be computed in $O(|\phi|)$ time.*

5.2. Translating from alternating to universal Co-Büchi tree automaton

In this section, we show how the alternating Co-Büchi tree automaton $\mathcal{A}(\phi) = (S, \Sigma_I, \Sigma_O, \text{start}, \delta, F, \text{input})$, from the previous section, can be translated into a universal Co-Büchi tree automaton $\mathcal{U}(\phi) = (S, \Sigma_I, \Sigma'_O, \text{start}, \delta', F, \text{input})$.

We begin by describing the construction. Let $\mathcal{T} \in \mathcal{L}(\mathcal{A}(\phi))$ be a transducer that is accepted by $\mathcal{A}(\phi)$. Since $\mathcal{T}$ is accepted by $\mathcal{A}(\phi)$, we know that there must exist a run graph $R = (V, E)$ of $\mathcal{A}(\phi)$ on $\mathcal{T}$. Note that, by the definition of a run graph, for each state $(q, t) \in V$ we use exactly one satisfying assignment of $\delta(q, l(t))$ to generate the outgoing edges from (q, t). The key idea behind this proof is to annotate the transducer with this satisfying assignment. We will expand the output alphabet of $\mathcal{T}$ so that each state (q, t) also outputs an assignment for $\delta(q, l(t))$. The universal Co-Büchi tree automaton will then check whether this assignment is a satisfying assignment of $\delta(q, l(t))$, and, if it is, then the automaton branches according to that assignment.

Formally, we define the extended alphabet $\Sigma'_O := \Sigma_O \times (S \to 2^{S \times \Sigma_I})$. Thus, each output letter of the transducer contains an actual output letter $\sigma_O \in \Sigma_O$ of $\mathcal{A}(\phi)$, along with $|S|$ assignments. We now describe the transition function δ'. Let $q \in S$ be a state of $\mathcal{A}(\phi)$, let $\sigma_I \in \Sigma_I$ be an input letter, and let $\sigma'_O \in \Sigma'_O$ be an output letter, which consists of the actual output $\sigma_O \in \Sigma_O$, and a list of satisfying assignments $\gamma : S \to 2^{S \times \Sigma_I}$. We define $\delta'(q, (\sigma_I, \sigma'_O))$ as follows:

- If $\gamma(q)$ is a satisfying assignment of $\delta(q, (\sigma_I, \sigma_O))$, then we set:

$$\delta'(q, \sigma'_O) = \bigwedge_{(q', \sigma'_I) \in \gamma(q)} (q', \sigma'_I).$$

- Otherwise, we set $\delta'(q, (\sigma_I, \sigma'_O)) = \texttt{false}$.

We now consider the space requirements of $\mathcal{U}(\phi)$. Our first concern is that each letter in Σ'_O should be representable in polynomial space. At first, this may not seem to be true, since subsets of $2^{S \times \Sigma_I}$ may require exponential space to write down. However, we can get around this by again considering the special format of the formulas used in the transition functions. Recall from Section 5.1 that the formulas in δ only ever quantify over all inputs using the operators $\bigwedge_{\sigma_I \in \Sigma_I}(q, \sigma_I)$ and $\bigvee_{\sigma_I \in \Sigma_I}(q, \sigma_I)$. In the first case, any satisfying assignment must include a pair (ϕ', σ_I) for every $\sigma_I \in \Sigma_I$. Thus we can represent this assignment by adding, for each state q, a special symbol $(q, *)$, which represents the set $\{(q, \sigma_I) : \sigma_I \in \Sigma_I\}$. In the second case, we can satisfy $\bigvee_{\sigma_I \in \Sigma_I} \cdots$ with a single pair (q, σ_I). Therefore, since Lemma 16 implies that each transition formula $\delta(q, \sigma_O)$ can be written down in $O(|\phi|)$ space, we argue that we can always write down our satisfying assignment of $\delta(q, \sigma_O)$ in $O(|\phi|)$ space. Since, by assumption, each letter in Σ_O can be represented in $O(|\phi|)$ space, we now have that each letter in Σ'_O can be written down in $O(|\phi|) + O(|\phi|) = O(|\phi|)$ space. The space required to store δ' does not increase, because δ' can be constructed from δ by a simple polynomial time algorithm.

Finally, we introduce notation that allows us to deal with the new, extended, output alphabet. Let $\mathcal{T} = (S, \Sigma_I, \Sigma_O, \tau, l_{\mathcal{T}}, \text{start}, \text{input})$ be a transducer in $\mathcal{L}(\mathcal{U}(\phi))$, and let Π_O be the set of output propositions for ϕ. We define $\mathcal{T} \restriction 2^{\Pi_O}$ to be the modification of $\mathcal{T}$ that only produces the outputs in 2^{Π_O}. Formally, $\mathcal{T} \restriction 2^{\Pi_O}$ has a modified labelling function l', which is defined as follows. Let $s \in S$ be a state, and $\sigma_I \in \Sigma_I$ be an input letter. If $l(s, \sigma_I) = (\sigma_O, \gamma)$, where $\sigma_O \in 2^{\Pi_O}$ is the actual output, and γ is the list of satisfying assignments, then we define $l'(s, \sigma_I) = \sigma_O$, which simply omits γ from the output. We also generalise this notation, in the obvious way, to perform the same operations on Mealy machines.

Thus, we can use Lemma 16, along with the arguments that we have made in this subsection, to prove the following lemma.

Lemma 13. *For every CTL formula ϕ there is a universal Co-Büchi tree automaton $\mathcal{U}(\phi) = (S, \Sigma_I, \Sigma'_O, \text{start}', \delta', F, \text{input})$ such that:*

- *For every $\mathcal{T} \in \mathcal{L}(\mathcal{U}(\phi))$, we have that $\mathcal{T} \restriction 2^{\Pi_O}$ is a model of ϕ.*
- *For every model $\mathcal{T}'$ of ϕ there is a $\mathcal{T} \in \mathcal{L}(\mathcal{U}(\phi))$ with $\mathcal{T} \restriction 2^{\Pi_O} = \mathcal{T}'$.*
- *S contains $O(|\phi|)$ many states.*

- *δ' can be represented in $O(|\phi|^2)$ space.*
- *Each transition formula $\delta(s, \sigma_O)$ can be represented in $O(|\phi|)$ space.*
- *The starting state* start′ *can be computed in $O(|\phi|)$ time.*
- *We can check whether a state $s \in F$ in $O(|\phi|)$ time.*
- *Each letter in Σ_I can be stored in $O(|\phi|)$ space.*
- *Each letter in Σ'_O can be stored in $O(|\phi|)$ space.*

5.3. Translating to a deterministic safety automaton

In the last of our reductions, we will translate the universal Co-Büchi tree automaton $\mathcal{U}(\phi) = (S, \Sigma_I, \Sigma_O, \text{start}, \delta, F, \text{input})$ to a deterministic safety tree automaton $\mathcal{F}(\phi) = (S', \Sigma_I, \Sigma_O, \text{start}', \delta', \text{input})$. To do this, we use the translation given by Schewe and Finkbeiner [17]. In this subsection, we give a summary of their construction, because our results depend on the precise size of the automaton that is produced.

