RWTH Aachen

Department of Computer Science
Technical Report

Strategy Machines and their Complexity

Marcus Gelderie

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2012-04

RWTH Aachen · Department of Computer Science · March 2012

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Strategy Machines and their Complexity

Marcus Gelderie

RWTH Aachen, Lehrstuhl für Informatik 7,
Logic and Theory of Discrete Systems,
D-52056 Aachen
gelderie@automata.rwth-aachen.de

Abstract. We introduce a machine model for the execution of strategies in (regular) infinite games that refines the standard model of Mealy automata. This model of controllers is formalized in the terminological framework of Turing machines. We show how polynomially sized controllers can be found for Muller and Streett games. We are able to distinguish aspects of executing strategies (”size”, ”latency”, ”space consumption”) that are not visible in Mealy automata. We show upper and lower bounds for these parameters for several classes of ω-regular games.

1 Introduction

Strategies obtained from ω-regular games are used to obtain controllers for reactive systems. The controllers obtained in this way are transition systems with output, also called Mealy machines. The physical implementation of such abstract transition systems raises interesting questions, starting with the observation that the naive implementation of a Mealy machine by a physical machine (such as a circuit or a register machine) essentially amounts to encoding a large case distinction based on the current input and state. This approach entails considerable complexity challenges [BGJ$^+$07], as it essentially preserves the size of the underlying Mealy machine. These machines are known to be large in general [DJW97]. The complexity of solving ω-regular games reflects this problem [HD05,HD08,Hor08,EJ99]. Reducing the size of a given Mealy machine has been investigated [HL07,GH11] but must ultimately obey the general bounds mentioned. These observations indicate that pursuing a direct approach, without the detour via Mealy machines, in synthesizing controllers may be worthwhile. We propose a new model for reasoning about such physical machines and their synthesis. This model, called a *strategy machine*, is based on an appropriate format of Turing machines. The concept of a Turing machine is widely used in theoretical scenarios to model computational systems, such as [GHS09]. A strategy machine is a multi-tape Turing machine which has two distinct tapes, one for input and output and another for storing information from one computation to the next. Referring to a given game graph, the code of an input vertex appears on the IO-tape and an output is produced. The process is then repeated. This model introduces new criteria for evaluating a strategy. For example, it now makes sense to investigate the number of steps required to transform an input into the corresponding output, called the *latency*. Likewise we may ask how much information needs to be stored from the computation of one output to the computation of the next output. Note that translating a Mealy machine into this model entails the problems discussed above and yields a machine with size (the number of control states) roughly equal to the size of the Mealy machine.

Introducing the model of a strategy machine, we present initial results for two classes of ω-regular games, namely Muller and Streett games. Building on Zielonka's algorithm [Zie98], we show that in both cases we may construct strategy machines implementing a winning strategy of an exponentially lower size than any Mealy machine winning strategy: Both the size of the strategy machine and the amount of information stored on the tape are bounded polynomially in the size of the arena and of the winning condition (given by a propositional formula ϕ or a set of Streett pairs). Moreover, for Streett games even the latency is bounded polynomially in these parameters. For Muller games the latency is linear in the size of the enumerative representation of the winning condition. We also present lower bounds for the latency and space requirement by translating some well known results from the theory of Mealy machine strategies to our model. This yields lower bounds for Muller, Streett, and LTL games.

In related work Madhusudan [Mad11] considers the synthesis of reactive programs over Boolean variables. A machine executing such a program falls into the model discussed above. The size of the program then loosely corresponds to the number of control states. In the same way, the requirement of tape cells loosely corresponds to the number of Boolean variables. The latency and space requirement are not studied in [Mad11].

The paper is structured as follows. First, we give a formal introduction of the Turing machine model mentioned above. We formally define the parameters latency, space requirement, and size. Next we recall some elementary concepts of the theory of infinite two player zero-sum games with ω-regular winning conditions. We show lower bounds for the latency and space requirement of machines implementing winning strategies in Muller, Streett and LTL games. Then we develop an adaptive algorithm for Muller games. This algorithm is based on Zielonka's construction [Zie98] using some ideas from [DJW97]. The latency and space consumption strongly depend on the way the winning condition is given. We illustrate this fact by showing how the algorithm can be used to obtain an efficient controller – that is, one with latency, size and space requirement bounded polynomially in the size of the arena and the winning condition. If the winning condition is a Streett condition, this is particularly promising, because Streett conditions are more succinct than explicit Muller conditions.

2 Strategy Machines – A Formal Model

The intuition of a *strategy*[1] *machine* is that of a "black box" which receives an input (a bit-string), does some internal computation and at some point produces an output. It then receives the next input and so forth. We will refer to such a sequence of steps – receive an input, compute, produce an output – as an *iteration*. In general it is allowed for such a machine to retain some part of its current internal configuration from one iteration to the next. However, it need not do so. Also, the amount of information (how this is quantified will be discussed shortly) is a priori not subject to any restriction. In particular, a strategy machine may require an ever growing amount of memory, increasing from one iteration to the next, to store this information.

[1] Strategies are not formally defined until the next section. The reader may want to skip ahead and read the basic terminology on games if a question arises.

Our model of a strategy machine is a deterministic $(k+2)$-tape Turing machine $\mathcal{M}$, $k \in \mathbb{N} = \{1, 2, \ldots\}$. The tapes have the following purpose. The first is a designated *IO-tape*, responsible for input to and output from the machine. A bit-string $w \in \mathbb{B}^*$, $\mathbb{B} = \{0, 1\}$, is the content of the IO-tape at the beginning of an iteration. The machine $\mathcal{M}$ is also in a designated *input-state* at this point. Next $\mathcal{M}$ performs some computation in order to produce an output. During this computation the remaining $k+1$ tapes may be used. We first discuss the *k computation tapes*. As the name suggests, these tapes are used – as in any Turing machine – to store all the data needed for the computation of the output. The content of the computation tapes is deleted immediately before a new iteration begins. In particular, they cannot be used to store information from one iteration to the next. Storing such information is the purpose of the *memory tape*. Its content is still available during the next iteration. In order to produce an output $\mathcal{M}$ will write this output, another bit-string, on the IO-tape. Then it will enter a designated *output-state*. Let $\hat{\mathbb{B}} = \mathbb{B} \uplus \{\#\}$. We define:

Definition 1 (k-tape Strategy Machine). *A* strategy machine *is a deterministic $(k+2)$-tape Turing machine $\mathcal{M} = (Q, \mathbb{B}, \hat{\mathbb{B}}, q_I, q_O, \delta)$ with two designated states q_I and q_O, called the* input-state *and the* output-state *of $\mathcal{M}$ respectively. We require the partial function $\delta\colon Q \times \hat{\mathbb{B}}^{k+2} \dashrightarrow Q \times (\hat{\mathbb{B}} \times \{\leftarrow, \downarrow, \rightarrow\})^{k+2}$ to be undefined for all pairs $(q_O, b_1, b_2, b_3) \in Q \times \hat{\mathbb{B}}^3$. Furthermore we require that no transition leads into q_I. The tapes of $\mathcal{M}$ are denoted t_{IO}, $t_{com}^{(i)}$, $1 \leq i \leq k$, t_{mem}, and are called the* input-output-tape (IO-tape)*, the* computation-tapes *and the* memory-tape *respectively.*

In the following paragraphs we assume $k = 1$ to simplify the notation. If $k = 1$ we simply write t_{com} for $t_{com}^{(1)}$.

Given a strategy machine $\mathcal{M}$ we denote its *size* by $\|\mathcal{M}\| = |Q| - 2$. The size is the number of states not counting the input and output state. A *configuration* c of $\mathcal{M}$ comprises the current state $q(c)$, the contents of the IO-, computation- and memory-tapes, $t_{IO}(c)$, $t_{com}(c)$ and $t_{mem}(c)$ as well as the current head positions $h_{IO}(c)$, $h_{com}(c)$ and $h_{mem}(c)$ on the respective tapes. It is defined as a tuple

$$(q(c), t_{IO}(c), t_{com}(c), t_{mem}(c), h_{IO}(c), h_{com}(c), h_{mem}(c)) \in Q \times (\hat{\mathbb{B}}^*)^3 \times \mathbb{Z}^3$$

The successor relation on configurations is defined as usual and denoted $\vdash$, its transitive closure is denoted by $\vdash^*$. An *iteration* of $\mathcal{M}$ is a sequence $c_1, \ldots, c_l$ of configurations, such that for $1 \leq i \leq l-1$ we have $c_i \vdash c_{i+1}$. It starts with $c_1 = (q_I, x, \varepsilon, w_{mem}, 0, 0, 0)$ and ends with $c_l = (q_O, y, w, w'_{mem}, h, h', h'')$ for some elements $x, y \in \mathbb{B}^*$, arbitrary words $w, w_{mem}, w'_{mem} \in \hat{\mathbb{B}}^*$ and integers $h, h', h'' \in \mathbb{Z}$. We denote iterations by pairs of configurations (c, c') where the state in c is q_I and that in c' is q_O. Since $\mathcal{M}$ is deterministic there exists at most one iteration leading from c to c' (since q_O has no outgoing transitions and q_I has no incoming transitions). If no such iteration exists, we say the pair (c, c') is an *illegal iteration*. Otherwise it is a *legal iteration*.

