Weak Muller Conditions Make Delay Games Hard

Winter, Sarah; Zimmermann, Martin

Published in:
Aspects of Computation and Automata Theory with Applications

DOI (link to publication from Publisher):
10.1142/9789811278631_0016

Publication date:
2024

Document Version
Accepted author manuscript, peer reviewed version

Weak Muller Conditions Make Delay Games Hard

Sarah Winter
Université libre de Bruxelles, Brussels, Belgium
sarah.winter@ulb.ac.be

Martin Zimmermann
Aalborg University, Aalborg, Denmark
mzi@cs.aau.dk

We show that solving delay games with winning conditions given by deterministic and non-deterministic weak Muller automata is 2EXPTIME-complete respectively 3EXPTIME-complete. Furthermore, doubly- and triply-exponential lookahead is necessary and sufficient to win such games. These results are the first that show that the succinctness of the automata types used to specify the winning conditions has an influence on the complexity of these problems.

1 Introduction

Is solving delay games with Muller conditions, i.e., determining its winner, harder than solving delay games with parity conditions? Is more lookahead required to win a delay game with a Muller condition than to win a delay game with a parity condition? Deterministic Muller automata are exponentially more succinct than deterministic parity automata [1][1] and solving classical, delay-free Muller games is PSpace-complete [14] while solving parity games is quasi-polynomial [3]. So, the answer to both questions should be yes, but surprisingly, and frustratingly, both questions are open. Here, we take a step towards resolving them by showing that the answer is indeed yes if we consider weak Muller conditions.

The games we consider here are Gale-Stewart games [9], arguably the simplest form of two-player games. In such a game, two players, called I and O, alternatingly pick letters, thereby constructing an infinite word α. The winner of such a play is determined by the winning condition of the game, a language of infinite words. Player O wins if α is in the winning condition, otherwise Player I wins.

Delay games were introduced by Hosch and Landweber [12] only three years after the seminal Büchi-Landweber Theorem [2] showing how to determine the winner of Gale-Stewart games with ω-regular winning conditions. Hosch and Landweber generalized the setting of Gale-Stewart games by allowing Player O to delay her moves to obtain a lookahead on Player I's moves. This advantage allows her to win games she cannot win without lookahead. These games capture the asynchronous interaction of two agents [6], the deterministic uniformization problem for relations over infinite words [4, 10], and are related to streaming transducers with delay [8].

In this work, we are interested in games with ω-regular winning conditions. These are typically represented by (deterministic or non-deterministic) ω-automata with various kinds of acceptance conditions,

[1] Strictly speaking this is only true when the size of automata is measured in the number of states, which is a very crude measure for Muller automata. See the discussion about the representation of Muller conditions later in the introduction and also the works of Boker [1] and Hunter and Dawar [14, 13].

Table 1: The complexity of solving delay games. Results for reachability, safety, and parity (in gray) are from [16, 17]; results for weak Muller are proven here.

	deterministic	non-deterministic
Reachability	PSPACE-complete	PSPACE-complete
Safety	EXPTIME-complete	2EXPTIME-complete
Parity	EXPTIME-complete	2EXPTIME-complete
weak Muller	2EXPTIME-complete	3EXPTIME-complete

Table 2: The lookahead required to win delay games. All bounds are asymptotically tight. Results for reachability, safety, and parity (in gray) are from [16, 17]; results for weak Muller are proven here.

	deterministic	non-deterministic
Reachability	exponential	doubly-exponential
Safety	exponential	doubly-exponential
Parity	exponential	doubly-exponential
weak Muller	doubly-exponential	triply-exponential

e.g., safety, parity or Muller. The choice between determinism and non-determinism and succinctness of the acceptance condition can have an influence on the complexity of solving the game.

Hosch and Landweber showed that it is decidable whether Player O wins a given delay game with bounded lookahead [12]. Forty years later, Holtmann, Kaiser, and Thomas [10] revisited delay games and proved that if Player O wins a delay game then she wins it already with doubly-exponential lookahead (in the size of a given deterministic parity automaton recognizing the winning condition). Thus, unbounded lookahead does not offer any advantage over doubly-exponential lookahead in games with ω-regular winning conditions. Furthermore, they showed that the winner of a delay game, again with its winning condition given by a deterministic parity automaton, can be determined in doubly-exponential time.

Both upper bounds were improved and matching lower bounds were proven by Klein and Zimmermann: Solving delay games is EXPTIME-complete and exponential lookahead is both necessary to win some games and sufficient to win all games that can be won [16]. Both lower bounds already hold for winning conditions specified by deterministic safety automata while the upper bounds hold for deterministic parity automata. They also considered reachability automata, for which there is no difference between the results for deterministic and non-deterministic automata.

For non-deterministic parity automata, solving delay games is 2EXPTIME-complete and doubly-exponential lookahead is both necessary and sufficient [17]. As before, both lower bounds already hold for non-deterministic safety automata while the upper bounds hold for non-deterministic parity automata. See Table 1 and Table 2 for an overview over the known results (in gray).

However, the parity condition is not very succinct, e.g., deterministic Rabin, Streett, and Muller automata can all be exponentially more succinct than deterministic parity automata while they all recognize exactly the ω-regular languages.

Here, we need to discuss briefly how to measure the size of an ω-automaton, say with set Q of states. For all acceptance conditions induced by a subset $F \subseteq Q$ (e.g., reachability, safety, Büchi, and co-Büchi) or by a coloring $\Omega\colon Q \to \mathbb{N}$ (e.g., parity) the size of the automaton is captured by its number of states, as the acceptance condition can be encoded with a polynomial overhead. However, for more succinct automata, the situation is different. Rabin and Streett conditions are given by a finite family $(R_i, G_i)_{i\in\mathcal{I}}$ of pairs of subsets $R_i, G_i \subseteq Q$. Here, $|\mathcal{I}|$ is called the index of the automaton and has to be taken into account when measuring its size.

Muller acceptance on the other hand is based on a set $\mathcal{F} \subseteq 2^Q$, which can be encoded in various

representations. The simplest one is just listing all subsets in $\mathcal{F}$, the so-called explicit representation whose size is $|\mathcal{F}|$. In this work we are mainly focused on representing $\mathcal{F}$ by a Boolean formula with variables in Q so that the models of the formula are exactly the sets in $\mathcal{F}$. In this case, we measure the size of the acceptance condition by the size of the formula. Other representations include the use of circuits, trees, DAGs, and colorings. The relative succinctness of these representations and their influence on the complexity of solving arena-based (i.e., delay-free) games has been studied by Hunter and Dawar [14] and Horn [11].

In the setting of arena-based games, parity games can be solved in quasi-polynomial time [3] while solving Rabin games is NP-complete [7], solving Streett games is co-NP-complete (they are dual to Rabin games), and solving Muller games can be P-complete, co-NP-complete, or PSpace-complete, depending on the representation of $\mathcal{F}$ [14, 11].

Thus, it is natural to ask whether winning conditions given by more succinct automata also make delay games harder to solve and increase the bounds on the necessary lookahead for Player O. However, no such results are known. Note that it is straightforward to obtain doubly-exponential (triply-exponential) upper bounds on the complexity and on the lookahead, as every deterministic (non-deterministic) Rabin, Street, or Muller automaton can be turned into an equivalent exponentially larger deterministic (non-deterministic) parity automaton. This result is independent of the size and encoding of the acceptance condition of the automaton that is transformed.

Here, we are interested in Muller conditions, as they are the most succinct ones. As a first step towards answering our motivating questions, we show that for *weak* Muller conditions, there is indeed an exponential increase, both in the solution complexity and in the necessary lookahead. Thus, we provide matching lower bounds to the upper bounds obtained by transforming (weak) Muller automata into parity automata.

Recall that a Muller condition $\mathcal{F} \subseteq 2^Q$ represents all those runs $\rho \in Q^\omega$ whose infinity set, the set of states visited infinitely often by ρ, is in $\mathcal{F}$. In contrast, a weak Muller condition $\mathcal{F} \subseteq 2^Q$ represents all those runs ρ whose occurrence set, the set of states visited by ρ, is in $\mathcal{F}$. Note that weak Muller automata are strictly less expressive than (standard) Muller automata.

In this paper we settle the complexity of solving delay games with deterministic and non-deterministic weak Muller winning conditions and provide tight bounds on the necessary lookahead to win such games. More precisely, we show that solving delay games with deterministic weak Muller winning conditions is 2ExpTime-complete and solving delay games with non-deterministic weak Muller winning conditions is 3ExpTime-complete. Similarly, we show that doubly-exponential lookahead is necessary and sufficient to win delay games with deterministic weak Muller conditions and triply-exponential lookahead is necessary and sufficient to win delay games with non-deterministic weak Muller conditions. All these results hold for $\mathcal{F}$ being represented by a Boolean formula. The lower bounds rely on the fact that in a weak Muller automaton, every visit to a state is relevant for acceptance. We show that this property allows to construct a small deterministic (non-deterministic) automaton that requires the players in a game to implement a cyclic counter with exponentially (doubly-exponentially) many values. This automaton is then used to construct games in which Player O needs doubly-exponential (triply-exponential) lookahead and to construct games that simulate AExpSpace (A2ExpSpace) Turing machines. Finally, we also give an exponential lower bound on the necessary lookahead in delay games with deterministic explicit weak Muller conditions.

We conclude this paper by discussing the obstacles one faces when trying to take the next step, i.e., to raise the lower bounds for (standard) Muller conditions. There, we also return to the discussion of the different representations of Muller conditions, i.e., we discuss whether our lower bounds, which hold for the representation by Boolean formulas, can be lifted to less succinct representations, thereby giving stronger results.

2 Preliminaries

We introduce our notation. We denote the non-negative integers by $\mathbb{N}$, and the set $\{0, 1\}$ by $\mathbb{B}$. The size of a set X is denoted by $|X|$.

Words, languages. An *alphabet* Σ is a non-empty finite set of *letters* or *symbols*. A *word* over Σ is a sequence of letters of Σ. The set of finite resp. infinite words over Σ is denoted by Σ^* resp. Σ^ω. The set of non-empty finite words over Σ is denoted by Σ^+. The *empty word* is denoted by ε. Given a finite or infinite word α, we denote by $\alpha(i)$ the i-th letter of α, starting with 0, i.e., $\alpha = \alpha(0)\alpha(1)\alpha(2)\cdots$. The length of α is denoted by $|\alpha|$. A subset $L \subseteq \Sigma^*$ resp. $L \subseteq \Sigma^\omega$ is a *language* resp. an *ω-language* over Σ. We drop the prefix ω when it is clear from the context.

Given two infinite words $\alpha \in (\Sigma_0)^\omega$ and $\beta \in (\Sigma_1)^\omega$, we define $\binom{\alpha}{\beta} = \binom{\alpha(0)}{\beta(0)}\binom{\alpha(1)}{\beta(1)}\binom{\alpha(2)}{\beta(2)}\cdots \in (\Sigma_0 \times \Sigma_1)^\omega$.

Automata. An *ω-automaton* is a tuple $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ where Q is a finite set of states, Σ is an alphabet, $q_I \in Q$ is an initial state, $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation, and $\mathrm{Acc} \subseteq \Delta^\omega$ is a set of accepting runs.

An infinite *run* ρ of $\mathfrak{A}$ is a sequence

$$\rho = (q_0, a_0, q_1)(q_1, a_1, q_2)(q_2, a_2, q_3)\cdots \in \Delta^\omega.$$

As usual, we say that ρ is initial if $q_0 = q_I$ and we say that ρ processes $a_0a_1a_2\cdots \in \Sigma^\omega$. Given a run ρ, let $\mathrm{Inf}(\rho)$ be the set of states visited infinitely often by ρ, and let $\mathrm{Occ}(\rho)$ be the set of states visited by ρ.

The language *recognized* by $\mathfrak{A}$ is the set $L(\mathfrak{A}) \subseteq \Sigma^\omega$ that contains all ω-words that have an initial run in Acc processing it.