The idea behind the construction is to track a run of $\mathcal{U}(\phi)$ by an *annotation function*, which counts the largest number of times that a final state has been seen by $\mathcal{U}(\phi)$. The construction is as follows:

- The set $S' = S \to \{\_\} \cup \mathbb{N}^{\leq c}$, where $c \in |\phi|^{O(|\phi|)}$. That is, each state in $\mathcal{F}(\phi)$ is an annotation function, which, for each state in $\mathcal{U}(\phi)$, either assigns a natural number that is at most c, or a blank sign.
- The state $\text{start}' = \gamma$, where $\gamma(\text{start}) = 0$, and $\gamma(s) = \_$, for every state $s \in S$ with $s \neq \text{start}$.
- Let $\gamma \in S'$ be a state of $\mathcal{F}(\phi)$, let $\sigma_O \in \Sigma_O$, and let $\sigma_I \in \Sigma_I$. The transition function $\delta'(\gamma, (\sigma_I, \sigma_O))$ is defined as follows.
  - We first define an auxiliary function, which, for each state, gives the set of possible annotations for that state. For each state s and input letter σ'_I, we define:

$$\text{Anns}(s, \sigma'_I) = \{\gamma(s') + f(s') \;:\; (s, \sigma'_I) \in \delta(s', (\sigma_I, \sigma_O)) \text{ and } \gamma(s) \neq \_\}.$$

  - Then, we define the successor annotations $\gamma'_{\sigma'_I}$. For each state s and input letter σ'_I, we define $\gamma'_{\sigma'_I}(s) = \max \text{Anns}(s, \sigma'_I)$, where we define $\max(\emptyset) = \_$.
  - Finally, we define the transition $\delta'(\gamma, (\sigma_I, \sigma_O)) = \bigwedge_{\sigma'_I \in \Sigma_I} (\gamma_{\sigma'_I}, \sigma'_I)$

It should be noted that $\mathcal{F}(\phi)$ is not language equivalent to $\mathcal{U}(\phi)$. Instead, we have that the two automata are *emptiness* equivalent: the language of $\mathcal{F}(\phi)$ is non-empty if, and only if, the language $\mathcal{U}(\phi)$ is non-empty. Moreover, we still have that every transducer in the language of $\mathcal{F}(\phi)$ is a model of ϕ. These two properties will be sufficient for our results.

Next, it is important to note that each state of $\mathcal{F}(\phi)$ can be stored in $O(|\phi|^2 \cdot \log|\phi|)$ space. This is because each state is an annotation, which assigns each state of $\mathcal{U}(\phi)$ a number between 0 and $|\phi|^{O(|\phi|)}$. Thus, each number can be stored in $O(|\phi| \cdot \log|\phi|)$ space, and, by Lemma 13, there are $O(|\phi|)$ many states in $\mathcal{U}(\phi)$.

Next, we must consider the transition function δ'. For our proofs, we will be concerned with the complexity of computing $\delta'(\gamma, (\sigma_I, \sigma_O))(\sigma'_I)$, for some annotation γ, some input letters $\sigma_I, \sigma'_I \in \Sigma_I$, and some output letter $\sigma_O \in \Sigma_O$. That is, if we know the current annotation, the current output of the transducer, and the next input letter, we want to compute the next annotation visited by $\mathcal{F}(\phi)$. The next lemma shows that this can be computed in $O(|\phi|^2)$ time.

Lemma 14. *For each annotation γ, each $\sigma_I, \sigma'_I \in \Sigma_I$, and each $\sigma_O \in \Sigma_O$, we can determine $\delta'(\gamma, (\sigma_I, \sigma_O))(\sigma'_I)$ in $O(|\phi|^2)$ time.*

Proof. Note that, by definition, we have $\delta'(s, (\sigma_I, \sigma_O))(\sigma'_I) = \gamma'_{\sigma'_I}$. Therefore, we must consider the time and space complexity of computing the annotation $\gamma'_{\sigma'_I}$. Let s be a state of $\mathcal{U}(\phi)$. We compute $\gamma'_{\sigma'_I}$ as follows. We begin by setting $\gamma'_{\sigma'_I}(s) = \_$ for every state s in $\mathcal{U}(\phi)$.

- For each state s of $\mathcal{U}(\phi)$, obtain the formula $\psi = \delta(s', (\sigma_I, \sigma_O))$.
- Compute $m = \gamma(s) + f(s)$.
- For every state s' that appears in some literal of ψ, check if $m > \gamma'_{\sigma'_I}(s')$, or if $\gamma'_{\sigma'_I}(s') = \_$. If either condition holds, then set $\gamma'_{\sigma'_I}(s') = m$.

By definition, this procedure correctly computes $\gamma'_{\sigma'_I}$. We argue that it runs in $O(|\phi|^2)$ time. This is because, by Lemma 13, each formula in $\delta(s', (\sigma_I, \sigma_O))$ can be stored in $O(|\phi|)$ space. Moreover, the encoding used to prove this property ensures that each transition formula in $\delta(s', (\sigma_I, \sigma_O))$ can refer to $O(|\phi|)$ many literals. Thus, the algorithm takes at most $O(|\phi|^2)$ time. □

Finally, it can easily be seen that the state start′ can be computed in $O(|\phi|^2 \cdot \log|\phi|)$ time. This follows from the definition of start′, and from Lemma 13, which implies that the state start can be computed in $O(|\phi|)$ time. Thus, in this section, we have shown the following lemma.