Given a word $u = x_1 \cdots x_n \in A^*$, where $A = \mathbb{B}^*$, a *run* of $\mathcal{M}$ on u is a sequence of legal iterations $(c_1, c'_1), \ldots, (c_n, c'_n)$, such that $t_{IO}(c_i) = x_i$ and $t_{mem}(c'_i) = t_{mem}(c_{i+1})$. By convention $t_{mem}(c_1) = \varepsilon$. Note that by the definition of an iteration $t_{com}(c_i) = \varepsilon$ and $h_{com}(c_i) = h_{IO}(c_i) = h_{mem}(c_i) = 0$. A strategy machine $\mathcal{M}$ defines a function $f\colon A \to A$ where $f(u) = t_{IO}(c'_n)$. By extension it defines a function $f_{\mathcal{M}}\colon A^*A \to A$, the *function implemented by* $\mathcal{M}$.

The *latency* $T(c, c')$ of a legal iteration (c, c') is the number of configurations on the unique path from c to c'. If the latency of all iterations in any run is bounded by a constant, we define the latency $T(\mathcal{M})$ of $\mathcal{M}$ to be the maximal latency over all such (legal) iterations. Finally we define the *space requirement* $S(c, c')$ of a legal iteration (c, c') to be the number of tape cells of t_{mem} visited during that iteration. Again the space consumption $S(\mathcal{M})$ of $\mathcal{M}$ is the maximum over all space consumptions, if such a maximum exists. Note that both latency and space consumption refer to quantities needed to execute a single iteration between reading $a \in A$ and outputting $b \in A$. In other words they capture the resources required to execute one iteration.

Recall that a Mealy machine is a quintuple $\mathfrak{M} = (M, \Sigma, m_0, \delta, \tau)$ with *states* M, *input/output alphabet* Σ, *initial state* m_0, *transition function* $\delta\colon M \times \Sigma \to M$ and *output function* $\tau\colon M \times \Sigma \to \Sigma$. There is a straightforward way to transform a Mealy machine $\mathfrak{M}$ into an equivalent strategy machine (that is one, which implements the same function). We take an arbitrary encoding $e\colon M \to \mathbb{B}^*$ which allows us to write elements from M onto the tape of a Turing machine. Note that we need $\lceil \log_2(|M|) \rceil$ bits to do so. Likewise we use an encoding $e'\colon \Sigma \to \mathbb{B}^*$. Then our machine $\mathcal{M}_{\mathfrak{M}}$ proceeds as follows. On the memory tape it maintains $e(m)$ where m is the current state of $\mathfrak{M}$. The program has a copy of the table $\langle (m, x, \delta(m, x), \tau(m, x)) \rangle_{(m,x) \in M \times \Sigma}$ in its state space. For efficient lookup the table can be represented as a binary search tree indexed by $e(m) \cdot e'(x) \in \mathbb{B}^*$. Thus traversing the tree requires time $\mathcal{O}(\log_2(|M|) + \log_2(|\Sigma|))$. The tree itself has size $\mathcal{O}(|M| \cdot |\Sigma|)$. The memory update requires at most $\log_2(|M|)$ steps. So we get:

Proposition 1. *For every Mealy machine $\mathfrak{M} = (M, \Sigma, m_0, \delta, \tau)$ there exists an equivalent 1-tape strategy machine $\mathcal{M}_{\mathfrak{M}}$ with size $\|\mathcal{M}_{\mathfrak{M}}\| \in \mathcal{O}(|M| \cdot |\Sigma|)$, space requirement $S(\mathcal{M}_{\mathfrak{M}}) \in \mathcal{O}(\log_2(|M|))$ and latency $T(\mathcal{M}_{\mathfrak{M}}) \in \mathcal{O}(\log_2(|M|) + \log_2(|\Sigma|))$.*

3 Basics on Games

In this section we briefly recall the most basic facts on ω-regular games. We assume the reader is familiar with these concepts. An introduction to the theory can be found in e.g. [GTW02,Löd11,PP04].

An *arena* is a directed graph $\mathcal{A} = (V, E)$ with the property that every vertex has an outgoing edge. We assume that there is a partition $V = V_0 \uplus V_1$ of the vertex set. A *subarena* is an induced subgraph $\mathcal{A} \restriction V'$ obtained by taking a subset $V' \subseteq V$ and considering the induced subgraph (restricting $E' = E \cap V' \times V'$). We tacitly require V' to be such that every vertex $v \in V'$ has a neighbor in V'. An i-trap is a subset $V' \subseteq V$, such that for every $v \in V_i$ one has $vE \subseteq V'$ and for all $v \in V_{1-i}$ one has $vE \cap V' \neq \emptyset$. An i-trap induces a subarena.

A *infinite two player game* (in this paper simply called a *game*) is a tuple $\mathbf{G} = (\mathcal{A}, \varphi)$ with an arena $\mathcal{A}$ and a *winning condition* $\varphi \subseteq V^\omega$. There are two players, called player 0 and player 1. Given an initial vertex $v_0 \in V$ they proceed as follows. If $v_0 \in V_0$ player 0 chooses a vertex v_1 in the *neighborhood* vE of v. Otherwise $v_0 \in V_1$ and player 1 chooses a neighbor v_1. The play then proceeds in the same fashion from the new vertex v_1. In this way the two players create an infinite sequence $\pi = \pi(0)\pi(1)\cdots = v_0 v_1 \cdots \in V^\omega$ of adjacent vertices $(\pi(i), \pi(i+1)) \in E$ called a *play*. Player 0 wins the play π if $\pi \in \varphi$. Otherwise player 1 wins. $\mathbf{G}$ is called *ω-regular* if φ is an ω-regular set. All games considered in this paper are ω-regular.

A *strategy* for player i is a mapping $\sigma\colon V^* V_i \to V$ assigning an element from vE to each string $w \in V^+$ with $\operatorname{last}(w) = v$ (where $\operatorname{last}(\cdot)$ denotes the last element of a sequence). π is *consistent* with σ if for every $n \in \mathbb{N}$ with $\pi(n) \in V_i$ we have $\pi(i+1) = \sigma(\pi(0)\cdots\pi(n))$. σ is a *winning strategy* for player i is every play consistent with π is won by player i.

The *winning region* of player i, written $\mathcal{W}_i$, is the set of vertices $v \in V$, such that player i has a winning strategy σ_v from v. It can be shown that in ω-regular games $V = \mathcal{W}_0 \uplus \mathcal{W}_1$, i.e. from any given vertex either one player has a winning strategy or the other one does. If $v \in \mathcal{W}_i$ we say *player i wins from v.*

One usually does not specify the winning condition φ directly. Instead there are several *types* of winning conditions. In this paper we are concerned with only three of them: Muller, Streett and LTL conditions. We describe each of them in turn.