An *ω-automaton* $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ is *deterministic* (and complete) if Δ is given as a total function $\delta\colon Q \times \Sigma \to Q$. If we speak of *the* run of $\mathfrak{A}$ on $\alpha \in \Sigma^\omega$, then we mean the unique initial run of $\mathfrak{A}$ processing α.

Acceptance conditions. We recall safety, reachability, parity, and (weak) Muller acceptance conditions.

An ω-automaton $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ is a *safety automaton*, if there is a set $F \subseteq Q$ of accepting states such that

$$\mathrm{Acc} = \{(q_0, a_0, q_1)(q_1, a_1, q_2)(q_2, a_2, q_3)\cdots \in \Delta^\omega \mid q_i \in F \text{ for every } i\}.$$

Moreover, an ω-automaton $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ is a *reachability automaton*, if there is a set $F \subseteq Q$ of accepting states such that

$$\mathrm{Acc} = \{(q_0, a_0, q_1)(q_1, a_1, q_2)(q_2, a_2, q_3)\cdots \in \Delta^\omega \mid q_i \in F \text{ for some } i\}.$$

Furthermore, an ω-automaton $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ is a *parity automaton*, if there is some coloring $\Omega\colon Q \to \mathbb{N}$ such that

$$\mathrm{Acc} = \{\rho \in \Delta^\omega \mid \max\{\Omega(q) \mid q \in \mathrm{Inf}(\rho)\} \text{ is even}\}.$$

In the remainder, we use the acronyms DPA resp. NPA to refer to deterministic parity automata resp. non-deterministic parity automata.

Finally, an ω-automaton $\mathfrak{A} = (Q, \Sigma, q_I, \Delta, \mathrm{Acc})$ is a *Muller automaton* resp. a *weak Muller automaton*, if there is a family $\mathcal{F} \subseteq 2^Q$ of sets of states such that

$$\mathrm{Acc} = \{\rho \in \Delta^\omega \mid \mathrm{Inf}(\rho) \in \mathcal{F}\}$$

resp.

$$\mathrm{Acc} = \{\rho \in \Delta^\omega \mid \mathrm{Occ}(\rho) \ \in \ \mathcal{F}\}.$$

In the remainder, we use the acronyms wDMA resp. wNMA to refer to deterministic weak Muller automata resp. non-deterministic weak Muller automata.

The sets Acc are infinite objects, but they have finite representations. E.g., for safety conditions Acc is fully specified by a subset $F \subseteq Q$ of safe states; for (weak) Muller conditions Acc is fully specified by a set $\mathcal{F} \subseteq 2^Q$ of sets of states. While these representations are usually small in the size of the automaton, this is not guaranteed for Muller acceptance conditions. Working with Muller automata, the representations of the acceptance condition influence the complexity of decision problems [14, 13]. Hence, we recall different ways of representing Muller conditions which are more succinct than simply listing the elements of $\mathcal{F}$.

Representations of acceptance conditions. Hunter and Dawar studied the impact of the representation of Muller conditions on the complexity of solving delay-free Muller games [14, 13]. In this paper, we follow their terminology and definitions for these representations.

The most straightforward representation of an acceptance condition of a Muller automaton is an *explicit condition*, i.e., an explicit list of the elements of $\mathcal{F}$. For short, we refer to a wDMA resp. wNMA where the acceptance condition is represented as an explicit condition as explicit-wDMA resp. explicit-wNMA.

But there are more succinct representations of Muller automata acceptance conditions, such as *Muller conditions over colors*, *win-set conditions*, *Emerson-Lei conditions*, and *circuit conditions* (see [14, 13] for definitions and for a comparison in the setting of delay-free games).

In this work, we consider Emerson-Lei conditions which are Boolean formulas φ with variables from Q. The set $\mathcal{F}_\varphi$ specified by φ is the set of sets $F \subseteq Q$ such that the truth assignment that maps each element of F to true and each element of $Q \setminus F$ to false satisfies φ. The size of a formula φ is denoted $|\varphi|$ and is defined as usual as its length. For short, we refer to a wDMA resp. wNMA where the acceptance condition is represented as an Emerson-Lei condition as an Emerson-Lei-wDMA resp. an Emerson-Lei-wNMA.

Let us conclude by mentioning that Emerson-Lei conditions are more succinct than Muller conditions over colors, which are more succinct than win-set conditions, which are more succinct than explicit conditions. Circuit conditions are at least as succinct as Emerson-Lei conditions, but it is open if circuit conditions are more succinct than Emerson-Lei conditions (meaning it is open whether circuits can always be translated into small formulas). These (and further) results can be found in [14, 13].

Size of automata. As discussed in the previous paragraph, the representation of the acceptance condition of a (weak) Muller automaton can be more or less succinct. This should be reflected when defining the size of a (weak) Muller automaton. Hence, we define the size of a (weak) Muller automaton with Emerson-Lei condition φ as $|Q| + |\varphi|$; the size of a (weak) Muller automaton with explicit condition $\mathcal{F}$ as $|Q| + |\mathcal{F}|$.

For other ω-automata acceptance conditions such as, e.g., reachability, safety or parity, we define the size of the automaton as its number of states, as the acceptance condition can be encoded with a polynomial overhead.

Delay games. A *delay function* is a mapping $f\colon \mathbb{N} \to \mathbb{N}_{\geq 1}$, which is said to be constant if $f(i) = 1$ for all $i > 0$. A *delay game* $\Gamma_f(L)$ consists of a delay function f and a *winning condition* $L \subseteq (\Sigma_I \times \Sigma_O)^\omega$ for some alphabets Σ_I and Σ_O. Such a game is played in rounds $i = 0, 1, 2, \ldots$ as follows: in round i, first Player I picks a word $u_i \in \Sigma_I^{f(i)}$, then Player O picks a letter $v_i \in \Sigma_O$. Player O *wins* a play $(u_0, v_0)(u_1, v_1)(u_2, v_2)\cdots$ if the outcome $\binom{u_0 u_1 u_2 \cdots}{v_0 v_1 v_2 \cdots}$ is in L; otherwise, Player I wins. Note that if f is a constant delay function, then Player O has a constant lookahead of $f(0)$ letters on her opponents moves. Hence, the moniker constant refers to the lookahead induced by f, not to f itself.

A *strategy* for Player I in $\Gamma_f(L)$ is a mapping $\tau_I\colon \Sigma_O^* \to \Sigma_I^*$ satisfying $|\tau_I(w)| = f(|w|)$ while a strategy for Player O is a mapping $\tau_O\colon \Sigma_I^+ \to \Sigma_O$. A play $(u_0, v_0)(u_1, v_1)(u_2, v_2)\cdots$ is *consistent* with τ_I if $u_i = \tau_I(v_0 \cdots v_{i-1})$ for all i, and it is consistent with τ_O if $v_i = \tau_O(u_0 \cdots u_i)$ for all i. A strategy for Player $P \in \{I, O\}$ is *winning*, if every play that is consistent with the strategy is won by Player P. We say that Player $P \in \{I, O\}$ *wins* a game $\Gamma_f(L)$ if Player P has a winning strategy in $\Gamma_f(L)$.

Problem statement. In this work, we investigate delay games with weak Muller automata winning conditions. We are interested in two aspects.

Firstly, what is the computational complexity of deciding whether Player O wins such a game? Secondly, how much lookahead does Player O need to win such a game if possible?

Note that delay games with wDMA resp. wNMA winning conditions are determined, because delay games with Borel winning conditions are determined [15]. Hence, we also speak of deciding whether Player O wins such a game as solving such a game.

We end this section with some introductory examples.

Example 2.1. In Fig. 1, we present a deterministic automaton gadget whose structure is suitable to store a 4-bit sequence. What we mean by that is, that for a word starting with $a_0a_1a_2a_3 \in \mathbb{B}^4$ the corresponding run $(p_0, a_0, p_1)(p_1, a_1, p_2)(p_2, a_2, p_3)(p_3, a_3, p_4)\cdots$ of the automaton yields that the state

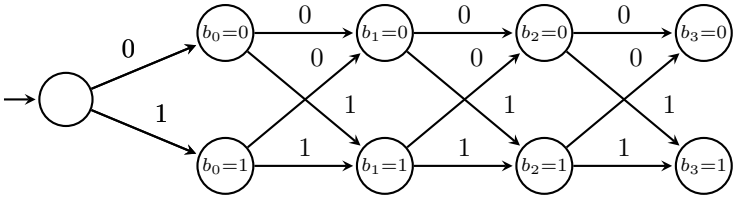

Figure 1: A gadget to store a 4-bit sequence.

p_{i+1} is equal to $_{b_i=a_i}$ for $0 \leq i \leq 3$. This idea is easily scaled to an n-bit sequence for arbitrary $n \in \mathbb{N}_{\geq 1}$. Note that the gadget is of linear size in n.

Now, we present a delay game which makes use of such a gadget.

Example 2.2. Let $L \subseteq (\mathbb{B}^2)^\omega$ be the ω-language that contains all words of the form (α, β) such that $\beta(i) = \alpha(4+i)$ for $0 \leq i \leq 3$. In words, the second four-bit segment of the input component is equal to the first four-bit segment of the output component.

We describe a wDMA $\mathfrak{A}$ that recognizes L. Basically, such an automaton is composed of two successive gadgets similar as in Fig. 1 and a sink state reached after the second gadget. In the first gadget, the output component of a letter determines the target state, and in the second gadget, the input component of a letter determines the target state. We assume that the states of the first gadget are called $G_1(_{b_0=0})$, $G_1(_{b_0=1})$, etc., and in the second gadget $G_2(_{b_0=0})$, $G_2(_{b_0=1})$, etc.

We give the acceptance condition of $\mathfrak{A}$ as a formula φ:

$$\begin{array}{ll} & (G_1(_{b_0=0}) \leftrightarrow G_2(_{b_0=0})) \\ \wedge & (G_1(_{b_1=0}) \leftrightarrow G_2(_{b_1=0})) \\ \wedge & (G_1(_{b_2=0}) \leftrightarrow G_2(_{b_2=0})) \\ \wedge & (G_1(_{b_3=0}) \leftrightarrow G_2(_{b_3=0})) \end{array}$$

Clearly, $L(\mathfrak{A}) = L$.

Now, let us consider a delay game with L as winning condition. Player O can win such a game if she is aware of the $(4+k)$-th letter that Player I plays before she has to give her k-th letter (for $1 \leq k \leq 4$). The constant delay function f with $f(0) = 5$ gives enough lookahead to ensure this.

We show that Player O wins $\Gamma_f(L)$. Due to the choice of $f(0)$, in each round i, Player I has produced a prefix $\alpha(0) \cdots \alpha(i+4)$ that Player O can base her move in that round on. Hence, she picks $\beta(i) = \alpha(i+4)$ in round i. This strategy is winning for her, as every consistent outcome is in L.

As for Example 2.1, this example is easily scaled to an n-bit sequence for arbitrary $n \in \mathbb{N}_{\geq 1}$. Note that a formula φ_n specifying the acceptance condition is of size $\mathcal{O}(n)$ while an explicit representation of $\mathcal{F}_{\varphi_n}$ is of size $\mathcal{O}(2^n)$ as all possible n-bit sequences must be explicitly represented.

In the above example, we presented a delay game where a small lookahead is sufficient for Player O to win. We now give an example of a family of ω-automata where, when used as a winning condition for a delay game, Player O needs exponential lookahead (in the size of the automaton) to win.

Example 2.3. Pick some $n \in \mathbb{N}_{\geq 1}$, and let $\Sigma_n = \{0, \ldots, n-1\}$. A sequence $\alpha \in (\Sigma_n)^* \cup (\Sigma_n)^\omega$ is said to contain a so-called *bad j-pair* if there are two positions $p < p'$ such that $\alpha(p) = \alpha(p') = j$ and $\alpha(q) < j$ for all $p < q < p'$.[2]

Now consider the language $P_n \subseteq (\Sigma_n \times \Sigma_n)^\omega$ that contains a word $\binom{\alpha}{\beta}$ if, and only if, $\alpha(1)\alpha(2)\cdots$ contains a bad j-pair where $j = \beta(0)$. In words, if the first output letter is j, then there is a bad j-pair in the input letter stream starting from the second letter.