Lemma 15. *(See [17].) For every CTL formula ϕ there is a deterministic safety tree automaton $\mathcal{F}(\phi) = (S', \Sigma_I, \Sigma_O, \mathsf{start}', \delta', \mathsf{input})$ such that:*

- *If $\mathcal{L}(\mathcal{U}(\phi))$ is not empty, then $\mathcal{L}(\mathcal{F}(\phi))$ is not empty.*
- *If $\mathcal{T}$ is a transducer in $\mathcal{L}(\mathcal{F}(\phi))$, then $\mathcal{T} \upharpoonright 2^{\Pi_O}$ is a model of ϕ.*
- *Each state in S' can be stored in $O(|\phi|^2 \cdot \log|\phi|)$ space.*
- *Each transition $\delta'(\gamma, (\sigma_I, \sigma_O))(\sigma_I')$ can be computed in $O(|\phi|^2)$ time.*
- *The state start' can be computed in $O(|\phi|^2 \cdot \log|\phi|)$ time.*
- *Each letter in Σ_I can be stored in $O(|\phi|)$ space.*
- *Each letter in Σ_O can be stored in $O(|\phi|)$ space.*

5.4. Automata that accept Mealy machines

In the final step in our chain of reductions, we will deal with the fact that our automata accept transducers rather than Mealy machines. To see why this is a problem, it is important to note that transducers are not necessarily *input preserving*. According to the definitions in Section 2.6, we determine whether a transducer $\mathcal{T}$ is a model of ϕ by only considering the labelling function of $\mathcal{T}$. Recall that this labelling function gives, for each state s, a pair of letters (σ_I, σ_O). Note that σ_I may not be the input letter that was actually given to the transducer in the last step.

For example, suppose there is one input proposition p, and no output propositions. The CTL formula $\phi = \neg p \; U \; \texttt{false}$ certainly does not have a Mealy machine model, because it requires that the input is always false, which a Mealy machine cannot do. However, ϕ does have a transducer model: consider a transducer with one state s, where $l(s) = \emptyset \times \emptyset$, and $\tau(s, \emptyset) = \tau(s, \{p\}) = s$. Here we see that, since a transducer can lie about the input that it has just received, it is possible to model more specifications.

Since we are solving the synthesis problem, we are not interested in formulas that cannot be modelled by Mealy machines. For this reason, we say that a transducer $\mathcal{T}$ is *input preserving* if for every transition $\tau(s, \sigma_I) = s'$, where $l(s') = (\sigma_I', \sigma_O')$, we have $\sigma_I = \sigma_I'$. In the remainder of this subsection, we will show how to transform the automaton $\mathcal{F}(\phi) = (S, \Sigma_I, \sigma_O, \mathsf{start}, \delta, \mathsf{input})$ into an alternating Co-Büchi tree automaton $\mathcal{F}'(\phi) = (S', \Sigma_I, \Sigma_O, \mathsf{start}', \delta', \mathsf{input})$ that only accepts input preserving transducers.

The idea behind the construction is to blow up the state space by affixing an input letter to each state: we set $S' = S \times \Sigma_I$. The state $(s, \sigma_I) \in S'$ represents that we are in the state s of $\mathcal{F}(\phi)$, and σ_I was the last input letter read by the transducer. We then modify the transition function to ensure that the transducer correctly reports σ_I as the last input letter. We also ensure that, whenever the automaton sends an input σ_I' to the transducer, the automaton moves to some state (s', σ_I').

Formally, the automaton $\mathcal{F}'(\phi)$ is constructed as follows.

- The state space $S' = S \times \Sigma_I$. That is, each state is affixed by an input letter.
- We set $\mathsf{start}' = (\mathsf{start}, \mathsf{input})$.
- Let $s' = (s, \sigma)$ be a state in S, and let $l = (\sigma_I, \sigma_O)$ be some output from a transducer.
  - If $\sigma \neq \sigma_I$, then the transducer has failed to preserve the input, and we set $\delta(s', l) = \texttt{false}$.
  - Otherwise, if $\delta(s, l) = \psi$, then we construct ψ' by changing every literal (q, σ_I') in ψ to $((q, \sigma_I'), \sigma_I')$. We then set $\delta'(s', l) = \psi'$.

It should be clear from the definition that $\mathcal{L}(\mathcal{F}'(\phi))$ is precisely the set of input preserving transducers in $\mathcal{L}(\mathcal{F}(\phi))$. Note also that $\mathcal{F}'(\phi)$ is still a deterministic tree automaton, because the formulas used in δ' are simple modifications of the formulas used in δ.

Note that, for every Mealy machine $\mathcal{M}$, we can construct a corresponding input preserving transducer $\mathcal{T}$ by replacing each transition with a state that has the appropriate label. Conversely, for every input preserving transducer $\mathcal{T}$, we can construct a corresponding Mealy machine $\mathcal{M}$ by moving the output from each state onto the incoming transitions of that state. In both translations, we have that $\mathcal{M}$ is a model of ϕ if, and only if, $\mathcal{T}$ is a model of ϕ. With these translations in mind, from now on we will assume that, whenever we have a tree automaton that accepts input preserving transducers, we will view it as an automaton that accepts Mealy machines.

Note that the construction used in this section does not change any of the claims made in Lemma 15. Since each letter in Σ_I can be stored in $O(|\phi|)$ space, we have that S' can be stored in $O(|\phi|^2 \cdot \log|\phi|)$ space. The algorithm used to compute $\delta'(\gamma, (\sigma_I, \sigma_O))(\sigma_I')$ just needs an extra initial step to check whether the input has been preserved. Thus, the only difference between the following lemma, and Lemma 15 is that $\mathcal{F}'(\phi)$ accepts Mealy machines, rather than transducers.

Lemma 16. *For every CTL formula ϕ there is a deterministic safety tree automaton $\mathcal{F}'(\phi) = (S', \Sigma_I, \Sigma_O, \mathsf{start}', \delta', \mathsf{input})$ such that:*

- *If $\mathcal{L}(\mathcal{U}(\phi))$ is not empty, then $\mathcal{L}(\mathcal{F}(\phi))$ is not empty.*
- *If $\mathcal{T}$ is a Mealy machine in $\mathcal{L}(\mathcal{F}(\phi))$, then $\mathcal{T} \upharpoonright 2^{\Pi_O}$ is a model of ϕ.*
- *Each state in S' can be stored in $O(|\phi|^2 \cdot \log|\phi|)$ space.*

- *Each transition* $\delta'(\gamma, (\sigma_I, \sigma_O))(\sigma_I')$ *can be computed in* $O(|\phi|^2)$ *time.*
- *The state* start′ *can be computed in* $O(|\phi|^2 \cdot \log|\phi|)$ *time.*
- *Each letter in* Σ_I *can be stored in* $O(|\phi|)$ *space.*
- *Each letter in* Σ_O *can be stored in* $O(|\phi|)$ *space.*

5.5. Emptiness queries for $\mathcal{F}'(\phi)$ are in EXPTIME for every starting state

Let $\mathcal{F}'(\phi)$ be the deterministic tree automaton from Lemma 16. In this subsection, we prove a key lemma that will be the foundation of the rest of our proofs: the emptiness problem for $\mathcal{F}'(\phi)$ lies in the complexity class EXPTIME for every starting state.