A *Muller condition* is given by a set propositional formulas ϕ with variables V. A play π is won by player 0, if ϕ if the *infinity set* $\operatorname{Inf}(\pi) = \{v \in V \mid \forall n \exists m \geq n : \ \pi(m) = v\}$ is a model of ϕ, i.e. $\operatorname{Inf}(\pi) \models \phi$. In this situation we also say π *satisfies* ϕ. Equivalently Muller conditions are often defined as a set $\mathfrak{F} \subseteq \mathscr{P}(V)$ of subsets of V. Conditions given by such a set $\mathfrak{F}$ are called *explicit* Muller conditions. The two formalisms are equivalent. However, the first can be exponentially more succinct than the second. A game $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ with a Muller condition $\mathfrak{F}$ is called a *Muller game.*

A *Streett condition* is given by a set $\Omega = \{(R_1, G_1), \ldots, (R_k, G_k)\}$ of pairs of sets $R_i, G_i \subseteq V$. A set $X \subseteq V$ *violates* a Streett pair $(R, G) \in \Omega$ if $R \cap X \neq \emptyset$ but $G \cap X = \emptyset$. A play π *violates* (R, G) if $\operatorname{Inf}(\pi)$ violates (R, G). If π does not violate any pair $(R, G) \in \Omega$, then π *satisfies* Ω and is won by player 0. Otherwise there exists a pair which is violated and player 1 wins. A game $\mathbf{G} = (\mathcal{A}, \Omega)$ with a Streett condition Ω is called a *Streett game.*

Finally, an *LTL condition* is one where the set of winning plays for player 0, φ, is given by an LTL-formula[2]. More precisely, if V is the set of vertices of the arena and if ϕ is an LTL formula over the propositions V, we have $\varphi = \{\pi \in V^\omega \mid \pi \models \phi\}$. A game with an LTL condition is called an *LTL game.*

The last concept we would like to briefly recall is that of an *attractor*. Let $\mathcal{A} = (V, E)$ be an arena, $i \in \mathbb{B}$ and let $S \subseteq V$ be a set. Let $A_0^{(i)} = S$ and define

$$A_{k+1}^{(i)} = \{v \in V_i \mid vE \cap A_k^{(i)} \neq \emptyset\} \cup \{v \in V_{1-i} \mid vE \subseteq A_k^{(i)}\} \cup A_k^{(i)}$$

The set $\operatorname{Attr}_i^{\mathcal{A}}(S) = \bigcup_k A_k^{(i)}$ is called the *i-attractor on S*. It is the set of vertices from which player i can enforce to visit S. A corresponding strategy is called an *attractor strategy* (see [GTW02,Löd11]). In the following section we will use the fact that an attractor can be computed in time $\boldsymbol{\mathcal{O}}(|V| \cdot |E| \cdot \log_2(|V|))$ by a two-tape Turing machine. The details are explained in the appendix. Indeed:

---

[2] To describe the syntax and semantics of *linear temporal logic* (LTL) is beyond the scope of this paper. We refer the reader to [GTW02,Löd11] for more information on LTL and games with LTL winning conditions.

Remark 1. All complexity considerations in this paper are subject to the precise assumptions one makes about representing data structures in the Turing machine domain. We make all of those assumptions precise in the appendix. There we also explain how one can obtain the claimed complexity results.

4 Lower Bounds

We would like to address the question of lower bounds on the space requirement and latency of any machine implementing a winning strategy for player 0 in certain classes of games. The following proposition will be useful:

Proposition 2. *Let $\mathcal{M}$ be a strategy machine with a space requirement of $n \in \mathbb{N}$. Then the strategy implemented by $\mathcal{M}$ can be implemented by a Mealy machine with 2^n states*[3].

Proof. It suffices to observe that the only information retained from one iteration to the next is the content of the memory tape. If the bound on the space requirement is n then there can be at most 2^n different configurations of the memory tape. Any legal computation (c, c') is completely determined by the content $t_{mem}(c)$ of the memory tape and the input $t_{IO}(c)$. Hence a Mealy machine with state space $\mathbb{B}^n$ and transitions $\delta(w, v) = w'$, $\tau(w, v) = v'$ for $w = t_{mem}(c)$, $w' = t_{mem}(c')$ and $v = t_{IO}(c)$, $v' = t_{IO}(c')$ will simulate $\mathcal{M}$, i.e. it will compute the same function. □

Let $f\colon \mathbb{N} \to \mathbb{N}$. Let $(\mathbf{G}_n)_{n\geq 0}$ be a family of games with $\mathbf{G}_n = ((V_n, E_n), \varphi_n)$ and $|V_n| \in \boldsymbol{\mathcal{O}}(n)$, such that no Mealy machine with less than $f(n)$ states implements a winning strategy for player 0 in $\mathbf{G}_n$. Then we say the family $(\mathbf{G}_n)_{n\geq 0}$ is *f-hard*. By extension we say that a class $\mathcal{C}$ of games is *f-hard* if there exists an f-hard family in $\mathcal{C}$. Such a family is called a *witnessing family*.

Note that the definition of hardness above does not bound the size of the underlying winning condition. For important classes of games we can bound this size linearly in n:

Proposition 3. *The classes of Muller games, given by a propositional formula, and the class of Streett games, given by a set of Streett pairs, are 2^n-hard. The conditions (propositional formulas, set of Streett pairs) of the witnessing families are no larger than $\boldsymbol{\mathcal{O}}(|V|)$.*

Note that Muller games are even $(\frac{n}{2})!$-hard ([DJW97]). However, in this case the size of the winning condition is not linear in the size of the graph.

Let $\mathcal{C}$ be a class of games. A *(solution) scheme for $\mathcal{C}$* is a mapping $\mathcal{S}$ assigning a strategy machine $\mathcal{S}(\mathbf{G})$ to every game $\mathbf{G} \in \mathcal{C}$, such that $\mathcal{S}(\mathbf{G})$ implements a winning strategy for player 0 in $\mathbf{G}$.

Lemma 1. *Let $f, g\colon \mathbb{R}_+ \to \mathbb{R}_+$, such that f is strictly monotone and differentiable with $f'(x) \neq 0$ for all $x \in \mathbb{R}_+$. Let $\mathcal{C}$ be a $2^{\boldsymbol{\Omega}(f(n))}$-hard class of games. Assume for some $q \in (0, 1)$ we have $g(x) \in \boldsymbol{\mathcal{O}}(f(x)^q)$. Then there can be no solution scheme $\mathcal{S}$ for $\mathcal{C}$, such that $\mathcal{S}(\mathbf{G})$ has space requirement in $\boldsymbol{\mathcal{O}}\big(g(m)\big)$ where $m = |V|$.*

[3] For simplicity we assume that on the memory tape only the symbols 0 and 1 are used and # is reserved for the unvisited part of the tape. If # is also allowed, some minor modifications to the proofs that follow are necessary. The underlying ideas, however, are the same.

Proof. We may choose a $2^{\mathbf{\Omega}(f(n))}$-hard family $(\mathbf{G}_n)_{n\geq 0}$ of games in $\mathcal{C}$. Choose $n_0 \in \mathbb{N}$ large enough and $\alpha, \beta, \gamma \in \mathbb{R}_+$ such that $(\mathbf{G}_n)_{n\geq n_0}$ is $(\alpha \cdot 2^f)$-hard and $g(x) \leq \beta \cdot f(x)^q$ for all $x \geq n_0$ and $|V_n| \leq \gamma \cdot n$ for all $n \geq n_0$. Up to choosing the family $(\mathbf{G}_n)_{n\geq n_0}$ we may assume $n_0 = 0$. One easily verifies that it is also no loss of generality to assume $\alpha = \beta = \gamma = 1$.

We have $2^{g(n)} < 2^{f(n)}$ iff $g(n) < f(n)$. Now we have

$$\frac{g(x)}{f(x)} \leq \frac{f(x)^q}{f(x)} =: \frac{h(x)}{l(x)}$$

Since f is strictly monotone, so is $x \mapsto f(x)^{1-q}$. Elementary calculus shows:

$$\frac{h'(x)}{l'(x)} = q \cdot \frac{1}{f(x)^{1-q}} \xrightarrow[x\to\infty]{} 0$$

Since $f'(x) \neq 0$ for all $x \in \mathbb{R}_+$ we may apply l'Hospital's rule. We obtain $\lim_{x\to\infty} \frac{g(x)}{f(x)} = 0$. □

We call a function $f\colon \mathbb{R}_+ \to \mathbb{R}_+$ *sub-linear* if $f \in \mathcal{O}(x^q)$ for some $q \in (0,1)$. It is a well known fact that all polylogarithmic functions are sub-linear. In fact, for every $k \in \mathbb{N}$ and every $q \in (0,1)$ one has $\log(x)^k \in \mathcal{O}(x^q)$. We obtain:

Theorem 1 (Lower Bound on Space Requirement). *There is no solution scheme $\mathcal{S}$ for the class of Muller games or Streett games which assigns a strategy machine $\mathcal{S}(\mathbf{G})$ to $\mathbf{G} = ((V,E),\varphi)$ that has a space requirement sub-linear in $|V|$. In particular, for no $k \in \mathbb{N}$ can there be such a scheme with space requirement in $\log_2(|V|)^k$.*

Proof. In the previous lemma set $g(x) = x^p$, $p \in (0,1)$ and $f(x) = x$. □

For Turing machines it is well known that any time-bound is also a space-bound. This also holds for strategy machines: Let $\mathcal{M}$ be a strategy machine with $T(\mathcal{M}) \leq N \in \mathbb{N}$. Note that the head position of all three heads is reset to zero at the beginning of every iteration. Consequently $\mathcal{M}$ will always only access the first N bits of the memory tape. This implies:

Proposition 4. *Let $\mathcal{M}$ be a strategy machine with latency $T(\mathcal{M})$ bounded by $N \in \mathbb{N}$. Then the space requirement $S(\mathcal{M})$ is also bounded by N.*

We immediately obtain:

Corollary 1 (Lower Bound on Latency). *There is no solution scheme $\mathcal{S}$ for the class of Muller games or Streett games which assigns a strategy machine $\mathcal{S}(\mathbf{G})$ to $\mathbf{G} = ((V,E),\varphi)$ that has a latency sub-linear in $|V|$. In particular, for no $k \in \mathbb{N}$ can there be such a scheme with space requirement in $\log_2(|V|)^k$.*

Finally. we would like to apply lemma 1 to LTL-games. We note (a proof is given in the appendix):

Proposition 5. *The class of LTL-games is 2^{2^n}-hard. The winning conditions of the witnessing family are of size $\mathcal{O}(n^2)$.*

A *subexponential function* is a mapping $f\colon \mathbb{R}_+ \to \mathbb{R}_+$ for which $f \in \mathcal{O}((2^x)^q) = \mathcal{O}(2^{qx})$ for some $q \in (0,1)$. Again it is well known that all polynomial functions are subexponential.

Theorem 2 (Lower Bounds for LTL-Games). *There can be no solution scheme for LTL-games, which assigns a strategy machine with latency or space requirement subexponential in $|V|$ to every LTL game $\mathbf{G} = ((V,E),\varphi)$. In particular, there can be no such scheme with a latency or space requirement polynomial in $|V|$.*

Proof. Letting $g(x) = 2^{qx}$ for some arbitrary $q \in (0,1)$ and $f(x) = 2^x$ we may apply lemma 1. $\square$

5 Muller Games

We consider Muller games $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ with an arena $\mathcal{A} = (V,E)$ and a winning condition $\mathfrak{F} \subseteq \mathscr{P}(V)$. We will briefly recall Zielonka's construction [Zie98,DJW97] in the next subsection and illustrate how to use it to construct a small strategy machine implementing a winning strategy for player 0 in $\mathbf{G}$.

5.1 Zielonka's Algorithm

We need some notation. Given a finite set V of vertices, a winning condition $\mathfrak{F}$ over V and a set $X \subseteq V$ we write $\mathfrak{F} \restriction X := \mathfrak{F} \cap \mathscr{P}(X)$ for the condition $\mathfrak{F}$ restricted to the vertices X. Next we define

$$\max(\mathfrak{F}) = \{V' \subseteq V \mid V' \notin \mathfrak{F} \wedge \forall V'' \supsetneq V' : \; V'' \in \mathfrak{F}\}$$

for the set of all subsets of V, which are maximal (w.r.t. set inclusion) with the property that they are not in $\mathfrak{F}$. We denote the complement of a set $X \subseteq \mathscr{P}(V)$ in $\mathscr{P}(V)$ by $X^{\mathscr{C}} = \mathscr{P}(V) \setminus X$.

Given a winning condition $\mathfrak{F} \subseteq \mathscr{P}(V)$, we define the *Zielonka tree* of $\mathfrak{F}$, denoted by $\mathcal{Z}_{\mathfrak{F}}$, as follows:

- The root of $\mathcal{Z}_{\mathfrak{F}}$ is labeled with V.
- If s is a node in $\mathcal{Z}_{\mathfrak{F}}$ labeled with a set $A \subseteq V$, such that $A \in \mathfrak{F}$ and $k = |\max(\mathfrak{F} \restriction A)|$ then s has exactly k children, each labeled with a distinct element from $\max(\mathfrak{F} \restriction A)$.
- If s is a node in $\mathcal{Z}_{\mathfrak{F}}$ labeled with a set $A \subseteq V$, such that $A \notin \mathfrak{F}$ and $k = |\max(\mathfrak{F}^{\mathscr{C}} \restriction A)|$ then s has exactly k children, each labeled with a distinct element from $\max(\mathfrak{F}^{\mathscr{C}} \restriction A)$.

Nodes which are labeled with an element from $\mathfrak{F}$ are called *0-level nodes*. The remaining nodes are called *1-level nodes*. A tree is a *0-tree* (resp. 1-tree) if all of its 0-level (resp. 1-level) nodes have at most one child.

For technical reasons we often refer to the Zielonka tree $\mathcal{Z}$ as a labeled tree $\mathcal{Z} = (N, \lambda)$ with node set $N \subseteq \mathbb{N}^*$ (assumed to be prefix closed) and labeling function $\lambda \colon N \to \mathscr{P}(V)$ as described above. Note that the root of $\mathcal{Z}$ is the empty sequence ε. We sometimes make use of additional labeling functions which will be defined in the sequel.

A *path* from the root to a leaf in a Zielonka tree $\mathcal{Z} = \mathcal{Z}_{\mathfrak{F}}$ is a sequence $p = x_0, \ldots, x_k$ of nodes $x_i \in N$. As indicated, we assume that the node x_k is a leaf in $\mathcal{Z}$. In particular, unless stated otherwise, when we use the term "path" we mean one that starts in the root and ends in a leaf. On the path p as given above we write $p(i) = x_i \in N$ for the i-th node in p. Observe that $p(0)$ is the root of $\mathcal{Z}$, which is part of every path.

Note that the sequence $\lambda(p(0)), \ldots, \lambda(p(k))$ is a $\supseteq$-decreasing sequence of sets $V = V_1 \supsetneq V_2 \supsetneq \cdots \supsetneq V_k$. Consequently $k \leq |V|$. We usually include the sets $\lambda(p(i))$ in the description of a path, writing $p = (x_0, \lambda(x_0)), \ldots, (x_k, \lambda(x_k))$. Again the i-th element of p is denoted by $p(i) = (x_i, \lambda(x_i))$. We write $\|p\| = k$ for the *length* of p. Denote the set of all such paths by $P_{\mathcal{Z}}$.

Zielonka's algorithm can now be described as follows (this presentation follows [DJW97]). Given an arena $\mathcal{A} = (V, E)$ and a winning condition $\mathfrak{F}$ over V, we consider the Zielonka tree $\mathcal{Z} = \mathcal{Z}_{\mathfrak{F}} = (N, \lambda)$.

We may assume that the winning region $\mathcal{W}_0$ of player 0 is all of V. Otherwise we restrict $\mathcal{A}$ to $\mathcal{W}_0$ (note that this is still an arena). We define a labeling $\kappa\colon N \to \mathscr{P}(V)$, which assigns *subarenas* to each node. For ease of notation we represent subarenas by their vertex set. In other words, if we use the labeling function λ, we talk about infinity sets good for either player 0 or player 1. If we use κ, we talk about arenas.

We set $\kappa(\varepsilon) = V$. If the labeling for $x \in N$ has been defined and if $x_0, \ldots, x_{k-1}$ are the children of x in $\mathcal{Z}$, we proceed as follows:

1. If x is 0-level we compute $A_i = \mathrm{Attr}_0^{\kappa(x)}(\lambda(x) \setminus \lambda(x_i))$ for $i = 0, \ldots, k-1$. We then set $\kappa(x_i) = \kappa(x) \setminus A_i$.
2. If x is 1-level the computation is much more involved. We define several sequences of sets, $(U_i)_{i\geq 0}$, $(A_i)_{i\geq 1}$, $(E_i)_{i\geq 1}$, $(X_i)_{i\geq 1}$, $(Y_i)_{i\geq 1}$ and $(Z_i)_{i\geq 1}$. Let $U_0 = \emptyset$ and $X_0 = \kappa(x)$. Think of U_i as the set of vertices from which player 0 knows how to win. For $i \geq 1$ we compute
   - $A_i = \mathrm{Attr}_0^{\kappa(x)}(U_{i-1})$
   - $X_i = \kappa(x) \setminus A_i$
   - $E_i = \mathrm{Attr}_1^{X_i}(\lambda(x) \setminus \lambda(x_{i \bmod k}))$
   - $Y_i = X_i \setminus E_i$ (note that $Y_i \subseteq \lambda(x_{i \bmod k})$)
   - Let Z_i be the winning region of player 0 in the subgame $(\mathcal{A} \restriction Y_i, \mathfrak{F} \restriction \lambda(x_{i \bmod k}))$
   - $U_i = A_i \cup Z_i$.
   - Finally we set $\kappa(x_i) = \bigcup_{j \equiv i \bmod k} Z_j$.