We have that $P_n = L(\mathfrak{P}_n)$ where $\mathfrak{P}_n$ is the automaton specified in Fig. 2 when it is, for example, interpreted as a reachability automaton where q_f has to be reached. Note that $\mathfrak{P}_n$ is deterministic and of size $\mathcal{O}(n)$.

[2]Note that we could equivalently require $\alpha(q) \leq j$ for all $p < q < p'$. The latter allows occurrences of j between positions p and p'. However, one can then pick two such occurrences without any j's in between, which satisfy the stricter definition with $<$.

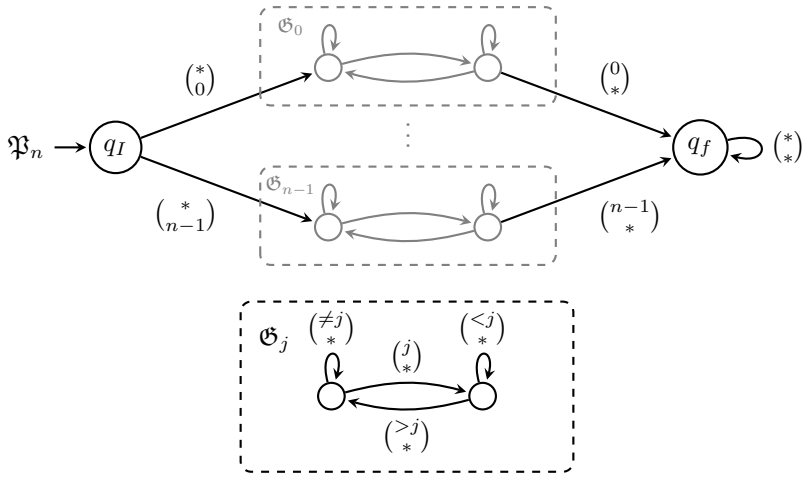

Figure 2: The automaton $\mathfrak{P}_n$ (top) contains gadgets $\mathfrak{G}_0, \ldots, \mathfrak{G}_{n-1}$ (bottom). Transitions not depicted lead to a sink state, which is not drawn. Here, $*$ denotes an arbitrary letter from the respective alphabet.

Klein and Zimmermann [16] have proven that in order to be guaranteed the existence of a bad j-pair for $0 \leq j \leq n-1$ a sequence over Σ_n of length at least 2^n is necessary.

Proposition 2.4. *Let $n \in \mathbb{N}_{\geq 1}$, and let $\Sigma_n = \{0, \ldots, n-1\}$.*

- *Every word $w \in (\Sigma_n)^*$ with $|w| \geq 2^n$ contains a bad j-pair for some $0 \leq j \leq n-1$.*
- *There is a word $w \in (\Sigma_n)^*$ with $|w| = 2^n - 1$ that does not contain a bad j-pair for every $0 \leq j \leq n-1$.*

With this in mind, it is intuitively true that in a delay game with winning condition P_n as given in Example 2.3, Player O can always win if she has at least a lookahead of 2^n on Player I's moves, but not with less lookahead. This was formally proven in [16]. In our upcoming proofs, we will incorporate variations of Example 2.3.

To conclude, we show that the game presented in Example 2.3 implies an exponential lower bound result on the necessary lookahead for Player O to win delay games with deterministic weak Muller automata winning conditions which are stated explicitly.

The following results is obtained by picking $L_n = P_n$ and showing that $\mathfrak{P}_n$ is an explicit-wDMA with set $\mathcal{F} = \{F_0, \ldots, F_{n-1}\}$, where the set F_i contains q_I and q_f as well as all (that is, the two states) from gadget $\mathfrak{G}_i$. To see this, consider a run that starts in q_I and reaches the sink q_f: only one of the gadgets $\mathfrak{G}_\mathfrak{i}$ is visited. Also, both states of $\mathfrak{G}_\mathfrak{i}$ must be visited.

Corollary 2.5. *For every $n \in \mathbb{N}_{\geq 1}$, there exists a language L_n recognized by an* explicit-wDMA *$\mathfrak{A}_n$ of size $\mathcal{O}(n)$ such that*

- *Player O wins $\Gamma_f(L_n)$ for some constant delay function f, but*
- *Player I wins $\Gamma_g(L_n)$ for every delay function g with $g(0) \leq 2^n$.*

3 Upper Bounds for Weak Muller Delay Games

We begin this section with a recap of known results used to prove our upper bounds on complexity and necessary lookahead. Klein and Zimmermann [16, 17] have shown the following complexity results about parity delay games.

Proposition 3.1.

1. *Solving delay games with* DPA *winning conditions is* ExpTime*-complete.*

2. *Solving delay games with* NPA *winning conditions is* 2ExpTime*-complete.*

Furthermore, Klein and Zimmermann [16, 17] have shown tight bounds on the necessary lookahead needed to win parity delay games. Here, we are interested in the upper bound results.

Proposition 3.2.

1. *For every* DPA $\mathfrak{A}$*, the following are equivalent:*
   - *Player* O *wins* $\Gamma_f(L(\mathfrak{A}))$ *for some constant delay function* f *where* $f(0)$ *is exponential in the size of* $\mathfrak{A}$*.*
   - *Player* O *wins* $\Gamma_g(L(\mathfrak{A}))$ *for some delay function* g*.*
2. *For every* NPA $\mathfrak{A}$*, the following are equivalent:*
   - *Player* O *wins* $\Gamma_f(L(\mathfrak{A}))$ *for some constant delay function* f *where* $f(0)$ *is doubly-exponential in the size of* $\mathfrak{A}$*.*
   - *Player* O *wins* $\Gamma_g(L(\mathfrak{A}))$ *for some delay function* g*.*

To obtain our upper bounds on weak Muller delay games stated below (see Theorems 3.4 and 3.5), we explicitly state the following easy result on the translation from weak Muller into parity automata.

Lemma 3.3.

1. *For every* wDMA *(in any representation) there exists an equivalent exponentially-sized* DPA*.*
2. *For every* wNMA *(in any representation) there exists an equivalent exponentially-sized* NPA*.*

Proof. Given a weak Muller automaton $\mathfrak{A}$, it suffices to construct a parity automaton $\mathfrak{A}'$ that additionally tracks the occurrence set of a run of $\mathfrak{A}$. To this end, the parity automaton uses the state set $Q \times 2^Q$ where Q is the set of states of $\mathfrak{A}$. The first component simulates a run of $\mathfrak{A}$ while the second one accumulates the occurrence set of the run prefix simulated thus far.

States of the form (q, O) with $O \in \mathcal{F}$ are assigned color 2 and those with $O \notin \mathcal{F}$ are assigned color 1. The resulting parity automaton is equivalent to the original weak Muller automaton, as the occurrence set eventually stabilizes. □

We are ready to prove our upper bounds (regarding complexity and necessary lookahead) for delay games with weak Muller automata winning conditions.

The combination of Proposition 3.1 and Lemma 3.3 immediately yields our first theorem which states upper bound complexity results.

Theorem 3.4.

1. *Solving delay games with* wDMA *winning conditions (in any representation) is in* 2ExpTime*.*
2. *Solving delay games with* wNMA *winning conditions (in any representation) is in* 3ExpTime*.*

The combination of Proposition 3.2 and Lemma 3.3 immediately yields our next theorem which gives upper bounds on the necessary lookahead for Player O to win delay games.

Theorem 3.5.

1. *For every* wDMA $\mathfrak{A}$ *(in any representation), the following are equivalent:*
   - *Player* O *wins* $\Gamma_f(L(\mathfrak{A}))$ *for some constant delay function* f *where* $f(0)$ *is doubly-exponential in the number of states of* $\mathfrak{A}$*.*
   - *Player* O *wins* $\Gamma_g(L(\mathfrak{A}))$ *for some delay function* g*.*
2. *For every* wNMA $\mathfrak{A}$ *(in any representation), the following are equivalent:*
   - *Player* O *wins* $\Gamma_f(L(\mathfrak{A}))$ *for some constant delay function* f *where* $f(0)$ *is triply-exponential in the number of states of* $\mathfrak{A}$*.*
   - *Player* O *wins* $\Gamma_g(L(\mathfrak{A}))$ *for some delay function* g*.*

In Sections 4 and 5 we show matching lower bounds for Theorems 3.4 and 3.5.

4 Lower Bounds for Deterministic Weak Muller Delay Games

This section is devoted to showing lower bounds for delay games with deterministic weak Muller automata winning conditions.

Lookahead. We begin this section by showing a doubly-exponential lower bound on the necessary lookahead for Player O to win, which yields a tight bound in combination with Theorem 3.5.

Recall Example 2.3, where the concept of bad j-pairs was introduced, and an automaton of size $\mathcal{O}(n)$ was given where the first letter, say $i_0 \in [0, n-1]$ of the output component indicates the existence of a bad i_0-pair in the input component (starting from the second letter) which is a word over $[0, n-1]$. Recall that according to Proposition 2.4, one needs exponential lookahead in n to correctly identify a bad j-pair in a word over $[0, n-1]$.

We are going to design a variant where a doubly-exponential lookahead is necessary. To this end, we encode the numbers in the range $[0, 2^n - 1]$ in binary and show how to construct a small automaton that checks whether the input component has a bad i_0-pair, where i_0 is the first number in the output component. This results in a game that Player O can only win with doubly-exponential lookahead.

In Example 2.3, the automaton stores i_0 in its state space, as there are only n possibilities. However, this is no longer feasible with the range $[0, 2^n - 1]$ and a polynomially-sized automaton, as there are 2^n possible values for i_0. Instead, Player O has to mark two numbers she claims to form a bad i_0-pair and the automaton then checks whether the two marked numbers are equal to i_0 and whether all numbers in between are strictly smaller than i_0. Using the succinctness of the Emerson-Lei condition, this can be achieved with a linearly-sized automaton.

Theorem 4.1. *For every $n \in \mathbb{N}_{\geq 1}$, there exists a language L_n recognized by an* Emerson-Lei-wDMA *$\mathfrak{A}_n$ of size $\mathcal{O}(n)$ such that*

- *Player O wins $\Gamma_f(L_n)$ for some constant delay function f, but*
- *Player I wins $\Gamma_g(L_n)$ for every delay function g with $\sum_{i=0}^{n-1} g(i) \leq 2^{2^n}$.*

Proof. Pick some $n \in \mathbb{N}_{\geq 1}$. Our goal is to give a language L_n recognized by an Emerson-Lei-wDMA $\mathfrak{A}_n$ of size $\mathcal{O}(n)$ such that the input component encodes a sequence $x_0, x_1, x_2, \ldots$ of numbers $x_i \in [0, 2^n - 1]$, and the output component encodes a sequence $y_0, y_1, y_2, \ldots$ of numbers $y_i \in [0, 2^n - 1]$. Each x_i and y_i is encoded as an n-bit sequence. For acceptance, we require that $x_1 x_2 \cdots$ contains a bad y_0-pair. According to Proposition 2.4, finding a bad j-pair in a word over $[0, 2^n - 1]$ requires in the worst case a word of length 2^{2^n}. Consequently, using L_n as the winning condition for a delay game will ensure that a doubly-exponential lookahead (in n) is necessary.

We go into details. The automaton we construct uses the product alphabet $\{0, 1, \#\} \times \{0, 1, \mathsf{X}, \#\}$. We first describe what we call valid input resp. output encodings:

- An input sequence α is a valid encoding if it is of the form

  $$((0+1)^n \#)^\omega.$$

  The i-th n-bit sequence encodes the number $x_i \in [0, 2^n - 1]$, the least significant bit is the leftmost one.

- An output sequence β is a valid encoding if it is of the form

  $$((0+1)^n (\mathsf{X} + \#))^\omega$$

  such that X occurs exactly twice. The i-th n-bit sequence encodes the number $y_i \in [0, 2^n - 1]$, the least significant bit is the leftmost one.

The behavior of the automaton is described below, where we distinguish three cases. The first two cases simply state that (viewed as a game) the player who first violates the correct encoding format loses.