We define the following emptiness problem. Let s be a state of $\mathcal{F}'(\phi)$, and let $\mathcal{F}^s(\phi)$ be the modification of $\mathcal{F}'(\phi)$, where the start state is changed to s. To solve the emptiness problem for $\mathcal{F}^s(\phi)$, we must determine whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$.

The following lemma shows that the emptiness problem lies in the complexity class EXPTIME, and gives a precise time bound that we will later use.

Lemma 17. *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* $|\phi|^{O(|\phi|^2)}$ *time.*

Proof. To prove this lemma, we will construct a *safety game*. This is a game played on a finite graph between two players: the *reachability* player, whose goal is to reach a vertex from the set F of *final* vertices, and the *safety* player, whose goal is to prevent the reachability player from visiting F. Formally, a reachability game is a tuple (V, V_R, V_S, E, F), where V is a set of vertices and E is a set of edges. The sets V_R and V_S partition V into the set of states belonging to the reachability player, and safety player, respectively. The set F gives the set of final states.

Let ϕ be a CTL formula, and suppose that $\mathcal{F}(\phi) = (S, \Sigma_I, \Sigma_O, s, \delta)$. We will construct a safety game over the vertices $S \cup S \times \Sigma_I \times \Sigma_O$. The safety player will control the vertices in S, and the reachability player will control the vertices in $S \times \Sigma_I \times \Sigma_O$. At each vertex $s \in S$, the safety player will select a pair (σ_I, σ_O), and moves to (s, σ_I, σ_O). The reachability player then selects some pair $(s', \sigma_I') \in \delta(s, (\sigma_I, \sigma_O))$, and moves to s'. The final vertices will be all vertices (s, σ_I, σ_O) for which $\delta(s, (\sigma_I, \sigma_O))$ is unsatisfiable.

Formally, we construct the safety game $\mathcal{S}(\phi) = (V, V_R, V_S, E, F)$ as follows.

- We set $V_R = S$ and $V_S = S \times \Sigma_I \times \Sigma_O$. Therefore, we have $V = S \cup S \times \Sigma_I \times \Sigma_O$.
- The set E is composed of two types of edges.
  - For each vertex $s \in S$, and each pair of letters $(\sigma_I, \sigma_O) \in \Sigma_I \times \Sigma_O$, we add an edge $(s, (s, \sigma_I, \sigma_O))$ to E.
  - For each vertex (s, σ_I, σ_O), and each $(s', \sigma_I') \in \delta(s, (\sigma_I, \sigma_O))$, we add an edge $((s, \sigma_I, \sigma_O), s')$ to E.
- The set F contains all vertices (s, σ_I, σ_O) where $\delta(s, (\sigma_I, \sigma_O)) = \emptyset$.

It can be seen that the reachability player takes the role of the environment, while the safety player takes the role of a Mealy machine. These definitions ensure that $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$ if, and only if, the reachability player can win from s in $\mathcal{S}$.

To complete the proof, we must show that, for each safety game vertex $v \in V$, we can decide the winner of v in exponential time, with respect to $|\phi|$. Firstly, note that Lemma 16 ensures that $\mathcal{S}$ can be constructed in exponential time:

- We can loop through each $(s, \sigma_I, \sigma_O) \in S \times \Sigma_I \times \Sigma_O$ and output the vertices s and (s, σ_I, σ_O), and the edge $(s, (s, \sigma_I, \sigma_O))$. Since (s, σ_I, σ_O) can be stored in $O(|\phi|^2 \cdot \log|\phi|)$ space, this procedure can take at most $|\phi|^{O(|\phi|^2)}$ time.
- Then, we loop through each state $s \in S$, and each pair of letters $(\sigma_I, \sigma_O) \in \Sigma_I \times \Sigma_O$ and each letter $\sigma_I' \in \Sigma_I$, and check whether there is a vertex $s' = \delta(s, (\sigma_I, \sigma_O))(\sigma_I')$. If so, then we output the edge $((s, \sigma_I, \sigma_O), s')$. This can take at most

$$|\phi|^{O(|\phi|^2)} \cdot 2^{O(|\phi|)} \cdot 2^{O(|\phi|)} \cdot 2^{O(|\phi|)} \cdot O(|\phi|^2) = |\phi|^{O(|\phi^2|)}$$

- Finally, we loop through each vertex $v \in V$, and check whether v has any outgoing edges. If v does not have any outgoing edges, then we add v to F. This can take at most $|\phi|^{O(|\phi|^2)}$ time.

So far, we have shown how to construct a safety game $\mathcal{S}$ with $|\mathcal{S}| \in |\phi|^{O(|\phi|^2)}$. The final step of the proof is to note that a safety game with m edges can be solved in $O(m)$ time. Thus, we can decide whether $\mathcal{L}(\mathcal{F}^s(\phi)) = \emptyset$ in $|\phi|^{O(|\phi|^2)}$ time. □

6. 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 a Mealy machine model, there exists a small Online Turing machine model of ϕ.

We will use the deterministic safety tree automaton $\mathcal{F}'(\phi)$ given by Lemma 16 to construct a polynomially sized model of ϕ. In particular, we will use the fact, proved in Lemma 17, that emptiness queries for $\mathcal{F}'(\phi)$ lie in EXPTIME. The key observation is that, under the assumption that PSPACE = EXPTIME, Lemma 17 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 small online Turing machine that is a model of ϕ.

Let ϕ be a CTL formula that over the set $\Pi_I \cup \Pi_O$ of propositions. Suppose that ϕ has a model, and let $\mathcal{F}'(\phi) = (S, \Sigma_I, \Sigma_O, \text{start}, \delta, F, \text{input})$ be the safety tree automaton from Lemma 16.

Every time that $\mathcal{M}(\phi)$ reads a new input letter $\sigma_I \in 2^{\Pi_I}$ from the input tape, the machine attempts to find a *correct* output letter. Intuitively, a correct output letter is one that ensures that the formula is satisfied. Formally, we define a correct output letter as follows.

Definition 18 *(Correct output letter).* Let s be a state in $\mathcal{F}'(\phi)$ and a current input letter $\sigma_I \in \Sigma_I$. We say that an output letter $\sigma_O \in \Sigma_O$ is a correct letter at s if, for every output letter σ_I', we have $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$, where $s' = \delta(s, (\sigma_I, \sigma_O))(\sigma_I')$.

In other words, an output letter is correct if when we output it, no matter which input letter σ_I' is received, we always move to a state s' for which $\mathcal{L}(\mathcal{F}^{s'}(\phi))$ is non-empty.