Proposition 6. *The sequence $(U_i)_{i\geq 0}$ is monotonically increasing. It becomes stationary when the last k elements are identical, where k is the number of children of x. Therefore the length of this sequence is bounded by $k \cdot |V|$.*

Remark 2. For a given 1-level node $x \in N$ we will sometimes need to refer to the sets $(U_i)_{i\geq 0}$ as computed above. To this end we will write U_i^x for the i-th set of this sequence as computed for x.

We are now left with a tree (N, λ, κ), which we will also denote by $\mathcal{Z}$. Recall that we assumed $\mathcal{W}_0 = V = \kappa(\varepsilon)$. For a proof of the following two lemmas the reader is referred to [DJW97,Zie98]:

Lemma 2. *For every $x \in N$ player 0 wins from every vertex in the subgame $(\kappa(x), \mathfrak{F} \restriction \lambda(x))$.*

Note that this implies that player 0 does not have to leave $\kappa(x)$ to win.

Lemma 3. *If x is 1-level, then then $\kappa(x) = \bigcup_i U_i^x$ where the U_i^x are computed as above. In particular, for every $v \in \kappa(x)$ there exists i with $v \in U_i^x$.*

The algorithm presented in [DJW97] works as follows. As memory it uses the set of paths $P_{\mathcal{Z}}$ leading from the root to a leaf. Given a vertex $v \in V$, we consider the current memory state, $p = x_0, \ldots, x_m$. We pick the maximal index i, such that $v \in \kappa(x_i)$. Following the terminology from [DJW97] x_i is called the *anchor node* of v with respect to p. If x_i is a 0-level node, this implies $v \in \mathrm{Attr}_0^{\kappa(x_i)}(\lambda(x_i) \setminus \lambda(x_{i+1}))$. Then we play the corresponding attractor strategy. As soon as a vertex from $\lambda(x_i) \setminus \lambda(x_{i+1})$ is reached we update the memory. To this end let $c_0, \ldots, c_{k-1}$ be the children of x_i in $\mathcal{Z}$. Let $x_{i+1} = c_j$. Then the update is performed by replacing p by a path to a leaf in the subtree below $c_{j+1 \bmod k}$.

If x_i is a 1-level node, we consider the sets $(U_j^{x_i})_{j \geq 0}$ as described above. Note that, since we assumed $\mathcal{W}_0 = V$, player 0 wins from every vertex in $\kappa(x_i)$. Note also that, by construction, player 0 cannot force the token outside of $\lambda(x_i) \in \mathfrak{F}_1$. Hence player 0 must be able to keep the token in one of the sets $\lambda(c_0), \ldots, \lambda(c_{k-1})$, where $c_0, \ldots, c_{k-1}$ are the children of x_i in $\mathcal{Z}$. Let t be minimal with the property that $v \in U_t$ (note that t exists by lemma 3). Then either $v \in Z_t$ or $v \in \mathrm{Attr}_0^{\kappa(x_i)}(U_{t-1})$. In the first case we pick $c_{t \bmod k}$ and update the memory to some path that goes through this node (note that $c_{t \bmod k} = x_{i+1}$ would contradict the maximality of i; hence a memory update is necessary). We move to an arbitrary node in Z_t. Otherwise we leave the memory state unchanged and play according to the attractor strategy for U_{t-1}. We will skip the proof that this constitutes a winning strategy (for details, see [DJW97,Zie98]).

5.2 An Adaptive Algorithm

Let $\mathfrak{F}$ be a Muller condition over V and let $\mathcal{A} = (V, E)$ be such that $\mathcal{W}_0 = V$ in $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$. In this section we will give a strategy machine $\mathcal{M}$ implementing a winning strategy for player 0 in $\mathbf{G}$. The strategy machine will use the memory tape to store the state of certain subcomputations. This will enable us to "spread out" costly computational tasks over the duration of the play. Thus the algorithm *adapts* to the current play.

We will need two auxiliary Turing machines. The latency, size and space consumption of $\mathcal{M}$ as given below will depend on these machines. In order to be able to talk about complexity, we will use the following assumptions. The details can be found in the appendix. We assume that $\mathcal{A}$ is given as a 2-tape Turing machine of size $|V|^2$ as described in appendix A.1. The strategy machine for $\mathbf{G}$ will be assumed to access this Turing machine as a subroutine. All auxiliary machines which need access to $\mathcal{A}$ will also use this representation. Secondly, we assume $\mathfrak{F}$ is given in the form of a propositional formula $\phi = \phi_{\mathfrak{F}}$. This formula is, in turn, given as a 2-tape Turing machine (see A.3). This requires space $\mathcal{O}(\|\phi\| \cdot \log_2(|V|))$. Again, $\mathcal{M}$ has a copy of this machine in its state space and all auxiliary machines can access it.

The first auxiliary machine, $\mathcal{M}_\lambda$, computes the labeling function λ. More precisely, $\mathcal{M}_\lambda$ is a two tape Turing machine working as follows: Let (X, X') with $X' \subseteq X \subseteq V$. We assume that if $X \in \mathfrak{F}$, then $X' \in \max(\mathfrak{F} \restriction X)$ and $X' \in \max(\mathfrak{F}^{\mathscr{C}} \restriction X)$ otherwise. The inputs to $\mathcal{M}_\lambda$ will be such that these assumptions are always satisfied. Given such an input, $\mathcal{M}_\lambda$ will then compute (using only the second tape) the next[4] label X'' from $\max(\mathfrak{F}\upharpoonright X)$ (resp. $\max(\mathfrak{F}^{\mathscr{C}}\upharpoonright X)$) and write it on the second tape. The reason we use a two tape machine here is to separate reading the input from computing the set.

For the sake of computing the label of a node where we do not have the label of a sibling available, we also require that $\mathcal{M}_\lambda$ accepts inputs (X, n) with $n \in \mathbb{N}$. Then we require that $\mathcal{M}_\lambda$ terminates with the label of the n-th child of the node labeled with X. If n exceeds the number of children of X the behavior is undefined.

Before describing the second auxiliary machine, we have to address a complexity issue: Let x be 1-level with children $c_0, \dots, c_{k-1}$. The number k can be quite large in general (exponential in $|V|$). The number m of sets in the sequence $(U_i^x)_{i\geq 0}$ is bounded by $m \leq k \cdot |V|$ (see proposition 6). However, there can be at most $|V|$ positions i with $U_i^x \subsetneq U_{i+1}^x$ since $(U_i^x)_{i\geq 0}$ is monotonically increasing. This implies that in the sequence $(D_i^x)_{i\geq 1}$, where $D_i^x = U_i^x \setminus U_{i-1}^x$, there are at most $|V|$ elements $\neq \emptyset$. Consequently, we can store the sequence $(D_i^x)_{i\geq 1}$ on tape as a set of at most $|V|$ pairs $\mathcal{D}^x = \{(\mathsf{bin}(i), D_i^x) \mid D_i^x \neq \emptyset\}$ where $\mathsf{bin}(i)$ denotes the binary representation of i. Since $i \leq |V| \cdot k \leq |V| \cdot 2^{|V|}$ we see that $\mathsf{bin}(i)$ requires at most $|V| + \log_2(|V|)$ bits. Furthermore $\sum_i |D_i^x| \leq |V|$ whence storing all pairs $(\mathsf{bin}(i), D_i^x)$ with $D_i \neq \emptyset$ requires $\leq \sum_i \log_2(|V|) + |V| + |D_i^x| \cdot \log_2(|V|) \leq |V| \cdot (\log_2(|V|) + |V|) + |V| \cdot \log_2(|V|) \in \mathcal{O}(|V|^2)$ bits to store all pairs. We can compute the label $\kappa(c_j)$ for any child c_j of x in $\mathcal{Z}$ using the data structure above:

Proposition 7. *Let x be a 1-level node and let with k_x children $c_0, \dots, c_{k_x-1}$. The set $\kappa(c_j)$ for any fixed $j \in \{0, \dots, k_x - 1\}$ can be computed from $\mathcal{D}^x = \{(\mathsf{bin}(i), D_i^x) \mid D_i^x \neq \emptyset\}$ and the number k_x of children of x by a 2-tape Turing machine in time $\mathcal{O}(|V|^3 \cdot |E|)$.*

Proof. We only need to scan the sequence for entries with index $i = j \bmod k$ and remove the attractor $\mathrm{Attr}_0^{\kappa(x)}(U_{j-1})$ from this set, obtaining Z_j. The union of all these remaining sets is then $\kappa(c_j)$.