1. If there exists some i such that $\alpha(0)\alpha(1)\cdots\alpha(i)$ cannot be completed to a valid input encoding, but $\beta(0)\beta(1)\cdots\beta(i-1)$ can be completed to a valid output encoding, the automaton accepts.

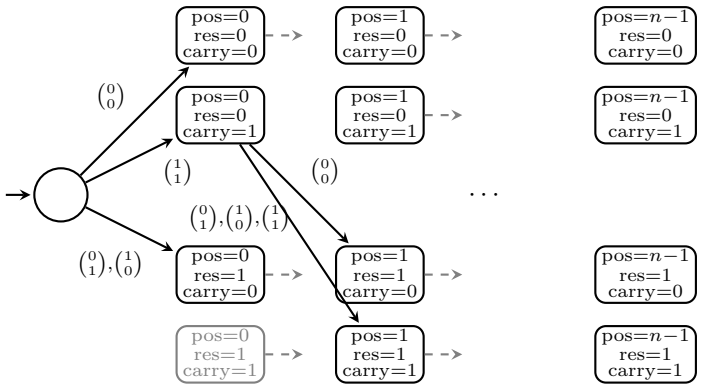

Figure 3: A gadget to add two n-bit sequences in little endian notation. Missing transitions are implied by gray dashed arrows. States for positions 2 to $n-2$ are not drawn. The state label $\begin{smallmatrix}\text{pos}=k\\ \text{res}=b\\ \text{carry}=b'\end{smallmatrix}$ indicates that adding the bits at position k yields the result b and b' is the carry bit relevant for the addition at position $k+1$. The state $\begin{smallmatrix}\text{pos}=0\\ \text{res}=1\\ \text{carry}=1\end{smallmatrix}$ is not reachable and can be omitted.

2. If there exists some i such that $\beta(0)\beta(1)\cdots\beta(i)$ cannot be completed to a valid output encoding, but $\alpha(0)\alpha(1)\cdots\alpha(i)$ can be completed to a valid input encoding, the automaton rejects.

3. If α and β are valid encodings, the automaton accepts if a bad y_0-pair is correctly marked.

We first explain the idea of Item 3. The intention of the marker ✕ is to be placed before two numbers x_j and $x_{j'}$ such that $x_j = x_{j'} = y_0$ and they enclose a bad y_0-pair, i.e., $x_i < y_0$ for $j < i < j'$.

In order to check whether $x_j = x_{j'} = y_0$ it suffices to have three gadgets similar as in Example 2.1 for n-bit sequences. The first gadget stores y_0, we index its states with S for start, The second and third gadgets are entered when the first resp. second ✕ is seen, and thus store x_j and $x_{j'}$, we index its states with M_1 resp. M_2 for marked numbers. Validating whether $x_j = x_{j'} = y_0$ is done via the acceptance condition by a formula similar to the formula presented in Example 2.2, which we present further below. Our formula also enforces that there are two marked numbers.

Instead of checking whether $x_i < y_0$ for $j < i < j'$, the automaton checks for another condition which is expressible with a small enough automaton and formula. Namely, whether $x_i + y_i = y_0$ for $j < i < j'$. Note that this is only equivalent to $x_i \leq y_0$. However, recall Footnote 2. The intention is that, during a play, Player O can make this condition true if, and only if, x_j and $x_{j'}$ indeed enclose a bad y_0-pair.

In order to check whether $x_i + y_i = y_0$ for all $j < i < j'$, one gadget is used that computes the addition of two n-bit sequences. Such a gadget is depicted in Fig. 3. We index its states by A for addition. This gadget is entered for every n-bit sequence between the first two marked sequences. Expressing that the automaton accepts if $x_i + y_i = y_0$ for all $j < i < j'$ is expressed via a formula which we give further below.

Now, we briefly explain the overall automaton structure. We construct a deterministic automaton, thus, Items 1 to 3 must be handled in parallel. Note that the gadgets we introduced for Item 3 can easily be extended to check whether either sequence is invalid (i.e., whether Item 1 or Item 2 holds) without introducing new states for the gadgets. The automaton needs two additional sink states q_{acc} resp. q_{rej} that are entered when the input resp. output sequence is not a valid encoding.

After the S-gadget and before the M_1-gadget, and after the M_2-gadget, it suffices to check whether Item 1 or Item 2 holds and go to the corresponding sink state. This can be done with $\mathcal{O}(n)$ many states.

All in all, $\mathfrak{A}_n$ is a sequence of six different gadgets which are all of size $\mathcal{O}(n)$, so $\mathfrak{A}_n$ is of size $\mathcal{O}(n)$:

1. The first gadget, S, stores y_0 the first number picked by Player O.

2. The second gadget checks that the format of the input and output is correct.

3. The third gadget, M_1, is entered when the first ✕ occurs, say at the beginning of the j-th block. It stores x_j.

4. The fourth gadget, A, is entered immediately after the j-th block and only left when the second ✕ occurs, say at the beginning of the j'-th block. It checks that $x_i + y_i = y_0$ holds for all $j < i < j'$.

5. The fifth gadget, M_2, is entered when the second ✕ occurs. It stores $x_{j'}$, the number marked by the second ✕.

6. The last gadget again checks that the format of the input and output is correct.

Note that the gadgets S, M_1, M_2, and A also check the format of the input and output is correct. As explained earlier, this does not increase their size.

Finally, we are ready to present the acceptance condition φ_n of the automaton $\mathfrak{A}_n$. We state its intended meaning in words first:

- the rejecting sink is not seen, and
- the accepting sink is seen, or
- for every position $k \in [0, n-1]$ and every bit value $b \in \mathbb{B}$:
  - If $y_0(k) = b$, then $x_j(k) = b$ and $x_{j'}(k) = b$, i.e., the two marked numbers are equal to the first one in the output.
  - If $y_0(k) = b$, then the k-th bit of $x_i + y_i$ is not $1 - b$ for every $j < i < j'$, i.e., all pairs of numbers between the marked ones add up to the first one in the output.

We define $\varphi_n = \neg q_{rej} \wedge (q_{acc} \vee \psi_n)$ with

$$\psi_n = \bigwedge_{k \in [0,n-1]} \bigwedge_{b \in \mathbb{B}} S(b_k{=}b) \rightarrow \left[M_1(b_k{=}b) \wedge M_2(b_k{=}b) \wedge \bigwedge_{b' \in \mathbb{B}} \neg A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}1{-}b \\ \text{carry}{=}b' \end{pmatrix} \right].$$

Note that φ_n is of size $\mathcal{O}(n)$.

Now, we show that

- Player O wins $\Gamma_f(L_n)$ for some constant delay function f, but
- Player I wins $\Gamma_g(L_n)$ for every delay function g with $\sum_{i=0}^{n-1} g(i) \leq 2^{2^n}$.

We start by giving a constant delay function f such that Player O has a winning strategy in $\Gamma_f(L(\mathfrak{A}_n))$. According to Proposition 2.4, the sequence $x_1 \cdots x_{2^{2^n}}$ of numbers is guaranteed to contain a bad j-pair for some $j \in [0, 2^n - 1]$. Hence, we let $f(0) = m$, where m is some number large enough to ensure that $x_1, \ldots, x_{2^{2^n}}$ is included in the lookahead (assuming that Player I plays a valid input encoding). If Player I plays a valid input encoding, Player O's strategy is to output some y_0 such that $x_1 \cdots x_{2^{2^n}}$ contains a bad y_0-pair. She marks the pair correctly with ✕, and can ensure that she produces output y_i such that $x_i + y_i = y_0$ in between the marked numbers. At other positions, she plays arbitrary numbers, and no other marks.

We argue that φ_n is satisfied. The state q_{rej} is never reached, as if Player I does not play a valid input encoding, the state q_{acc} is reached independently of Player O's moves and the formula is trivially satisfied. Assume that Player I plays a valid input encoding, and Player O behaves as described. Clearly, $S(b_k{=}b) \rightarrow (M_1(b_k{=}b) \wedge M_2(b_k{=}b))$ for all $k \in [0, n-1]$ and $b \in \mathbb{B}$. Since for all numbers y_i between the marked numbers $x_i + y_i = y_0$ holds, we also have that if $S(b_k{=}b)$ has been seen, then the k-th bit of $x_i + y_i$ is equal to b. Hence, only the states $A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}b \\ \text{carry}{=}0 \end{pmatrix}$ and $A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}b \\ \text{carry}{=}1 \end{pmatrix}$ can be seen. This implies that $\neg A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}1{-}b \\ \text{carry}{=}0 \end{pmatrix} \wedge \neg A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}1{-}b \\ \text{carry}{=}1 \end{pmatrix}$ is true.

For the other direction, assume that g is a delay function such that $\sum_{i=0}^{n-1} g(i) \leq 2^{2^n}$. We show that Player I wins $\Gamma_g(L_n)$. According to Proposition 2.4, there exists a word w over $[0, 2^n - 1]$ with $|w| = 2^{2^n} - 1$ that contains no bad j-pair for every $j \in [0, 2^n - 1]$. Player I's strategy is to play a valid input encoding such that x_i is the n-bit sequence that encodes $w(i)$ for all $0 \leq i \leq 2^{2^n} - 1$. After round $n - 1$, Player O has finished y_0. Furthermore, the condition $\sum_{i=0}^{n-1} g(i) \leq 2^{2^n}$ ensures that Player I has

not yet begun to spell the n-bit sequence that encodes $x_{2^{2^n}}$. Hence, since y_0 is known to Player I, he can pick a number $x \neq y_0$ and continues to produce a valid input encoding such that every x_i encodes x for all $i \geq 2^{2^n}$.

We show that Player I wins playing as described. Player O's goal is to satisfy the formula φ_n. Clearly, she loses if her moves do not produce a valid output encoding (as q_{rej} would be reached), so we assume that she produces a valid output encoding. The state q_{acc} is not reached, as it is only reached if the input sequence is invalid, hence, the other part of the formula must be satisfied. Note that the implication $S(b_k{=}b) \rightarrow (M_1(b_k{=}b) \wedge M_2(b_k{=}b))$ for all $k \in [0, n-1]$ and $b \in \mathbb{B}$ can only be satisfied if Player O marks two numbers with ✕, because the S-gadget stores y_0 making the lefthand-side of the implication true. However, since Player I's moves ensured that the input sequence does not contain a bad y_0-pair, the righthand-side of the implication

$$\bigwedge_{k\in[0,n-1]} \bigwedge_{b\in\mathbb{B}} S(b_k{=}b) \rightarrow \left[M_1(b_k{=}b) \wedge M_2(b_k{=}b) \wedge \bigwedge_{b'\in\mathbb{B}} \neg A \begin{pmatrix} \text{pos}{=}k \\ \text{res}{=}1{-}b \\ \text{carry}{=}b' \end{pmatrix} \right]$$

is not satisfied. We have shown that Player I wins $\Gamma_g(L(\mathfrak{A}_n))$. □

Complexity. Next, we settle the complexity of solving delay games with Emerson-Lei-wDMA winning conditions by proving a 2ExpTime lower bound. Our construction is a generalization of the ExpTime lower bound for solving delay games with winning conditions given by deterministic parity automata [16].

Intuitively, Player I produces a sequence of configurations of an alternating exponential-space Turing machine. Each of these is of exponential length, i.e., the cells of each configuration can be addressed with polynomially many bits. Now, Player I picks the successor configuration of universal configurations while Player O picks the successor configuration of existential configurations (by picking a transition to apply). Relying on the lookahead, Player O has always access to the full current configuration picked by Player I before she has to pick a transition to apply. Dually, to account for the lookahead, Player I is allowed to copy a configuration to fill the lookahead while he waits for Player O to pick a transition. Hence, two successive configurations played by Player I should either be equal or the second one should be a successor configuration of the first one.

To force Player I to faithfully simulate the Turing machine, i.e., to only copy configurations or to pick a valid successor configuration (in particular the one determined by Player O in case of existential configurations), Player O can mark cells for the automaton to check for correctness: This is sufficient as an error by Player I manifests itself in a single wrongly updated or copied cell. Using the addresses of the cells and the markers used by Player O, a small automaton can check whether there is an error or not. Finally, Player O is able to correctly apply the markers, as she has enough lookahead to always observe the next two configurations, enough to spot errors introduced by Player I.