We define the *correct letter* problem (CL-problem) as follows. Given a CTL formula ϕ, a state s in $\mathcal{F}'(\phi)$, an input letter σ_I, and a bit string w of length l, determine whether there is an output letter $\sigma_O \in \Sigma_O$ such that:

- the first l bits of σ_O are w, and
- σ_O is a correct output letter.

Note that, if there does exist a correct output letter σ_O for s and σ_I, then we can find it with at most $|\sigma_O|$ many CL-problem queries.

The next lemma is crucial for our proof: if we assume that PSPACE = EXPTIME, then we can construct an algorithm that solves the CL-problem in polynomial space.

Lemma 19. *The CL-problem lies in EXPTIME.*

Proof. Let $(\phi, s, \sigma_I, w, l)$ be a CL-problem instance. Our algorithm loops through all possible output letters $\sigma_O \in \Sigma_O$ that agree with w on their first l bits, and determines whether σ_O is correct for s and σ_I. We can check whether σ_O is correct using the following procedure:

- We loop through each input letter $\sigma_I' \in \Sigma_I$.
- For each one, we find the state $s' = \delta(s, (\sigma_I, \sigma_O))(\sigma_I')$.
- We then check whether $\mathcal{L}(\mathcal{F}^{s'}(\phi)) = \emptyset$.
- If $\mathcal{L}(\mathcal{F}^{s'}(\phi)) = \emptyset$ for all σ_I', then we output "yes". Otherwise, we move on to the next output letter.

It is obvious from Definition 18 that this procedure correctly decides the CL-problem for $(\phi, s, \sigma_I, w, l)$.

We now argue that the algorithm uses exponential time. This follows from Lemma 16. Each output letter σ_O can be written down in $O(|\phi|)$ space, and each input letter σ_I' can be written down in $O(|\phi|)$ space. Thus, we can loop through all possible combinations of the two in $2^{O(|\phi|)}$ time. Computing $\delta(s, (\sigma_I, \sigma_O))(\sigma_I')$ can be done in $O(|\phi|^2)$ time, and hence $2^{O(|\phi|^2)}$ space. Finally Lemma 17 implies that the language emptiness check can be performed in EXPTIME. Thus, we have constructed an EXPTIME algorithm for the CL-problem. □

Corollary 20. *If PSPACE = EXPTIME, then the CL-problem lies in PSPACE.*

We can now construct our small online Turing machine model $\mathcal{M}(\phi)$. The machine $\mathcal{M}(\phi)$ maintains a current state $s \in S$, which is initially set so that $s = \text{start}$. Lemma 16 implies that s can be stored in polynomial space, and that setting $s = \text{start}$ can be done in polynomial time, and hence polynomial space. In each step, the machine does the following to respond to the input σ_I:

1. Solve $O(|\phi|)$ many instances of the CL-problem to find a correct output letter σ_O for s. Note that we can always do this, because Lemma 16 implies that σ_O can be stored in $O(|\phi|)$ many bits.
2. Output $\sigma_O \restriction 2^{\Pi_O}$.
3. Read the next input letter σ_I'.
4. Move to the state $s' = \delta(s, (\sigma_I, \sigma_O))(\sigma_I')$, and return to Step 1.

Corollary 20 implies that this is a small online Turing machine.

The fact that a correct output letter always exists can be proved by a simple inductive argument. This argument starts with the fact that ϕ has a Mealy machine model, and therefore $\mathcal{L}(\mathcal{F}^{\text{start}}(\phi)) \neq \emptyset$, which implies that there must exist a correct output letter in the first step. Then, since we always output a correct letter, Step 4 must always move to a state s' with $\mathcal{L}(\mathcal{F}^{s'}(\phi)) \neq \emptyset$, which again implies that there must exist a correct output letter at s'.

Finally, we can argue that our online Turing machine $\mathcal{M}(\phi)$ must be a model of ϕ. This is because, since $\mathcal{M}(\phi)$ only ever gives correct outputs, we can view it as simulating some Mealy machine $\mathcal{T}$ that is contained in $\mathcal{L}(\mathcal{F}(\phi))$. Lemma 16, implies that $\mathcal{T}$ is a model of ϕ, which then implies that $\mathcal{M}(\phi)$ is a model of ϕ.

In summary, we have used the assumption that PSPACE = EXPTIME to construct $\mathcal{M}(\phi)$, which is an online Turing machine model that is both small, and a model of ϕ. Thus, we have shown the following theorem.

Theorem 21. *Let ϕ be a CTL formula that has a Mealy machine model. If PSPACE = EXPTIME then there is an online Turing machine $\mathcal{M}$ that is a model of ϕ, where $|\mathcal{M}|$ is polynomial in ϕ.*

7. P = LOGSPACE implies smaller models

In this section we show that, if we replace the assumption that PSPACE = EXPTIME with the assumption that P = LOGSPACE, then we can use the techniques from Section 6 to show a different result. More precisely, we show that if P = LOGSPACE, then every CTL formula ϕ that has a model, also has an online Turing machine model that uses $O(|\phi|^2 \cdot \log|\phi|)$ space.

Lemma 22. *If P = LOGSPACE, then the CL-problem can be solved in $O(|\phi|^2 \cdot \log|\phi|)$ space.*

Proof. We start by defining the *exponential CL-problem* (ECL-problem). The input of the ECL-problem is $(\phi, s, \sigma_I, w, l, b)$, where $(\phi, s, \sigma_I, w, l)$ is an instance of the CL-problem defined in Section 6, and where b is a sequence $|\phi|^{O(|\phi|^2)}$ blank symbols. To solve the ECL-problem $(\phi, s, \sigma_I, w, l, b)$, an algorithm must compute a solution to the CL-problem $(\phi, s, \sigma_I, w, l)$. The sole purpose of the sequence b is to act as padding to increase the input length.

We argue that the ECL-problem lies in P. To do this, we use the same algorithm that was used in the proof of Lemma 19: loop through each output symbol σ_O that agrees with w. For each one, loop through each possible input symbol σ_I', and solve an instance of the emptiness problem using the algorithm from Lemma 17. Let x be in the length of the input of the ECL-problem instance. We know that $x = |\phi|^{O(|\phi|^2)}$, so our analysis will change. Each output letter now has length $O(|\phi|) = O(\log x)$, so we can loop through all possible output letters in $2^{O(\log x)} = x^{O(1)}$ time. The same property holds for the input letters. Finally, the algorithm from Lemma 17 runs in $|\phi|^{O(|\phi|^2)} = O(x)$ time. Therefore, the ECL-problem lies in P.