To compute $\kappa(c_j)$ a Turing machine proceeds as follows. It computes the union of all sets D_i^x seen so far. This requires time $\mathcal{O}(\log_2(i) + |D_i^x| \cdot \log_2(|V|))$ for every D_i^x encountered. If the set D_i^x additionally satisfies $i = j \bmod k_x$ (which requires time $\mathcal{O}(|V|^2)$ to check) we compute an attractor on the union so far. This requires $\mathcal{O}(|V| \cdot |E| \cdot \log_2(|V|)) \subseteq \mathcal{O}(|V|^2 \cdot |E|)$ steps. The difference of D_i^x and this attractor is Z_i^x. The dominating factor in the computation for each i is thus $|V|^2 \cdot |E|$ and there are at most $|V|$ indices i. □

For the second auxiliary machine, let $\mathcal{M}_\kappa$ be a Turing machine which computes the set $\mathcal{D}^x$ as well as the number k_x of children of x. More specifically, $\mathcal{M}_\kappa$ takes a set $V' \notin \mathfrak{F}$ and a subarena $\mathcal{B} = (V'', E|_{V''})$ of $\mathcal{A}$ with $V'' \subseteq V'$ as input. Let $\mathfrak{F}' = \mathfrak{F}^{\mathscr{C}} \upharpoonright V'$. Then $\mathcal{M}_\kappa$ terminates with output $\mathcal{D}$ and $k = |\max(\mathfrak{F}')|$ with respect to $\mathcal{B}$ and to the root in $\mathcal{Z}_{\mathfrak{F}'}$.

We now construct a strategy machine $\mathcal{M}$ of a winning strategy for the Muller game $\mathbf{G}$ as follows. Initially the memory tape contains the leftmost path (given some fixed order on the elements of $\mathscr{P}(V)$) of the Zielonka tree $\mathcal{Z} = \mathcal{Z}_{\mathfrak{F}}$. We compute a labeling function $\kappa(x)$ plus some additional data (the set $\mathcal{D}^x$ and the number k_x of children if x is 1-level) for a given node x as the play evolves. If the

[4] We assume that a fixed order of the labels is given.

memory state is updated to a path p which does not visit x, then the computation for x is lost. The justification for this is the following. In every infinite play π there exists a unique lowest node x_π which is the anchor node of $\pi(n)$ for infinitely many $n \in \mathbb{N}$. x_π is the root of the smallest subtree of $\mathcal{Z}$, such that the play is entirely confined within $\kappa(x_\pi)$ from some point onward. Consequently, the label $\kappa(x_\pi)$ will, from some point onward, no longer be deleted.

To formally define $\mathcal{M}$ we need some notation. First of all we slightly redefine the labeling κ from the previous subsection to a labeling κ'. The label $\kappa'(x) = \kappa(x)$ for a 0-level node x is unchanged. The label for a 1-level node x is changed to $\kappa'(x) = (\kappa(x), \mathcal{D}^x, k_x)$ to also include the set $\mathcal{D}^x$ as defined above as the set of all $D_i^x = U_i^x \setminus U_{i-1}^x$ which are non-empty. Furthermore $\kappa'(x)$ contains the number k_x of children of x in $\mathcal{Z}$.[5] Let x be 1-level and let $\mathcal{D}^x$ and k_x be as described above. Note that, by proposition 7, $\kappa(c)$ can be computed from $\kappa(x)$, $\mathcal{D}^x$ and k_x in time $\mathcal{O}(|V|^3 \cdot |E|)$ for any child c of x.

As mentioned before, our memory states will essentially be paths in $\mathcal{Z}$ with some additional information attached to the nodes. In order to be able to formally define a strategy machine $\mathcal{M}$ we have to specify how these states are stored on tape. To this end we note that we need the following information for every node $p(i)$ on a path p:

- the number n_i indicating that $p(i)$ is the n_i-th child of $p(i-1)$
- the set $\lambda(p(i))$
- the set $\kappa'(p(i))$ where this information has already been computed or the the state of the computation if the information has not been computed yet

Thus our memory states will have the following form:

$$p = (n_0, \lambda(p,i), \theta(p,i)), \dots, (n_r, \lambda(p,r), \theta(p,r)) \tag{1}$$

Here n_i means that $p(i)$ is the n_i-th child of $p(i-1)$.[6] We write $\lambda(p,i)$ for the set $\lambda(p(i))$ and likewise $\kappa(p,i)$ for $\kappa(p(i))$. The values $\theta(p,i)$ satisfy the following properties: For every $i \in \{0, \dots, r\}$ we have

- $\theta(p,i) = \kappa'(p(i))$
- or $\theta(p,i) = \emptyset$
- or $p(i)$ is 1-level and $\theta(p,i) = (\kappa(p,i), c)$, where c is a configuration of $\mathcal{M}_\kappa$

We require that $\theta(p,i) = \kappa'(p(i))$ implies $\theta(p,j) = \kappa'(p(j))$ for all $j < i$ and that $\theta(p,i) = \emptyset$ implies $\theta(p,k) = \emptyset$ for all $k > i$. Additionally we require that if $\theta(p,i)$ contains a configuration of $\mathcal{M}_\kappa$ then $\theta(p,j) = \kappa'(p(j))$ for all $j < i$ and $\theta(p,k) = \emptyset$ for all $k > i$. We say $\theta(p,i)$ (or the node $p(i)$) is *untouched* if $\theta(p,i) = \emptyset$. If $\theta(p,i)$ is not untouched and does not contain a configuration of $\mathcal{M}_\kappa$ we say $\theta(p,i)$ (or $p(i)$) is *finished*. Otherwise $\theta(p,i)$ contains a configuration of $\mathcal{M}_\kappa$ and we say $\theta(p,i)$ (or $p(i)$) is *active*.

Proposition 8. *The labeling function θ requires space $\mathcal{O}(|V| + S_{\mathcal{M}_\kappa})$, where $S_{\mathcal{M}_\kappa}$ is the space consumption of $\mathcal{M}_\kappa$. Hence the space required to store p is in $\mathcal{O}(|V| \cdot (3|V| + S_{\mathcal{M}_\kappa})) = \mathcal{O}(|V| \cdot (|V| + S_{\mathcal{M}_\kappa}))$.*

[5] Note that although this additional information is actually associated with the children of x we still attach it to x itself. The reason is that in the strategy presented in the previous subsection, we only used the sets $(U_i^x)_{\geq 0}$ if the anchor node of a vertex v was x.

[6] By convention we set $n_0 = \perp$. This value is not used in any computation but to treat the root in a different way than the rest of the nodes would unnecessarily complicate the notation.

Proof. Note that the number $n_i \leq k_x$ (where k_x denotes the number of children of x) requires at most $|V|$ bits (the number of children of a node is bounded by $2^{|V|}$). Secondly, note that since $\lambda(p,i) \supsetneq \lambda(p,i+1)$ we necessarily have $r \leq |V|$. □

The Strategy Machine Formally $\mathcal{M}$ proceeds as follows. We first define the initial memory state p_I. To this end we set $n_0 = \perp$ and $n_1 = n_2 = \cdots n_r = 0$ to denote the leftmost path in $\mathcal{Z} = (N, \lambda, \kappa)$. In particular, we require r to be maximal (i.e. the node $n_1 \cdots n_r \in N$ is a leaf). Let i be minimal with the property that $n_1 \cdots n_i$ is 1-level (i.e. either $i = 0$ or $i = 1$). If $i = 0$ then we set $p_I(0) = (n_0, \lambda(p_I, 0), (V, c_I))$, where c_I is the initial configuration of $\mathcal{M}_\kappa$. Otherwise $p_I(0) = (n_0, \lambda(p_I, 0), V)$ and $p_I(1) = (n_1, \lambda(p_I, 1), (\kappa(p_I, 1), c_I))$. The properties of θ described above then imply $p(j) = (n_j, \lambda(p_I, j), \emptyset)$ for all $j > i$. The memory update below will ensure that, for any memory state p, the highest node $p(i)$, such that $\kappa(p,i)$ is not yet available (i.e. $\theta(p,i) = \emptyset$), is 0-level. Note that the initial memory state p_I satisfies this invariant. Additionally we assume that at any given moment either all sets $\kappa(p,i)$ are computed or there exists a node which is active. The memory update will also preserve this second invariant.