Altogether, we obtain a delay game simulating an alternating exponential-space Turing machine, and AExpSpace = 2ExpTime yields the desired result.

Theorem 4.2. *Solving delay games with* Emerson-Lei-wDMA *winning conditions is* 2ExpTime*-complete.*

Proof. The upper bound was shown in Theorem 3.4, we show the corresponding lower bound.

We give a reduction from the word problem for alternating exponential space Turing machines. Since 2ExpTime = AExpSpace [5], we obtain the desired result.

Let $\mathcal{M} = (Q = Q_\exists \uplus Q_\forall, \Sigma, q_I, \Delta, q_A, q_R)$ be an alternating exponential space Turing machine, where $\Delta \subseteq Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, 1\}$. Furthermore, let p be a polynomial such that 2^p bounds the space consumption of $\mathcal{M}$ and let $w \in \Sigma^*$ be an input word. We fix $n = p(|w|)$. Wlog., we assume that the accepting state q_A and the rejecting state q_R are equipped with a self-loop. Furthermore, wlog., we assume that either q_A or q_R is reached in every run of $\mathcal{M}$ on w. An alternating exponential space Turing machine that does not have this property can be turned into an equivalent TM $\mathcal{M}'$ that satisfies the property as follows: The number of configurations that can be reached on w is bounded (doubly-exponentially in n). Hence, $\mathcal{M}$ can be equipped with an exponentially-sized counter in order to detect that a configuration repetition has occurred (when the counter has surpassed the maximum) and reject if so. Then, $\mathcal{M}'$ is used as the starting point for the reduction instead of $\mathcal{M}$.

We construct an Emerson-Lei-wDMA $\mathfrak{A}$ of polynomial size in $(|\Delta| + n)$ such that $\mathcal{M}$ accepts w if, and only if, Player O wins $\Gamma_f(L(\mathfrak{A}))$ for some delay function f.

The idea is that, in the delay game, the players build a run of $\mathcal{M}$ on w in the form of a configuration sequence. Player I controls the universal states and Player O the existential ones. Furthermore, Player I spells all configurations with his moves. The configurations are of exponential size in n, so each cell of $\mathcal{M}$'s tape is addressed by an n-bit counter. In between configurations there is a delimiter, either C or N, to denote if the next configuration is a *copy* of the previous one or a *new* one. The need to copy configurations arises since, in the delay game, Player O is behind with her moves, so Player I needs to wait for Player O's pick if the current configuration is existential.

We go into details. The automaton we construct uses the product alphabet

$$(\{0, 1, C, N, \textvisiblespace\} \cup Q \cup \Sigma) \times (\{\checkmark, \text{✗}, \text{✻}\} \cup \Delta)\,.$$

Recall that Q, Δ, and Σ are components of the Turing machine $\mathcal{M}$ and are used to encode configurations and to pick transitions to apply. We first describe what we call valid input resp. output formats:

- An input sequence α is valid if it is of the form

$$\Big((C + N)\left[\,(0 + 1)^n\,(\Sigma + Q + \textvisiblespace)\,\right]^+\Big)^\omega,$$

  to be interpreted as a sequence of configurations $(C + N)c_0(C + N)c_1 \cdots$ where each c_i is a configuration delimited by C or N. A configuration is encoded by a sequence $\left[(0+1)^n(\Sigma+Q+\textvisiblespace)\right]^+$, where each n-bit sequence encodes an address $a \in [0, 2^n-1]$, the least significant bit is the rightmost one, and the letter after the address encoding is either the tape cell content (that is, a letter $\sigma \in \Sigma$, or a blank (denoted as $\textvisiblespace$)) or a state $q \in Q$. Note that the format does not imply that a sequence in $\left[(0 + 1)^n(\Sigma + Q + \textvisiblespace)\right]^+$ encodes a configuration of length 2^n. This will be later enforced by the rules of the game.

- An output sequence β is valid format if it is of the form

$$\Big(\Delta\big[(\checkmark + \text{✗} + \text{✻})^{n+1}\big]^+\Big)^\omega,$$

  and if for every position i, $\alpha(i) \in \{C, N\}$ if, and only if, $\beta(i) \in \Delta$, i.e., the letters $\tau \in \Delta$ occur at the same positions as the configuration delimiters C and N. We give the intended interpretation of this format, which is explained in more detail below: The markers ✓ resp. ✗ and ✻ are intended to indicate the absence of resp. the existence of an error. If ✗ marks a bit in an address block, it should indicate an error in updating the address block at this bit, if ✻ marks the first bit of an address block, then it should indicate that there is a copy or update mistake in the successive configuration which manifests itself at the cell with the marked address.

The behavior of the automaton is as follows:

1. If there exists some i such that $\alpha(0)\alpha(1)\cdots\alpha(i)$ cannot be completed to have a valid format, but $\beta(0)\beta(1)\cdots\beta(i-1)$ can be completed to have a valid format, the automaton accepts.
2. If a configuration encoding c contains not exactly one state or the first address block is not zero, the automaton accepts.
3. If c_0 does not encode the initial configuration on w, it accepts.
4. If there exists some i such that $\beta(0)\beta(1)\cdots\beta(i)$ cannot be completed to have a valid format, but $\alpha(0)\alpha(1)\cdots\alpha(i)$ can be completed to have a valid format, the automaton rejects.

We refer to Items 1 to 3 as simple input errors, and to Item 4 as simple output error. These conditions can be easily checked using the automaton structure with $\mathcal{O}(|\Delta| + n)$ states. We use designated sink states q_{acc} and q_{rej}.

5. If there are no simple errors, and there is a position marked by ✗ in some address block a_i (say $a_i(k)$), the automaton checks if $a_{i+1}(k)$ (assuming there is no $a_i(k')$ for some $k' < k$ which is also marked by ✗) is wrongly updated in the next address block a_{i+1}. We call this an *address update error*. If the address is indeed wrongly updated, then the automaton accepts, otherwise it rejects. We explain further below how to achieve this behavior.

6. If there are no simple errors, and there is an address block a whose first position is marked by ✻ the following is checked:
   - The next configuration must contain an address block a' whose first position is marked by ✻, otherwise the automaton rejects. (This is easily checkable via the automaton structure.)
   - If $a \neq a'$, the automaton rejects.
   - If $a = a'$, check whether the tape cell addressed by a contains an error:
     - If the delimiter between the configurations is C: The cell content is not correctly copied. We call this *copy error*.
     - If the delimiter between the configurations is N: The cell content is not correctly updated according to the picked transitions. We call this *update error*.

     If an error is claimed correctly, the automaton accepts, otherwise it rejects.

   We explain further below how to achieve this behavior.

7. Lastly, if none of the above cases occurs, if there is a configuration that contains q_A, then the automaton accepts, if there is a configuration that contains q_R, then the automaton rejects. Therefore, when a rejecting resp. accepting configuration has been seen, the automaton goes into designated new sink states q_R resp. q_A.

Now, we explain how to handle Item 5, i.e., the detection of an address update error. When a bit is marked with ✗, its value is stored in the state space. Using a modulo counter, it is easy to find the same bit in the next address block. It is not difficult to see that the value of a bit only flips if all remaining bits (that is, all bits to the right) are one, because then adding one to the address counter causes an overflow of all less significant bits (which are denoted to the right of the marked bit). Hence, an automaton can check with $\mathcal{O}(n)$ states whether a marked address bit has been updated correctly.

Regarding Item 6, checking whether the two marked (with ✻) n-bit sequences are the same can be done by using two gadgets and a formula as explained in Examples 2.1 and 2.2. Such a formula is of size $\mathcal{O}(n)$.

In order to check whether there is a copy error between successive configurations it suffices to remember whether the delimiter between them is C and the cell content of the marked cell.

Checking whether there is an update error between successive configurations c and c' is more involved:

- The delimiter between them has to be N.
- A window of three cell contents around the cell whose address is marked must be remembered.
- If c is a universal configuration, it has to be verified (using the stored three-cell window) whether the update in c' is achievable by applying a transition from Δ. (It is not specified in the input sequence which transition should be applied.)
- If c is an existential configuration, the transition to be applied is specified in the output sequence. However, the transition to be applied has been given (possibly much) earlier: When the delimiter before a configuration is N, the configuration is new. If this configuration is existential, in the output sequence, the letter after the configuration specifies the transition to be applied.
- Transitions that are given after copied configurations as well as universal configurations are irrelevant.
- Thus, the automaton stores a specified transition only if it occurs after a new existential configuration. This is the transition to be applied to obtain the next new configuration.

An automaton needs a polynomial number of states (in $(|\Delta| + n)$) to store the required pieces of information and verify that there is an update error.

Since we are constructing a wDMA, all of Items 1 to 7 need to be checked simultaneously. Therefore we build a product automaton which we denote by $\mathfrak{A}$ with one special feature, namely, the introduced sink states $q_{acc}, q_{rej}, q_A, q_R$ are global sink states (as opposed to product states).

This product automaton has a number of states polynomial in $(|\Delta| + n)$ since it is constructed from gadgets whose number of states are polynomial in $(|\Delta| + n)$. As mentioned before, regarding Item 6, the formula needed to verify that two n-bit addresses are the same is of size $\mathcal{O}(n)$. Since $\mathfrak{A}$ is a product automaton, the formula describing that two address are equal needs to be adjusted to incorporate the product structure which causes a polynomial in $(|\Delta|+n)$ blow-up of the formula. We briefly explain how to take a product structure into account by a small example. Let $\mathfrak{C}_1, \mathfrak{C}_2, \mathfrak{C}_3$ be some gadgets with state sets Q_1, Q_2, Q_3, respectively, and the product automaton is of the form $\mathfrak{C}_1 \times \mathfrak{C}_2 \times \mathfrak{C}_3$. Say a formula only refers to states in $\mathfrak{C}_1$ which contains the state p. Then the updated formula must replace all occurrences of p with $\bigvee_{q \in Q_2} \bigvee_{r \in Q_3} (p, q, r)$.

All in all, the final formula φ looks as follows:

$$\varphi = \neg(q_{rej} \vee q_R) \wedge (q_{acc} \vee q_A \vee \varphi_{error?}),$$

where $\varphi_{error?}$ describes that either the gadgets to store the n-bit address sequence have not been entered (meaning no copy/update error needs to be verified), or that they store the same n-bit address sequence (the claimed copy/update error is verified via the automaton structure). The formula $\varphi_{error?}$ is adjusted to the product automaton structure.

Recall that $n = p(|w|)$, hence, φ is of polynomial size in $(|\Delta| + p(|w|))$. All in all, $\mathfrak{A}$ is of polynomial size in $(|\Delta| + p(|w|))$.

It is left to show that $\mathcal{M}$ accepts w if, and only if, Player O wins $\Gamma_f(L(\mathfrak{A}))$ for some delay function f.

To begin with, assume $\mathcal{M}$ accepts w. Let f be a constant delay function such that $f(0) = m$, where m is chosen such that Player O has enough lookahead to see a full configuration before she has to start with its output. To be more concrete, $m = 2^n(1 + n) + 1$ suffices to capture a whole configuration including the address blocks and the delimiter.

We show that Player O wins $\Gamma_f(L(\mathfrak{A}))$. We assume that either player does not make simple errors (recall that this means playing in the wrong format). Firstly, assume that Player I either does not update an address correctly, or introduces a copy or update error. The lookahead is large enough to correctly claim such an error. The formula is then satisfied. Secondly, assume that Player I does not introduce such errors. Since $\mathcal{M}$ accepts w, Player O can pick existential transitions such that if Player I applies the picked transitions an accepting configuration will be reached. Note that, in order to pick an existential transition such that an accepting configuration will be reached, it is only necessary to have the knowledge of the current configuration. This fact can be easily seen when visualizing the run tree of $\mathcal{M}$ on w. If an accepting configuration is seen, the formula φ is clearly satisfied. Player I can prevent that an accepting configuration is seen by always copying the current configuration. But then, the formula φ is also satisfied, because the sub-formula $\varphi_{error?}$ is satisfied if no copy/update error is claimed or a copy/update error is correctly claimed. Player O does not claim such an error, as we assumed that Player I does not introduce such an error.