Now we can use our assumption that P = LOGSPACE to conclude that, for each ECL-problem input of length $x = |\phi|^{O(|\phi|^2)}$, there must exist an algorithm that solves the problem in $\log(x)$ space. Thus, the algorithm solves the ECL-problem in

$$\log(|\phi|^{O(|\phi|^2)}) = O(|\phi|^2 \cdot \log|\phi|)$$

space.

Finally, we observe that this leads to an $O(|\phi|^2 \cdot \log|\phi|)$ space algorithm for the CL-problem. Suppose that we want to solve the CL-problem instance $(\phi, s, \sigma_I, w, l)$, which is represented by a bit string of length k. Let $\mathcal{L}$ be the $O(|\phi|^2 \cdot \log|\phi|)$ space Turing machine that solves the ECL-problem. We simulate $\mathcal{L}$, while maintaining a variable h that tracks the position of the read/write head of $\mathcal{L}$. In each step, if $h \leq k$, then we give $\mathcal{L}$ the corresponding bit from $(\phi, s, \sigma_I, w, l)$. If $h > k$, then we give $\mathcal{L}$ a blank symbol. Note that h can be represented in $\log(|\phi|^{O(|\phi|^2)})$ many bits. Therefore, we can simulate $\mathcal{L}$ in $O(|\phi|^2 \cdot \log|\phi|)$ space. □

Now, if we look at the construction of $\mathcal{M}(\phi)$ that was used in Section 6, we observe that, when we plug in our new algorithm from Lemma 22, it uses $O(|\phi|^2 \cdot \log|\phi|)$ space. This is because it uses $O(|\phi|)$ space to store the current input and output letters, and Lemma 22 implies that the CL-problem query can be solved in $O(|\phi|^2 \cdot \log|\phi|)$ space. Thus, we have shown the following theorem.

Theorem 23. *Let ϕ be a realisable CTL formula. If P = LOGSPACE then there is an online Turing machine $\mathcal{M}$ that is a model of ϕ, where $|\mathcal{M}| \in O(|\phi|^2 \cdot \log|\phi|)$.*

8. EXPTIME ⊆ P/poly implies small and fast models

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

The techniques used in this section are similar to the techniques used in Section 6. In particular, we will construct an online Turing machine that solves instances of the CL-problem. However, in this section, our assumption that EXPTIME ⊆ P/poly allows us to prove stronger results. In the next lemma we show that, if EXPTIME ⊆ P/poly, and if the CTL formula ϕ is fixed, then we have a polynomial time algorithm that can solve all CL-problems that involve ϕ for all possible partitions of input and output propositions.

Lemma 24. *If EXPTIME ⊆ P/poly, and if the CTL formula ϕ is fixed, then the CL-problem can be solved in polynomial time.*

Proof. The first step is to slightly modify the inputs to the CL-problem. Note that, if ϕ is fixed, then, by Lemma 16, all other inputs to the problem have bounded length. For each formula ϕ, let $(\phi, s, \sigma_I, l, w)$ be the input of the CL-problem that requires the longest representation. We pad the representation of all other inputs involving ϕ, so that they have the same length as $(\phi, s, \sigma_I, l, w)$. Note that, even if the inputs are padded in this way, Lemma 19 still implies that the problem lies in EXPTIME.

Since, the problem lies in EXPTIME, the assumption that EXPTIME ⊆ P/poly implies that the problem also lies in P/poly. Let $\mathcal{T}$ and f be the polynomial time Turing machine and advice function that witness the inclusion of the problem in P/poly.

Our padding ensures that, for each CTL formula ϕ, there must be a unique advice string in f that is used by $\mathcal{T}$ to solve all CL-problems involving ϕ. Hence, if we include this advice string on the storage tape of $\mathcal{T}$, then we have constructed a polynomial time Turing machine (with no advice function) that solves all instances of the CL-problem involving ϕ. □

The construction of an online Turing machine that is a model of ϕ is then the same as the one that was provided in Section 6, except that we use the polynomial algorithm from Lemma 24 to solve the CL-problem in each step. Thus, we have constructed an online Turing machine model of ϕ that responds to each input in polynomial time. Moreover, our online Turing machine still obviously uses only polynomial space. Thus, we have established the main result of this section.

Theorem 25. *Let ϕ be a CTL formula that has a model. If EXPTIME ⊆ P/poly then there is a polynomially sized online Turing machine $\mathcal{M}$ that is a model of ϕ that responds to every input after a polynomial number of steps.*

9. LTL specifications

In this section we extend our results to LTL. Our starting point is the following theorem of Kupferman and Vardi, which shows that every LTL specification ϕ can be translated into a universal Co-Büchi tree automaton $\mathcal{U}(\phi)$.

Theorem 26. *(See [12].) For every LTL formula ϕ, we can construct a universal Co-Büchi tree automaton $\mathcal{U}(\phi)$ with $2^{O(|\phi|)}$ states that accepts a Mealy machine $\mathcal{T}$ if, and only if, $\mathcal{T}$ is a model of ϕ.*

Note that this translation gives a universal Co-Büchi tree automaton that has exponentially many states in $|\phi|$. This should be contrasted with Lemma 13, where we showed that every CTL formula can be translated to a polynomially sized universal Co-Büchi tree automaton. On the positive side, since we can translate LTL formulas to universal Co-Büchi tree automata, we can mostly reuse the proofs given for CTL, while keeping in mind that we start with an exponentially sized automaton.

We start with results regarding small online Turing machines. Due to the exponential blowup incurred when translating to universal Co-Büchi tree automata, we have to change the definition of "small". An online Turing machine $\mathcal{M}$ is small with respect to an LTL formula ϕ if:

- the storage tape of $\mathcal{M}$ has length exponential in $|\phi|$, and
- the discrete control (that is, the action table) of $\mathcal{M}$ has size exponential in $|\phi|$.

We can prove the following theorem about small online Turing machine models of LTL formulas.

Theorem 27. *We have both of the following:*

1. *If every LTL formula ϕ has an online Turing machine model $\mathcal{M}$, where $|\mathcal{M}|$ is exponential in $|\phi|$, then EXPSPACE = 2EXPTIME.*
2. *If PSPACE = EXPTIME, then every LTL formula has an exponentially sized online Turing machine model.*

Proof. For the first claim, we adapt the techniques developed in Section 3. In particular, we show that the LTL synthesis problem can be solved in exponential space using the following procedure. We loop through every online Turing machine $\mathcal{M}$ where $|\mathcal{M}|$ is exponential in $|\phi|$, and for each one, we do the following: We first translate the LTL formula ϕ into the exponentially sized universal Co-Büchi tree automaton $\mathcal{U}(\phi)$. Since it known that model checking for universal Co-Büchi tree automata can be performed in $O\big((\log|\mathcal{U}| + \log|\mathcal{T}(\mathcal{M})|)^2\big)$ space [19, Theorem 3.2], we can apply the model checking algorithm to the logspace Turing machine from Lemma 6. This produces an algorithm for checking whether $\mathcal{M}$ is a model of ϕ that uses space exponential in $|\phi|$. Since we can loop through all exponentially sized online Turing machines in exponential space, we have produced an EXPSPACE algorithm for LTL synthesis, which, by Theorem 2, is a 2EXPTIME-complete problem.