Suppose the input to $\mathcal{M}$ is $v \in V$ and suppose the current state is p as in (1). We assume that the highest node in p for which κ has not been computed is 0-level. Let i be the maximal index with $p(i) = (n_i, \lambda(p,i), \theta(p,i))$, $\theta(p,i) \neq \emptyset$ and $v \in \kappa(p,i)$. This just means that we pick i maximal such that $\kappa(p,i)$ has been computed and contains v. This requires time $\mathcal{O}(|V|^3)$. We assume that $i < \|p\|$ (the case $i = \|p\|$ requires only minor modifications). Note that under this assumption we either have $v \notin \kappa(p,i+1)$ or $\kappa(p,i+1)$ has not been computed yet (i.e. $\theta(p,i+1) = \emptyset$). We distinguish these two cases:

(a) $\kappa(p,i+1)$ has not been computed yet. Then $p(i)$ is 1-level by assumption. Furthermore $p(i)$ must be active. Let $\theta(p,i) = (\kappa(p,i), c)$ where c is a configuration of $\mathcal{M}_\kappa$. Assume first that c is not a terminal configuration, i.e. $\mathcal{M}_\kappa$ requires more computation steps. Then $\mathcal{M}$ simulates another computation step of $\mathcal{M}_\kappa$ and replaces c by its unique successor configuration c'. $\mathcal{M}$ outputs an arbitrary vertex neighboring v in $\kappa(p,i)$. This requires no more than $\mathcal{O}(\log_2(|V|))$ steps.
If c is a terminal configuration we replace c by the set $\mathcal{D}^{p(i)}$ and the integer $k_{p(i)}$ which have been computed (requiring $\mathcal{O}(|V|^2)$ steps since as many tape cells are required to store the sequence). We then compute $\kappa(p,i+1)$ from $\mathcal{D}^{p(i)}$ (requiring $\mathcal{O}(|V|^3 \cdot |E|)$ by proposition 7). If $i+1 < \|p\|$ then we also compute $\kappa(p(i+2))$. This is a simple attractor on the set $\lambda(p,i+1) \setminus \lambda(p,i+2)$ as $p(i+1)$ is 0-level (requiring $\mathcal{O}(|V| \cdot |E| \cdot \log_2(|V|))$ steps). Finally, if even $i+2 < \|p\|$ then we set $p(i+2)$ active, i.e. we replace $\theta(p,i+2) = \emptyset$ by $(n_{i+2}, \lambda(p,i+2), (\kappa(p,i+2), c_I))$.
For the next move we have two cases:
   (i) $v \notin \kappa(p,i+1)$: Then $p(i)$ is the anchor node of v with respect to p. We proceed as the classical Zielonka strategy indicates and compute the minimal index t with $v \in U_t^{p(i)}$ (note that this can be done while computing $\mathcal{D}^{p(i)}$ above). Then v is either in the 0-attractor to $U_{t-1}^{p(i)}$ or $v \in Z_t^{p(i)}$. In the first case we play the corresponding attractor strategy.

In the second case we update our memory state to a path through the corresponding child of $p(i)$. This requires $\mathcal{O}(|V| \cdot T_{\mathcal{M}_\lambda})$ steps, where $T_{\mathcal{M}_\lambda}$ is the runtime of $\mathcal{M}_\lambda$.

(ii) $v \in \kappa(p, i+1)$: We move to an arbitrary neighbor of v in $\kappa(p, i+1)$.

(b) If $\kappa(p, i+1)$ has been computed, then again $p(i)$ is the anchor node of v with respect to p. We proceed as the usual Zielonka strategy indicates: If $p(i)$ is 0-level, we play an attractor strategy on the set $\lambda(p,i) \setminus \lambda(p, i+1)$ ($\mathcal{O}(|V| \cdot |E| \cdot \log_2(|V|))$ steps). Otherwise $p(i)$ is 1-level. Then we compute the minimal t with $v \in U_t^{p(i)}$ and play an attractor strategy on $U_t^{p(i)}$ or update the memory to the leftmost path through the unique child c of $p(i)$ with $Z_t \subseteq \kappa(c)$ (requiring $\mathcal{O}(|V|^3 \cdot |E| + |V| \cdot T_{\mathcal{M}_\lambda})$ steps) and output an arbitrary neighbor of v in $\kappa(c)$.

The complete algorithm is given in algorithm 1.

To prove that this indeed implements a winning strategy we have make a few preparations. Let $M_{\mathcal{M}}$ denote the set of possible *memory paths* of $\mathcal{M}$, i.e. the set of paths p of the form (1). Recall from section 2 that $f_{\mathcal{M}}$ denotes the function implemented by $\mathcal{M}$. Given an infinite play π consistent with $f_{\mathcal{M}}$ let $\rho_\pi \colon \mathbb{N} \to M_{\mathcal{M}}$ be the function which assigns the memory path *after* the n-th iteration to any given natural number $n \in \mathbb{N}$. So $\rho_\pi(1)$ is the memory path after the first memory update and so forth. Let $\mathcal{Z} = \mathcal{Z}_{\mathfrak{F}} = (N, \lambda, \kappa)$ be the fully labeled Zielonka tree. We claim:

Lemma 4. *Let π be a play in $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ consistent with $f_{\mathcal{M}}$. Then there exists a unique lowest node x_π in $\mathcal{Z}$ and a natural number $m_0 \in \mathbb{N}$, such that all of the following assertions hold:*

(i) x_π occurs in $\rho_\pi(m)$ for all $m \geq m_0$.
(ii) x_π is finished in $\rho_\pi(m_0)$ (and therefore in $\rho_\pi(m)$ for all $m \geq m_0$).
(iii) x_π is 0-level.

Proof. For simplicity we refer to ρ_π simply as ρ. We first observe that the root $x_0 \in N$ occurs finished in every memory path from some point onwards – if the root is 0-level, it will occur finished from the beginning; if it is 1-level, it will occur finished from the point onwards, at which the computation of $\mathcal{M}_\kappa$ ends. Hence the set of nodes which occur finished from some point onwards is non-empty and so has a unique lowest element x_π (if there are two such that neither is a descendant of the other then evidently one of them must be deleted time and again).

It remains to show that x_π is 0-level. Indeed, suppose x_π were 1-level. Let $\kappa'(x_\pi) = (\kappa(x_\pi), \mathcal{D}^{x_\pi}, k_{x_\pi})$. We first observe that there are infinitely many $m \in \mathbb{N}$, such that $\pi(m) \in \kappa(x_\pi) \setminus \kappa(x')$, where x' is the successor of x_π in $\rho(m-1)$ (recall that $\kappa(x')$ is always available for any successor x' of a finished node). However, this implies that we attract the token to a set $U_t^{x_\pi}$ with a lower index t every time. Since this can be done only finitely many times, the play must from some point onwards either remain in the label $\kappa(x')$ of one of the children x' of x_π or it must leave the arena $\kappa(x_\pi)$. Both options contradict the choice of x_π and so x_π must be 0-level. $\square$

We are now ready to prove the following lemma:

```
v = read();
p = load();                                                    // Load current memory state p
p' = p;
(n_i, λ(p,i), θ(p,i)) = findAnchorNode(p, v);
if θ(p,i+1) = ∅ then                                           // Case (a) above
    if c is not terminal configuration then
        c' = simulateNextComputationStep(c);
        p'(i) = (n_i, λ(p,i), κ(p,i), c');
        v' = some neighbor of v in κ(p,i);
    else
        p'(i) = (n_i, λ(p,i), κ(p,i), outputTM(c));            // (n_i, λ(p,i), κ(p,i), D^{p(i)}, k_{p(i)})
        B = computeKappa(n_{i+1}, κ(p,i), D^{p(i)}, k_{p(i)}); // κ for 0-level node
        C = computeKappa(B, λ(p,i), λ(p,i+1));                 // κ for 1-level node
        p'(i+1) = (n_{i+1}, λ(p,i+1), B);
        p'(i+2) = (n_{i+2}, λ(p,i+2), C, c_I);
        if v ∉ κ(p',i+1) then goto(32);
        else v' = some neighbor of v in κ(p',i+1);
    end
else                                                           // Case (b) above
    if p(i) is 0-level then
        if v' ∉ λ(p,i) \ λ(p,i+1) then
            v' = nextAttractorMove(κ(p,i), λ(p,i) \ λ(p,i+1));
        else
            n', V' = computeLambda(φ,λ(p,i),λ(p,i+1));         // new label
            B = computeKappa(κ(p,i),λ(p,i),V');                // compute κ(n_1 ··· n_i n')
            p'(i+1) = (n', V', B);
            p' = finishPath(p',i+1);
            v' = nextAttractorMove(κ(p',i), λ(p',i) \ λ(p',i+1));
        end
    else
        t = minimalSetContainingVertex(v, D^{p(i)});           // Minimal t with v ∈ U_t
        if v ∈ Attr_0^{κ(p,i)}(U_{t-1}^{p(i)}) then
            v' = nextAttractorMove(κ(p,i),U_{t-1}^{p(i)});
        else
            n' = t mod k_{p(i)};
            B = computeKappa(n', κ(p,i), D^{p(i)}, k_{p(i)});
            V' = computeLambda(φ,λ(p,i),n');
            p'(i+1) = (n', V', B);
            p' = finishPath(p',i+1);
            v' = some neighbor of v in B;
        end
    end
end
store(p');
write(v');
```

Algorithm 1: The Adaptive Muller Algorithm.

```
Input: unfinished path p' and index t
while p'(t) has child do                                       // compute complete path
    V_t = computeLambda(φ,λ(p',t),0);
    p'(t+1) = (0, V', ∅);
    t = t + 1;
end
return p';
```

Algorithm 2: The **finishPath** subroutine.