For the other direction, assume $\mathcal{M}$ rejects w. We show that Player I wins $\Gamma_g(L(\mathfrak{A}))$ for every delay function g. Again, we assume that either player does not make simple format errors. No matter the existential transitions that Player O choses, Player I can always pick universal transitions such that eventually a rejecting configuration is reached. Hence, we assume that Player I advances the run by giving new configurations according to his and Player O's choices.

We go into more detail. Whenever Player I has provided a new existential configuration, he has to wait for Player O to provide a transition to be applied since she is behind with her moves. While Player O has not provided her transition choice, Player I copies the current configuration until the choice of Player O is known. Then, Player I produces a new configuration obtained by applying Player O's choice. Whenever Player I has provided a new universal configuration there is no need to copy the configuration, a new configuration can be provided directly after.

Furthermore, we assume that Player I does not introduce address update errors or configuration copy/update errors as this is not beneficial for him. Player O loses a play where a rejecting configuration is reached. Hence, her only chance is to claim an error. Since there are no errors introduced by Player I, this approach is not fruitful as this consequently falsifies φ. Player O cannot not win a play, i.e., Player I wins $\Gamma_g(L(\mathfrak{A}))$. □

5 Lower Bounds for Non-Deterministic Weak Muller Delay Games

In this section, we show lower bounds for delay games with non-deterministic weak Muller automata winning conditions.

Lookahead. We show a triply-exponential lower bound on the necessary lookahead for Player O to win, which yields a tight bound in combination with Theorem 3.5.

To this end, we again implement the bad j-pair game described in Example 2.3, this time with numbers in the range $[0, 2^{2^n} - 1]$. As argued before, this allows Player O to win with triply-exponential lookahead, but not with less. Thus, we show that winning conditions specified by non-deterministic automata allow to implement the bad j-pair game with exponentially larger numbers than deterministic automata. This increase requires some additional mechanism to allow the automaton to check the existence of a bad j-pair, which we describe below.

The binary encoding of each number in $[0, 2^{2^n} - 1]$ is of exponential length, i.e., each bit can be addressed with n additional address bits. Player I produces a sequence of such encodings of numbers and Player O picks a number i_0 in her first move, claiming that the sequence picked by Player I contains a bad i_0-pair. To verify this claim, Player O is required to repeat i_0 ad infinitum.

Thus, both players can now cheat during their moves, Player I by not updating the addresses correctly and Player O by not copying i_0. Thus, both players need to use markers to claim such errors, unlike in the analogous result for deterministic automata.

However, due to the lookahead, Player I cannot mark the position where Player O has wrongly copied i_0. However, such an error manifests itself in a single bit and Player I is able to mark its address at a later position. As there are exponentially many addresses, we rely on non-determinism to guess for each address whether it is equal to the marked one, i.e., it needs to be checked for a copy error, or whether it is not equal to the marked address. It is this exponential succinctness, that leads to the increase in required lookahead from doubly-exponential to triply-exponential when allowing non-determinism.

Theorem 5.1. *For every $n \in \mathbb{N}_{\geq 1}$, there exists a language L_n recognized by an* Emerson-Lei-wNMA $\mathfrak{A}_n$ *of size $\mathcal{O}(n^3)$ such that*

- *Player O wins $\Gamma_f(L_n)$ for some constant delay function f, but*
- *Player I wins $\Gamma_g(L_n)$ for every delay function g with $\sum_{i=0}^{(n+2)\cdot 2^n} g(i) \leq 2^{2^{2^n}}$.*

Proof. Pick some $n \in \mathbb{N}_{\geq 1}$. Our goal is to give an Emerson-Lei-wNMA $\mathfrak{A}_n$ of polynomial size in n such that Player O needs triply-exponential lookahead in n to win the corresponding delay game. Recall Example 2.3 and Proposition 2.4. We are constructing a bad j-pair automaton over numbers ranging between 0 and $2^{2^n} - 1$ which enforces the need of a word composed of at least $2^{2^{2^n}}$ numbers to be guaranteed the existence of a bad j-pair.

To build an automaton of polynomial size in n, we encode a number $x \in R = [0, 2^{2^n} - 1]$ as a word over $\mathbb{B}$ as follows: To represent a number $x \in R$ in binary one needs 2^n bits, we address each bit by an n-bit sequence.

We go into more detail. A sequence of the form $a_0 b_0 \cdots a_{2^n-1} b_{2^n-1}$ such that $a_i \in \mathbb{B}^n$ and $b_i \in \mathbb{B}$ for $0 \leq i \leq 2^n - 1$ is called a *superblock*. A block $a_i \in \mathbb{B}^n$ is interpreted as an *address block* whose least significant bit is the rightmost one. The bit sequence $b_0 \cdots b_{2^n-1}$ is interpreted as a number $x \in R$, the least significant bit is the rightmost one. We refer to these bits as *special bits*.

In order to explain the idea, assume that in a play, Player I plays an input sequence that encodes a sequence of numbers $x_0, x_1, \ldots \in R$. We give a construction such that Player O's moves must produce an output sequence that encodes a sequence of numbers $y_0, y_1, \ldots \in R$ such that $x_1 x_2 \cdots$ contains a bad y_0-pair and it holds that $y_0 = y_1 = y_2 = \cdots$. In words, we enforce that Player O has to copy her first number ad infinitum. The technique to enforce Player O to copy her first number ad infinitum is as follows: When Player I realizes that she has not faithfully copied her number, then there exists an address a such that there are two superblocks whose values of the special bits addressed by a differ. Player I marks an address block that encodes a, and the winning condition (i.e., the automaton) enforces that all special bits which are addressed by a must have the same value. This is then violated, so Player O loses if she does not copy her choice of y_0 ad infinitum.

Furthermore, the automaton we construct verifies that the input sequence indeed contains a bad y_0-pair. Since it is enforced that $y_0 = y_1 = y_2 = \cdots$, verifying whether a bad y_0-pair exists is doable with a small enough automaton which we explain further below.

We go into details. The automaton we construct uses the product alphabet $\{0, 1, \times, \#\} \times \{0, 1, \#, \checkmark, \times\}$. We first describe what we call valid input resp. output format:

- An input sequence α has a valid format if it is of the form

$$\big((\times + \#)(0+1)^{n+1}\big)^{\omega},$$

  where $\times$ occurs exactly once.

- An output sequence β has a valid format if it is of the form

$$\big(\#(\checkmark + \times)^n(0+1)\big)^{\omega}.$$

  The intention is that $\checkmark$ resp. $\times$ are used to mark the absence resp. the presence of errors in updating the address blocks.

We now describe how the automaton $\mathfrak{A}_n$ behaves, then how to achieve this behavior.

1. If there exists some i such that $\alpha(0)\alpha(1)\cdots\alpha(i)$ cannot be completed to have a valid format, but $\beta(0)\beta(1)\cdots\beta(i-1)$ can be completed to have a valid format, the automaton accepts.
2. If there exists some i such that $\beta(0)\beta(1)\cdots\beta(i)$ cannot be completed to have a valid format, but $\alpha(0)\alpha(1)\cdots\alpha(i)$ can be completed to have a valid format, the automaton rejects.
3. If the very first address is not zero, it accepts.
4. If the k-th bit of an address block is marked with $\times$ (and no earlier $\times$ occurs), then it is verified if the k-th bit of the next address block is correctly updated. If the update is faulty, the automaton accepts, if the update is correct, the automaton rejects.
5. If $\times$ marks an address a, then the special bits picked by Player O that are addressed with a either have all value zero or all have value one. If this condition is violated, the automaton rejects.
6. If none of the conditions before have lead to acceptance or rejection, the automaton verifies whether $x_1x_2\cdots$ contains a bad y_0-pair. If the answer is yes, it accepts, otherwise it rejects.

We now describe how to achieve the desired behavior. Note that Items 1 to 3 can be easily checked in parallel via the automaton structure of an automaton with $\mathcal{O}(n)$ states. We use designated sink states q_{acc} resp. q_{rej} that are reached when the automaton accepts resp. rejects.

Note that Item 4 can be handled as Item 5 introduced in the proof of Theorem 4.2 where we re-use the sink states q_{acc}, q_{rej}. Again, $\mathcal{O}(n)$ states are used.

It is left to handle Items 5 and 6. Regarding Item 5, we need one gadget that stores the n-bit sequence that has been marked by $\times$. We refer to this sequence as address a. Such a gadget uses $\mathcal{O}(n)$ states and was first introduced (and used) in Examples 2.1 and 2.2. We index its states with A for address.

The automaton needs to verify that all special bits that are addressed by a have the same value. Therefore, the automaton behaves as follows:

- Before a new address block begins, the automaton guesses whether this address will be equal to a.
- If the automaton guesses that the addresses are equal:
  - It must be verified that the guess is right. If the guess is wrong, the automaton rejects.
  - In order to do the verification we need one additional gadget that stores an n-bit sequence such as in Example 2.1. We index its states by $A_=$ for equal address. The automaton enters this gadget every time it guesses that the address will be equal to a. Further below, we give a formula that is satisfied if, and only if, the guess is always right.

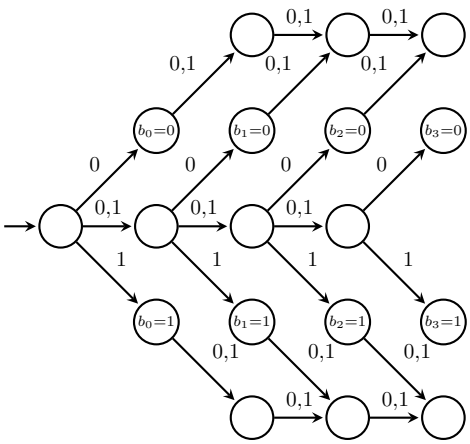

Figure 4: A gadget to (non-deterministically) store a single bit from a 4-bit sequence. The top and bottom row of states serve the same purpose (namely, a bit has been stored) so they could be merged, but it is easier to draw the gadget in this way. This gadget is easily generalized to an n-bit sequence.

  - Additionally, the automaton must store the values (as a set) of the special bits picked by Player O that occur as the next letter.

- If the automaton guesses that the addresses are not equal:
  - It must be verified that the guess is right. If the guess is wrong, the automaton rejects.
  - To verify, guess some $i \in [0, n-1]$ and check that the value of the i-th address bit is different from the i-th bit of the address a (recall, which is marked by ✕, and stored in gadget A).
  - In Fig. 4 we give a gadget that implements the guessing and storing of a bit. We index its states by G for guessing. The automaton enters this gadget every time it guesses that the address will be not equal to a.
  - Further below, we give a formula that is satisfied if, and only if, always a bit could be found such that the values at this bit in the inspected address and in address a differ.

Such an automaton needs $\mathcal{O}(n)$ states.

Lastly, regarding Item 6, the automaton guesses before the beginning of some superblock that its encoded number x_j is such that $x_j = y_0$. Whether a position is the beginning of a superblock can be guessed and verified by the automaton, as all its special bits are zero. Since Item 5 ensures that $y_j = y_0$ and y_j occurs below x_j it is easy to verify that the guess is correct (if the guess is wrong, the automaton rejects). Then, for the following numbers $x_{j+1}, x_{j+2}, \ldots$ it has to verify that $x_{j+1} < y_{j+1}, x_{j+2} < y_{j+2}, \ldots$ until some $x_{j'}$ is seen such that $x_{j'} = y_{j'}$. To verify that some number x is smaller than y, first, recall that x and y are encoded by their sequence of special bits such that the least significant bit is the rightmost one. If $x < y$, there exists a special bit at some position i such that all previous special bits had the same value for x and y, the special bit at position i has value zero in x and value one in y. The values of the special bits to the right play no role. If the verification fails, the automaton rejects by going to some new sink $q_{verif-fail}$. If some $x_{j'} = y_{j'}$ has been seen, the automaton goes into a new accepting sink $q_{verif-ok}$. Such an automaton needs a constant number of states, as the positions of special bits can be easily identified in a valid output sequence.