For the second claim, we can reuse the proof of Theorem 21 directly. We start by translating the LTL formula ϕ into the universal Co-Büchi tree automaton $\mathcal{U}(\phi)$, and then continue onwards from Lemma 13. Note that, in this version of the proof,

every input to the CL-problem requires exponential space with respect to $|\phi|$, and that the procedure given by Lemma 19 now takes doubly-exponential time, with respect to $|\phi|$. Thus, when we apply the assumption that PSPACE = EXPTIME, we obtain an algorithm for solving the CL-problem in exponential space with respect to $|\phi|$. Therefore, at the end of the proof, we obtain an online Turing machine model of ϕ that uses exponential space with respect to $|\phi|$. □

We now turn our attention to small and fast online Turing machines. Once again, we must change the definition of "fast". Note that a Turing machine equipped with an exponentially long tape can take doubly-exponential time to terminate. Thus, we say that an online Turing machine is a small and fast with respect to an LTL formula ϕ if:

- it is small with respect to ϕ, and
- it always responds to each input in time exponential in $|\phi|$.

We have the following theorem about small and fast online Turing machines for LTL formulas.

Theorem 28. *If EXPTIME ⊆ P/poly then every LTL formula has an exponentially sized online Turing machine model, which responds to every input after an exponential number of steps.*

Proof. We reduce the LTL formula ϕ to the exponentially sized universal Co-Büchi tree automaton $\mathcal{U}(\phi)$, and then follow the proof of Theorem 25. Using the same observations that were made in Theorem 27, we can argue that we now obtain an exponentially sized model of ϕ that responds to each input in exponential time. □

However, we cannot prove the opposite direction of Theorem 28. This is because, if we attempted to emulate the proof of Theorem 11, we would obtain the following statement: If every satisfiable LTL formula has a small and fast model, then 2EXPTIME ⊆ EXPTIME/exp. In other words, 2EXPTIME would be contained in the class of problems solvable by an exponential time Turing machine equipped with a function that gives exponentially sized advice strings. This, however, is not a meaningful statement. If we have access to an exponentially sized advice string, then we can simply use the advice string for input length i to encode a lookup table that gives the solution for all inputs of length i. Hence EXPTIME/exp = ALL, the class of all problems, and therefore we trivially have 2EXPTIME ⊆ EXPTIME/exp.

9.1. Slightly stronger results

In this subsection, we observe that, if we stray away from well known complexity classes, then we can obtain slightly stronger results for LTL formulas. Let QPSPACE be the class of problems that can be solved in quasi-polynomial space. That is, QPSPACE is the class of problems that can be solved in $O(2^{(\log n)^c})$ space for some constant c. Note that QPSPACE ⊇ EXPTIME is a weaker assumption than PSPACE = EXPTIME, as PSPACE is strictly contained in QPSPACE. Thus, the following theorem is a strengthening of the second part of Theorem 27.

Theorem 29. *If QPSPACE ⊇ EXPTIME, then every LTL formula has an exponentially sized online Turing machine model.*

Proof. As in Theorem 27, we turn our LTL formula into a universal Co-Büchi tree automaton, and follow the proof of Theorem 21. Now, we know that all inputs to the CL-problem have length x, which is exponential in $|\phi|$, and we know that Lemma 19 can solve the CL-problem in exponential time with respect to x. Thus, if we apply the assumption that QPSPACE ⊇ EXPTIME, then we obtain an algorithm for solving the CL-problem in $O(2^{(\log x)^c})$ space. Since $x \in 2^{O(\phi)}$, we therefore obtain an algorithm for solving the CL-problem in $O(2^{|\phi|^k})$, for some constant k. Thus, we can construct an online Turing machine model of ϕ that uses $O(2^{|\phi|^k})$ space, which is still exponential in $|\phi|$. □

We can also provide a similar strengthening of Theorem 28. Let QP be the class of problems solvable in quasi-polynomial time. That is, QP is the class of problems that can be solved in $O(2^{(\log n)^c})$ time for some constant c. Furthermore, let QP/qpoly be the class of problems solvable by a quasi-polynomial time Turing machine equipped with a function that gives quasi-polynomial length advice strings. That is, advice strings of length $O(2^{(\log n)^c})$. Since P/poly ⊆ QP/qpoly, the following is a strengthening of Theorem 28.

Theorem 30. *If EXPTIME ⊆ QP/qpoly then every LTL formula has an exponentially sized online Turing machine model, which responds to every input after an exponential number of steps.*

Proof. This proof is very similar to the proof of Theorem 29. We again find that all inputs to the CL-problem have length x, which is exponential in $|\phi|$. The construction used in Lemma 24 then yields a Turing machine that runs in $O(2^{(\log n)^c})$ time. Using the arguments from the proof of Theorem 29, we then find that our online Turing machine model of ϕ runs in $2^{|\phi|^k}$ time, for some constant k, which is still exponential in $|\phi|$. □

10. Bounded synthesis for online Turing machines

In this section, we consider the bounded synthesis problem for online Turing machines. The problem is: given a bound $b \in \mathbb{N}$, and a CTL formula ψ, is there an online Turing machine $\mathcal{M}$ that is a model of ψ, and where $|\mathcal{M}| \leq b$? It is easy to show that this problem is in PSPACE, by adapting the results from Section 3: we can non-deterministically guess an online Turing machine $\mathcal{M}$ with $|\mathcal{M}| \leq b$, and then apply model checking to verify whether $\mathcal{M}$ is a model of ψ. In this section, we show a corresponding lower bound, by showing that the problem is PSPACE-hard.

We show the lower bound via a reduction from quantified boolean formulas (QBF). Let $T = \forall a_1 \exists e_1 \ldots \forall a_n \exists e_n \phi(a_1, e_1, \ldots, a_n, e_n)$ be a QBF instance. We will construct a CTL formula ψ^T with a single input proposition b, and with $2n$ output propositions: for each i in the range $1 \leq i \leq n$, we have a_i and e_i as output propositions.