Recall that we construct a non-deterministic automaton, hence, the desired automaton $\mathfrak{A}_n$ is given as

$$(\mathfrak{A}_{format} \times \mathfrak{A}_{address}) \cup (\mathfrak{A}_{format} \times \mathfrak{A}_{copy} \times \mathfrak{A}_{pair})$$

where $\mathfrak{A}_{format}$ handles Items 1 to 3, $\mathfrak{A}_{address}$ handles Item 4, $\mathfrak{A}_{copy}$ handles Item 5, and $\mathfrak{A}_{pair}$ handles Item 6. The automaton has $\mathcal{O}(n^3)$ states.

The acceptance condition is given by the formula φ_n defined as

$$\neg q_{rej} \wedge \neg q_{verif-fail} \wedge (q_{acc} \vee (q_{verif-ok} \wedge \varphi_{copy}))\,,$$

where φ_{copy} is defined as

$$\begin{array}{ll} & \bigwedge_{k\in[0,n-1]} \bigwedge_{b\in\mathbb{B}} A_{=}(b_k{=}b) \rightarrow A(b_k{=}b) \\ \wedge & \bigwedge_{k\in[0,n-1]} \bigwedge_{b\in\mathbb{B}} A(b_k{=}b) \rightarrow \neg G(b_k{=}b) \\ \wedge & \varphi_{same-value}. \end{array}$$

The first line of φ_{copy} is satisfied if, and only if, the automaton whenever it has guessed "current address equals marked address" is right. If a such guess is wrong there is some $k \in [0, n-1]$ and $b \in \mathbb{B}$ such that $A(b_k{=}b)$ and $A_{=}(b_k{=}1{-}b)$ have been visited, falsifying the formula.

The second line of φ_{copy} is satisfied if, and only if, the automaton whenever it has guessed "current address is not equal to marked address" is right. Since the current address and the marked address differ in the value of at least one bit, the automaton can always guess to store this different bit in the gadget G. Hence, the second line of the formula is satisfiable in this way.

The formula $\varphi_{same-value}$ expresses that the relevant special bits either all have value zero, or all have value one.

Note that the formula φ_n needs to be adjusted to talk about the (product-like) structure of $\mathfrak{A}_n$ (which can be done as explained in the proof of Theorem 4.2), hence, its final size is $\mathcal{O}(n^3)$.

It is left to prove that

- Player O wins $\Gamma_f(L(\mathfrak{A}_n))$ for some constant delay function f, but
- Player I wins $\Gamma_g(L(\mathfrak{A}_n))$ for every delay function g with $\sum_{i=0}^{(n+2)\cdot 2^n} g(i) \leq 2^{2^{2^n}}$.

We start by giving a constant delay function f such that Player O has a winning strategy in $\Gamma_f(L(\mathfrak{A}_n))$. According to Proposition 2.4, the sequence $x_1 \cdots x_{2^{2^{2^n}}}$ of numbers is guaranteed to contain a bad j-pair for some $j \in [0, 2^{2^n} - 1]$. Hence, we let $f(0) = m$, where m is some number large enough to ensure that $x_1, \ldots, x_{2^{2^{2^n}}}$ is included in the lookahead (assuming that Player I plays valid superblocks with either # or ✕ (used once) in front of an address block). We call playing valid superblocks with either # or ✕ in front of an address block playing valid for short.

If Player I plays valid, Player O's strategy is to output some y_0 such that $x_1 \cdots x_{2^{2^{2^n}}}$ contains a bad y_0-pair. The number y_0 is repeated ad infinitum.

If Player I does not play valid, Player I either violates the format, or he introduces an error in updating the address counter. In the former case, q_{acc} is reached and the formula φ_n is trivially satisfied. In the latter case, Player O has enough lookahead to realize this and mark the error. Then, also q_{acc} is reached. The state q_{rec} is never reached as she does not violate the desired format.

The sub-formula φ_{copy} is also satisfied as she faithfully copies her first number y_0 ad infinitum. It is left to explain why $q_{verif-fail}$ is not reached and $q_{verif-ok}$ is reached. Since the sequence $x_1 x_2 \cdots$ contains a bad y_0-pair, it possible for the automaton to pick a run that correctly verifies this, hence, only $q_{verif-ok}$ is reached. Thus, we have shown that the formula φ_n is satisfied when Player O plays according to her strategy.

We turn to the other direction. Let g be a delay function with $\sum_{i=0}^{(n+2)\cdot 2^n} g(i) \leq 2^{2^{2^n}}$. We show that Player I wins $\Gamma_g(L(\mathfrak{A}_n))$.

According to Proposition 2.4, there exists a word w over $[0, 2^{2^n} - 1]$ with $|w| = 2^{2^{2^n}} - 1$ that contains no bad j-pair for every $j \in [0, 2^{2^n} - 1]$. Player I's strategy is to play a valid input sequence such that x_i is the bit sequence that encodes $w(i)$ for all $0 \leq i \leq 2^{2^{2^n}} - 1$.

Recall, a superblock is of length $(n+2) \cdot 2^n$ as we have 2^n address blocks of length n, 2^n special bits, and, additionally, 2^n address "markers" (# signifying unmarked, ✕ signifying marked). After round $(n+2) \cdot 2^n$, Player O has finished y_0. Furthermore, the condition $\sum_{i=0}^{(n+2)\cdot 2^n} g(i) \leq 2^{2^n}$ ensures that Player I has not yet begun to spell the superblock that encodes $x_{2^{2^{2^n}}}$. Hence, since y_0 is known to Player I, he can pick a number $x \neq y_0$ and continue to produce a valid input sequence such that every x_i encodes x for all $i \geq 2^{2^n}$.

We show that Player I wins playing as described. Player O's goal is to satisfy the formula φ_n. Clearly, she loses if her moves have the wrong format (as q_{rej} would be reached), so we assume that she produces an output sequence that has the right format. The state q_{acc} is not reached, as it is only reached if the input sequence is invalid, hence, $\neg q_{verif-fail} \wedge (q_{verif-ok} \wedge \varphi_{copy})$ must be satisfied. We can assume that Player O faithfully copies y_0 to satisfy φ_{copy}. But, since the input sequence does not contain a

bad y_0-pair, no matter which run $\mathfrak{A}_n$ takes, the verification of a bad y_0-pair is never successful. Hence, $q_{verif-fail}$ is reached, falsifying φ_n. Player I wins. □

Complexity. Finally, we turn to complexity showing a 3ExpTime lower bound on solving delay games with Emerson-Lei-wNMA winning conditions. To this end, we generalize the analogous construction for Emerson-Lei-wDMA presented in the proof of Theorem 4.2: Here, the players simulate a doubly-exponential space Turing machine. Thus, the use of non-deterministic automata for the winning condition allows us to add an additional exponential blowup in the space consumption of the Turing machine. Again, this requires some additional mechanisms, e.g., both players have to check their opponent's moves for correctness. We describe these changes in comparison to the construction for deterministic automata below.

As before, Player I is in charge of producing the sequence of configurations (with repetitions to fill the lookahead) and of picking successors of universal configurations while Player O picks successors for existential configurations (by picking a transition for Player I to apply). As configurations are now of doubly-exponential length, we address their cells by exponentially long addresses. Hence, the bits of these addresses have to be addressed using linearly-sized addresses as well, i.e., there are two levels of addresses. Player O has again markers to force Player I to faithfully simulate the Turing machine while Player I also has markers to ensure that Player O correctly marks errors. Altogether, we obtain a delay game that simulates an alternating doubly-exponential space Turing machine, and A2ExpSpace = 3ExpTime yields the desired lower bound.

Theorem 5.2. *Solving delay games with* Emerson-Lei-wNMA *winning conditions is* 3ExpTime*-complete.*

Proof. The upper bound was shown in Theorem 3.4, we show the corresponding lower bound.

We give a reduction from the word problem for alternating doubly-exponential space Turing machines. Since 3ExpTime = A2ExpSpace [5], we obtain the desired result.

Let $\mathcal{M} = (Q = Q_\exists \uplus Q_\forall, \Sigma, q_I, \Delta, q_A, q_R)$ be an alternating doubly-exponential space Turing machine, where $\Delta \subseteq Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, 1\}$. Furthermore, let p be a polynomial such that 2^{2^p} bounds the space consumption of $\mathcal{M}$ and let $w \in \Sigma^*$ be an input word. We fix $n = p(|w|)$. Wlog., we assume that the accepting state q_A and the rejecting state q_R are equipped with a self-loop. Furthermore, wlog., we assume that either q_A or q_R is reached in every run of $\mathcal{M}$ on w. We have explained in the proof of Theorem 4.2 that such an assumption can be made.

We construct an Emerson-Lei-wNMA $\mathfrak{A}$ of polynomial size in $(|\Delta| + n)$ such that $\mathcal{M}$ accepts w if, and only if, Player O wins $\Gamma_f(L(\mathfrak{A}))$ for some delay function f.

As in the proof of Theorem 4.2, the idea is that, in the delay game, the players build a run of $\mathcal{M}$ on w in the form of a configuration sequence. Player I controls the universal states and Player O the existential ones. Furthermore, Player I spells all configurations with his moves.

The configurations are of doubly-exponential size in n, so each cell of $\mathcal{M}$'s tape is addressed by a superblock as introduced in the proof of Theorem 5.1.

In between configurations there is a delimiter, either C or N, to denote if the next configuration is a *copy* of the previous one or a *new* one. The need to copy configurations arises since, in the delay game, Player O is behind with her moves, so Player I needs to wait for Player O's pick if the current configuration is existential.

We go into details. The automaton we construct uses the product alphabet

$$(\{0, 1, C, N, \lrcorner, \times, \#\} \cup Q \cup \Sigma) \times (\{0, 1, \#, ✱, ✖, ✓, ✗\} \cup \Delta)\,.$$

Recall that Q, Δ, and Σ are components of the Turing machine $\mathcal{M}$ and are used to encode configurations and to pick transitions to apply. We first describe what we call valid input resp. output formats:

- An input sequence α is valid if it is of the form

$$\Big((C + N)\big[S(\Sigma + Q + \lrcorner)\big]^+\Big)^\omega\,,$$

  where S stands for superblock and is defined as

$$S = \big((\times + \#)(0 + 1)^{n+1}\big)^+\,.$$

– The sequence α is to be interpreted as a sequence of configurations

$$(C \ + \ N)c_0(C \ + \ N)c_1 \cdots$$

where each c_i is a configuration delimited by C or N. Note that the correct encoding of such configurations is enforced by the rules of the game.

– In a configuration encoding, a superblock defines the number of a cell, the letter after the superblock is either the tape content (that is, a letter $\sigma \in \Sigma$, or a blank (denoted as $\lrcorner$)) or a state $q \in Q$.

– Recall that a superblock S is to be interpreted as first a marker ✕ or # to mark (the position before) an address block inside a superblock, then an n-bit address block encoding an address a, followed by a special bit b. Then again, a marker, an address block, a special bit, a marker, and so on.

– The sequence of special bits of a superblock defines the number of the cell.

– The marker ✕ can be used once in the input sequence to mark (the position before) an address inside a superblock. The purpose is explained further below.

– The letter # is to be interpreted as unmarked.

- An output sequence β is valid if it is of the form

$$\Big(\Delta\big[(\text{✻} + \text{✖} + \#)(\text{✓} + \text{✗})^n(0+1)\big]^+\Big)^\omega,$$

and if for every position i, $\alpha(i) \in \{C, N\}$ if, and only if, $\beta(i) \in \Delta$, i.e., the letters $\tau \in \Delta$ occur at the same positions as the configuration delimiters C and N. We give the intended interpretation of this format, which is detailed below.

– The markers ✓ resp. ✗ are intended to indicate the absence of resp. the existence of an error an address block.

– If ✗ marks a bit in an address block, it should indicate an error in updating the address block at this bit.

– If ✻ marks (the position before) an address block, say the block encodes address a, then it should indicate that there is an error in updating the special bit with address a in the next superblock.

– Finally, if ✖ marks (the position before) an address block, then it should indicate that there is a copy or update mistake in the successive configuration.

The behavior of the automaton is as follows. We first describe the occurrence of simple format errors that lead to acceptance resp. rejection:

1. If there exists some i such that $\alpha(0)\alpha(1)\cdots\alpha(i)$ cannot be completed to have a valid format, but $\beta(0)\beta(1)\cdots\beta(i-1)$ can be completed to have a valid format, the automaton accepts.

2. If a configuration encoding c contains not exactly one state or the first address block in a superblock is not zero, or the first superblock of a configuration does not encode the number zero, the automaton accepts.

3. If c_0 does not encode the initial configuration on w, it accepts.

4. If there exists some i such that $\beta(0)\beta(1)\cdots\beta(i)$ cannot be completed to have a valid format, but $\alpha(0)\alpha(1)\cdots\alpha(i)$ can be completed to have a valid format, the automaton rejects.

Items 1 to 4 can be handled similar to Items 1 to 4 in the proof of Theorem 4.2. One needs automata of polynomial size in $(|\Delta| + n)$ to do so.

We now describe different kinds of (more involved) errors and how they are handled:

5. If the k-th bit of an address block is marked with ✗, then it is verified if the k-th bit of the next address block is correctly updated. If the update is faulty, the automaton accepts, if the update is correct, the automaton rejects.

This item can be handled as Item 5 in the proof of Theorem 4.2 with $\mathcal{O}(n)$ states.

6. If ✱ marks an address a in a superblock, then the automaton checks whether the special bit that is addressed by a in the next superblock has been correctly updated. If an error has been correctly claimed, the automaton accepts, otherwise it rejects.

To achieve this behavior, a gadget as introduced in Example 2.1 is needed that stores the address marked by ✱. To determine how the value of the special bit addressed by the marked address changes it suffices to check whether all special bits (in the same superblock) to the right are one. If at least one zero is seen, the value of the special bit does not flip. Otherwise it does. In the next superblock, the automaton has to find the special bit that has the same address. It has to guess when it has advanced to this address block and verify that the guess is correct. Therefore, we need another gadget as introduced in Example 2.1 to store the guessed bit and compare via a formula that the two stored addresses are equal. The gadgets need $\mathcal{O}(n)$ many states, checking whether the special bit is correctly updated can be done with a constant number of states.

7. If the ✖ marker occurs in a superblock S, then the automaton checks whether there is a *copy* or *update error* between successive configurations that manifests in the cell whose number is encoded by S. If an error had been correctly claimed, the automaton accepts, otherwise it rejects.

We have detailed in the proof of Theorem 4.2 (regarding Item 6) how a copy or update error is checked when the relevant pieces of information are known (which requires $\mathcal{O}(|\Delta| + n)$ states). The difficulty lies in finding the same cell in the current and in the successor configuration. Therefore, the automaton guesses at the beginning of a superblock, say S, whether it has some address that is marked with ✖. (A wrong guess leads to rejection.) If the superblock S contains a ✖, then the automaton enforces that, from then on, the output must copy the special bits of S ad infinitum. This condition makes it easy to find the cell addressed by S in the next configuration, because it suffices to find the cell addressed by a superblock such that the special bits in the input and output sequence are equal.

We have described in detail in the proof of Theorem 5.1 (see Item 5) how to enforce that a bit sequence is faithfully copied. We use the same technique here, which needs $\mathcal{O}(n)$ states. To recap, in case of an error, the input sequence marks an address of a special bit with ✕. This address is where a copy error has manifested.

The last condition is concerned with whether w is accepted by $\mathcal{M}$:

8. If the format is valid and no error occurred so far: If a configuration with q_A resp. q_R is seen, the automaton accepts resp. rejects.

This can be handled as Item 7 in the proof of Theorem 4.2 with a constant number of states.

All in all, the automaton $\mathfrak{A}$ needs a polynomial number of states (in $|\Delta| + n$).

Finally, we are ready to describe the acceptance condition of $\mathfrak{A}$ as a formula φ defined as

$$\neg q_{rej} \wedge \neg q_R \wedge (q_{acc} \vee q_A \vee \varphi_{✱} \vee \varphi_{✖}) .$$

The formula $\varphi_{✱}$ is satisfied if the gadgets to store two addresses needed to find an update error regarding a special bit have not been entered (no special bit error is claimed) or the stored addresses are equal.

The formula $\varphi_{✖}$ is satisfied if the gadget to check faithful copying is either not entered or the gadget is entered and indicates that copying is successful. There are two reasons why the copy gadget is not entered. The marker ✖ is not used in the output, i.e., no configuration copy or update error is claimed. Or, ✖ is used, but the marker ✕ is not used in the input, i.e., no error in copying the special bits is claimed.

The size of φ is polynomial in $(|\Delta| + n)$.

It is left to show that $\mathcal{M}$ accepts w if, and only if, Player O wins $\Gamma_f(L(\mathfrak{A}))$ for some delay function f. To begin with, assume $\mathcal{M}$ accepts w. Let f be a constant delay function such that $f(0) = m$, where m is chosen such that Player O has enough lookahead to see a full configuration before she has to start

with her output. This means m is triply-exponential in n. We show that Player O wins $\Gamma_f(L(\mathfrak{A}))$. We assume that either player does not make simple errors (recall that means playing in the wrong format). Firstly, assume that Player I either does not update an address block correctly, does not update a special bit in a superblock correctly, or introduces a configuration copy or update error. The lookahead is large enough for Player O to correctly claim such an error. The formula is then satisfied. Secondly, assume that Player I does not introduce such errors. Since $\mathcal{M}$ accepts w, Player O can pick existential transitions such that if Player I applies the picked transitions an accepting configuration will be reached. If an accepting configuration is seen, the formula φ is clearly satisfied. Player I can prevent that an accepting configuration is seen by always copying the current configuration. But then, the formula φ is also satisfied, because the sub-formulas φ_{*} and $\varphi_{\text{✖}}$ are satisfied if no error of any kind is claimed. Player O does need to claim an error, as we assumed that Player I does not introduce an error.

For the other direction, assume $\mathcal{M}$ rejects w. We show that Player I wins $\Gamma_g(L(\mathfrak{A}))$ for every delay function g. Again, we assume that either player does not make simple format errors. No matter the existential transitions that Player O chooses, Player I can always pick universal transitions such that eventually a rejecting configuration is reached. Hence, we assume that Player I advances the run by giving new configurations according to his and Player O's choices. A more detailed description what it means to advance a run is given in the proof of Theorem 4.2.

Furthermore, we assume that Player I does not introduce any kind of errors as this is not beneficial for him. Player O loses a play where a rejecting configuration is reached. Hence, her only chance is to claim an error. Since there are no errors introduced by Player I, this approach is not fruitful as this consequently falsifies φ. Player O cannot win a play, i.e., Player I wins $\Gamma_g(L(\mathfrak{A}))$. □

6 Conclusion

We took the first step towards investigating delay games with Muller conditions by showing doubly- and triply-exponential bounds for deterministic and non-deterministic delay games with weak Muller conditions. More, specifically we have shown that solving such games is 2ExpTime-complete (for deterministic automata) and 3ExpTime-complete (for non-deterministic automata) and that doubly-exponential (for deterministic automata) and triply-exponential lookahead (for non-deterministic automata) is necessary and sufficient. These results have to be compared to those for deterministic safety and parity automata, for which solving delay games is ExpTime-complete and exponential lookahead is necessary and sufficient. Similarly, for non-deterministic safety and parity automata solving delay games is 2ExpTime-complete and doubly-exponential lookahead is necessary and sufficient. Thus, the succinctness of weak Muller conditions (encoded by formulas) yields an exponential increase in comparison to both safety and parity conditions.

There are two immediate directions for further research: Try to lift our results to (standard) Muller conditions and to strengthen the lower bounds by proving them for less succinct representations of $\mathcal{F}$.

Recall that our lower bounds for weak Muller conditions rely on gadgets comparing n-bit strings for some fixed n. This is possible for a fixed number of comparisons by having one gadget for each string to be compared and then using the formula defining $\mathcal{F}$ to implement the actual comparison. A natural way to lift, say, the bad j-pair lower bound game is to just play this game infinitely often. However, this requires to reuse the gadgets infinitely often, which then means one cannot compare strings from one iteration of the bad j-pair game, but compares strings from all iterations.

Let us stress again that all our upper bounds are independent of the representation of the acceptance condition. However, our lower bounds only hold for Emerson-Lei conditions, which is the most succinct representation. Strengthening the lower bounds via less succinct representations most likely requires a new approach, as all lower bounds we have proven require the comparison of n-bit strings. This can be done with a small formula, as shown here, but not with the less succinct presentations considered by Hunter and Dawar for delay-free games [14, 13]. In particular, it is even open whether delay games with explicit Muller conditions (the least succinct representation) are harder than delay games with parity or safety conditions. Note that in the delay-free case, explicit Muller games can be solved in polynomial time [11].

References

[1] Udi Boker. On the (in)succinctness of Muller automata. In Valentin Goranko and Mads Dam, editors, *CSL 2017*, volume 82 of *LIPIcs*, pages 12:1–12:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[2] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. Amer. Math. Soc.*, 138:pp. 295–311, 1969.

[3] Cristian S. Calude, Sanjay Jain, Bakhadyr Khoussainov, Wei Li, and Frank Stephan. Deciding parity games in quasipolynomial time. In Hamed Hatami, Pierre McKenzie, and Valerie King, editors, *STOC 2017*, pages 252–263. ACM, 2017.

[4] Arnaud Carayol and Christof Löding. Uniformization in automata theory. In *Logic, Methodology and Philosophy of Science - Proceedings of the 14th International Congress*. College Publications, 2015.

[5] Ashok K. Chandra, Dexter Kozen, and Larry J. Stockmeyer. Alternation. *J. ACM*, 28(1):114–133, 1981.

[6] Mingshuai Chen, Martin Fränzle, Yangjia Li, Peter Nazier Mosaad, and Naijun Zhan. Indecision and delays are the parents of failure - taming them algorithmically by synthesizing delay-resilient control. *Acta Informatica*, 58(5):497–528, 2021.

[7] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29(1):132–158, 1999.

[8] Emmanuel Filiot and Sarah Winter. Synthesizing computable functions from rational specifications over infinite words. In Mikolaj Bojanczyk and Chandra Chekuri, editors, *FSTTCS 2021*, volume 213 of *LIPIcs*, pages 43:1–43:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[9] David Gale and Frank M. Stewart. Infinite games with perfect information. *Annals of Mathematics*, 28:245–266, 1953.

[10] Michael Holtmann, Lukasz Kaiser, and Wolfgang Thomas. Degrees of lookahead in regular infinite games. *Log. Methods Comput. Sci.*, 8(3), 2012.

[11] Florian Horn. Explicit Muller games are PTIME. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS 2008*, volume 2 of *LIPIcs*, pages 235–243. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2008.

[12] Frederick A. Hosch and Lawrence H. Landweber. Finite delay solutions for sequential conditions. In Maurice Nivat, editor, *ICALP 1972*, pages 45–60. North-Holland, Amsterdam, 1972.

[13] Paul Hunter. *Complexity and Infinite Games on Finite Graphs*. PhD thesis, Computer Laboratory, University of Cambridge, 2007.

[14] Paul Hunter and Anuj Dawar. Complexity bounds for regular games. In Joanna Jedrzejowicz and Andrzej Szepietowski, editors, *MFCS 2005*, volume 3618 of *LNCS*, pages 495–506. Springer, 2005.

[15] Felix Klein and Martin Zimmermann. What are strategies in delay games? Borel determinacy for games with lookahead. In Stephan Kreutzer, editor, *CSL 2015*, volume 41 of *LIPIcs*, pages 519–533. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[16] Felix Klein and Martin Zimmermann. How much lookahead is needed to win infinite games? *Log. Methods Comput. Sci.*, 12(3), 2016.

[17] Felix Klein and Martin Zimmermann. Prompt delay. In Akash Lal, S. Akshay, Saket Saurabh, and Sandeep Sen, editors, *FSTTCS 2016*, volume 65 of *LIPIcs*, pages 43:1–43:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.