The formula will specify a procedure that takes place over n rounds. In each round, the environment will specify a value for a_i, and the system is required to choose a value for e_i. Afterwards, in round $n+1$, we require that $\phi(a_1, e_1, \ldots, a_n, e_n)$ is satisfied by these choices.

We encode this in a CTL formula as follows. We use the standard shorthand $AG\psi = \neg E(\texttt{true} U \neg\psi)$. For each i in the range $1 \leq i \leq n$ we define:

$$P_i = (b \to AGa_i) \wedge (\neg b \to AG\neg a_i),$$
$$Q_i = AGe_i \vee AG\neg e_i,$$
$$\psi_i^T = P_i \wedge Q_i \wedge AX\psi_{i+1}^T.$$

We also define $\psi_{n+1}^T = \phi(a_1, e_1, \ldots, a_n, e_n)$, and we define $\psi^T = \psi_1^T$. Note that P_i forces the system to correctly remember the decisions made by the environment for the a_i variables, and Q_i prevents the system from changing its decision for the e_i variables. Note also that $|\psi^T|$ is polynomial in T.

It can be seen that ϕ^T has a model if, and only if, the QBF instance T is valid. If T is valid, then no matter what the environment does, the system can ensure that $\phi(e_1, a_1, \ldots, e_n, a_n)$ is satisfied in round $n+1$. Conversely, if T is not valid, then no matter what the system does, the environment can ensure that $\phi(e_1, a_1, \ldots, e_n, a_n)$ is not satisfied in round $n+1$.

We show that if ϕ^T has a model at all, then it has a small online Turing machine model. We construct the online Turing machine $\mathcal{M}_T$ as follows. The machine keeps a record of the variables that have already been set. Thus, in round i, the machine has values for e_j and a_j for all $j < i$. It then reads the input bit b from the environment, and checks whether it can set $e_i = 1$. More precisely it checks whether:

$$\forall a_{i+1} \exists e_{i+1} \ldots \forall a_n \exists e_n \phi(a_1, e_1, \ldots, a_{i-1}, e_{i-1}, b, 1, \ldots, a_n, e_n)$$

holds, where a_j and e_j are free variables whenever $j > i$. Note that, since this is a QBF instance, this check can be carried out in polynomial space. If the check passes, then the machine sets $e_i = 1$, and if the check fails then the machine sets $e_i = 0$. It then outputs the stored values for e_j and a_j for all $j \leq i$, and moves to round $i+1$.

Since the machine always checks if the choice for e_i is correct, it is not difficult to see that, if T is valid, then $\mathcal{M}_T$ is a model of ψ^T. Moreover, all operations used by $\mathcal{M}_T$ use polynomial space in $|\psi^T|$. Therefore, we set $b = |\mathcal{M}_T|$, and we construct a bounded synthesis problem with inputs ψ^T and b. Note that the size of this instance is polynomial in T. Moreover, the answer is "yes" for the bounded synthesis instance if, and only if, T is valid. Therefore, we have shown the following theorem.

Theorem 31. *The CTL bounded synthesis problem for online Turing machines is PSPACE-complete.*

It is also easy to adapt this result for LTL. For PSPACE containment, we can refer to the results from Section 9: given a bound b and an LTL formula $|\psi|$, we can guess an online Turing machine $\mathcal{M}$ with $|\mathcal{M}| \leq b$, and then apply model checking to verify whether $\mathcal{M}$ is a model of ψ. For the lower bound, note that ψ^T can also be interpreted as an LTL formula. Therefore, we can repeat the same argument for LTL formulas. Thus, we have the following theorem.

Theorem 32. *The LTL bounded synthesis problem for online Turing machines is PSPACE-complete.*

Acknowledgments

We thank the reviewers for their careful reading, and for their thorough and constructive comments.

References

[1] A.K. Chandra, D. Kozen, L.J. Stockmeyer, Alternation, J. ACM 28 (1) (1981) 114–133.
[2] A. Church, Logic, arithmetic and automata, in: Proceedings of the International Congress of Mathematicians, 15–22 August 1962, Stockholm, Institut Mittag-Leffler, Djursholm, Sweden, 1963, pp. 23–35.

[3] A. Cimatti, E.M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, A. Tacchella, NuSMV 2: an opensource tool for symbolic model checking, in: Proc. of CAV, 2002, pp. 359–364.
[4] R. Ehlers Unbeast, Symbolic bounded synthesis, in: Proc. of TACAS, 2011, pp. 272–275.
[5] J. Fearnley, D. Peled, S. Schewe, Synthesis of succinct systems, in: Proc. of ATVA, 2012, pp. 208–222.
[6] E. Filiot, N. Jin, J.-F. Raskin, An antichain algorithm for LTL realizability, in: Proc. of CAV, 2009, pp. 263–277.
[7] S. Gulwani, S. Jha, A. Tiwari, R. Venkatesan, Synthesis of loop-free programs, in: Proc. of PLDI, 2011, pp. 62–73.
[8] G.J. Holzmann, The model checker SPIN, IEEE Trans. Softw. Eng. 23 (5) (1997) 279–295.
[9] R. Karp, R. Lipton, Turing machines that take advice, Enseign. Math. 28 (2) (1982) 191–209.
[10] O. Kupferman, M.Y. Vardi, Freedom, weakness, and determinism: from linear-time to branching-time, in: Proc. of LICS, June 1995.
[11] O. Kupferman, M.Y. Vardi, Synthesis with incomplete informatio, in: Proc. of ICTL, July 1997, pp. 91–106.
[12] O. Kupferman, M.Y. Vardi, Safraless decision procedures, in: Proc. of FOCS, 2005, pp. 531–540.
[13] O. Kupferman, M.Y. Vardi, P. Wolper, An automata-theoretic approach to branching-time model checking, J. ACM 47 (2) (March 2000) 312–360.
[14] C. Papadimitriou, Computational Complexity, Addison Wesley Pub. Co., 1994.
[15] N. Piterman, A. Pnueli, Y. Sa'ar, Synthesis of reactive(1) designs, in: Proc. of VMCAI, 2006, pp. 364–380.
[16] A. Pnueli, R. Rosner, On the synthesis of a reactive module, in: Proc. of POPL, 1989, pp. 179–190.
[17] S. Schewe, B. Finkbeiner, Bounded synthesis, in: Proc. of ATVA, 2007, pp. 474–488.
[18] A. Solar-Lezama, L. Tancau, R. Bodík, S.A. Seshia, V.A. Saraswat, Combinatorial sketching for finite programs, in: Proc. of ASPLOS, 2006, pp. 404–415.
[19] M.Y. Vardi, P. Wolper, An automata-theoretic approach to automatic program verification, in: Proc. of LICS, Cambridge, UK, 1986, pp. 322–331.