Lemma 5. *$\mathcal{M}$ implements a winning strategy for player 0.*

Proof. Let π be consistent with $f_{\mathcal{M}}$ and let x_π be the unique 0-level node chosen according to lemma 4. Then, by choice of x_π as the lowest node which is seen infinitely often in ρ_π we must have infinitely many vertices $\pi(m) \in \kappa(x_\pi) \setminus \kappa(x')$, where x' is the successor of x_π in $p(m-1)$. This implies that $\pi(m)$ is in the attractor $\mathrm{Attr}_0^{\kappa(x_\pi)}(\lambda(x_\pi) \setminus \lambda(x'))$. Since we update the memory path to a path through the next sibling of x' (in some fixed order) in some later iteration, we conclude that we see a vertex from $\lambda(x_\pi) \setminus \lambda(x')$ for every child x' of x_π. Since x_π is 0-level this implies that π is won by player 0. $\square$

Altogether we have shown the following lemma:

Lemma 6. *Let $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ be a Muller game, where $\mathcal{A} = (V, E)$ and $\mathfrak{F}$ is given by a propositional formula ϕ. Then there exists a strategy machine $\mathcal{M}$ of size $\|\mathcal{M}\| \in \mathcal{O}(|V|^2 + \|\phi\| \cdot \log_2(|V|) + \|\mathcal{M}_\lambda\| + \|\mathcal{M}_\kappa\|)$, space consumption $S(\mathcal{M}) \in \mathcal{O}(|V| \cdot (|V| + S_{\mathcal{M}_\kappa}))$ and latency $T(\mathcal{M}) \in \mathcal{O}(|V|^3 \cdot |E| + |V| \cdot T_{\mathcal{M}_\lambda})$.*

Note that the latency and space complexity are parametrized by the runtime and space complexity of $\mathcal{M}_\lambda$ and $\mathcal{M}_\kappa$ respectively. Since solving Muller games is PSPACE complete [HD05,HD08] it follows that we can assume $S_{\mathcal{M}_\kappa}$ to be polynomial in $|V| + \|\phi\|$. In fact, as is outlined in appendix C, a 2-tape Turing machine can decide in space $\mathcal{O}(|V| \cdot (|V| + \|\phi\|) \cdot \log_2(|V|))$ if a vertex v belongs to the winning set of player 0 or not. Using this complexity bound, one can show that $S_{\mathcal{M}_\kappa}$ is also in $\mathcal{O}(|V| \cdot (|V| + \|\phi\|) \cdot \log_2(|V|))$. Note also that we may assume $\|\mathcal{M}_\kappa\|$ to be constant in $\mathcal{A}$ and ϕ, since by assumption all references to $\mathcal{A}$ and ϕ use the internal representation of $\mathcal{M}$. So $\mathcal{M}_\kappa$. The bottleneck is $T_{\mathcal{M}_\lambda}$. We can bound T_λ by $|V| \cdot \log_2(|V|) \cdot \max\{|\mathfrak{F}|, |\mathfrak{F}^{\mathscr{C}}|\}$. Note that if ϕ allows us to obtain a faster algorithm for finding a maximal model we can decrease this bound. In the following subsection we will show how to obtain an efficient machine $\mathcal{M}_\lambda$ if the winning condition ϕ is given by a Streett condition.

For the general case we now have:

Theorem 3. *Let $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ be a Muller game, where $\mathcal{A} = (V, E)$ and $\mathfrak{F}$ is given by a propositional formula ϕ. There exists a strategy machine $\mathcal{M}$ of size and space requirement polynomial in $|V|$ and $\|\phi\|$. The latency is linear in $\max\{|\mathfrak{F}|, |\mathfrak{F}^{\mathscr{C}}|\}$. More precisely we have size $\|\mathcal{M}\| \in \mathcal{O}(|V|^2 + \|\phi\| \cdot \log_2(|V|))$, space consumption $S(\mathcal{M}) \in \mathcal{O}(|V|^2 \cdot (|V| + \|\phi\|) \cdot \log_2(|V|))$ and latency $T(\mathcal{M}) \in \mathcal{O}(|V|^2 \cdot (|V| \cdot |E| + \log_2(|V|) \cdot \max\{|\mathfrak{F}|, |\mathfrak{F}^{\mathscr{C}}|\}))$.*

Muller games are usually solved via *latest appearance records* [GH82,McN93], yielding a Mealy machine of size $|V| \cdot |V|!$. The strategy machine obtained by applying proposition 1 is of size $|V|^2 \cdot |V|!$. The size we obtain in the above theorem is exponentially lower at the price of an exponentially longer latency.

Remark 3. Note that, unlike the situation of computing κ', we do not spread out the computation of $\mathcal{M}_\lambda$ across the infinite play. The reason for this is that we do not have a compact representation of the sets $\lambda(x_0), \ldots, \lambda(x_k)$ for the children $x_0, \ldots, x_k$ of x. This is in contrast to the labels $\kappa(x_0), \ldots, \kappa(x_k)$, where the set $\mathcal{D}^x$ provided such a compact representation.

6 Streett Games

Note that every Streett condition Ω can be written as an equivalent Muller condition $\mathfrak{F}_\Omega = \{X \subseteq V \mid X \text{ violates no pair from } \Omega\}$. There is a characterization of those Muller conditions, which are equivalent to a Streett condition.

Proposition 9 (see [Zie98]). *Let $\mathfrak{F}$ be a Muller condition. The following are equivalent:*

(a) There exists a Streett condition Ω with $\mathfrak{F}_\Omega = \mathfrak{F}$.
(b) The Zielonka tree $\mathcal{Z}_\mathfrak{F}$ is a 1-tree[7].

Let $\Omega = \{(R_1, G_1), \ldots, (R_k, G_k)\}$ be a Streett condition over V. We may assume that $R_i \cap G_i = \emptyset$ for all $i = 1, \ldots, k$. Given Ω the computation of the sets $\max(\mathfrak{F})$ becomes much easier, as the following considerations show:

Proposition 10. *Let $X \subseteq V$, such that $X \in \mathfrak{F}_\Omega$. If $Y \subseteq X$ is maximal with $Y \notin \mathfrak{F}_\Omega$ then there exists $i \in \{1, \ldots, k\}$ with $Y = X \setminus G_i$ and $R_i \cap X \neq \emptyset$.*

Proof. Let Y be such a set. Then clearly Y violates at least one Streett pair (R_i, G_i). Note that this implies $X \cap R_i \neq \emptyset$. Let $Y' = X \setminus G_i$. Then $Y \subseteq Y'$. Also we have $Y' \notin \mathfrak{F}_\Omega$. The maximality of Y implies $Y = Y'$. □

This implies that there are at most k elements in $\max(\mathfrak{F}')$ for every sub-condition $\mathfrak{F}'$. We now show that under certain conditions the converse is also true. To this end let $X \in \mathfrak{F}_\Omega$ be fixed. We define a partial order on Ω. We let $(R, G) \leq_X (R', G')$ if $R \cap X \neq \emptyset \neq R' \cap X$ and $G' \cap X \subseteq G \cap X$. Note that if $(R, G) \leq_X (R', G')$ then $X \setminus G \subseteq X \setminus G'$. We define

$$\Omega_X^{\max} = \{(R, G) \in \Omega \mid \forall (R', G') \in \Omega : \ (R', G') \not>_X (R, G)\}$$

We can now show:

Proposition 11. *Let $X \in \mathfrak{F}_\Omega$. Then $Y \subseteq X$ is maximal with $Y \notin \mathfrak{F}_\Omega$ iff there exists $(R, G) \in \Omega_X^{\max}$ with $Y = X \setminus G$. Moreover, for all such pairs $(R, G), (R', G') \in \Omega_X^{\max}$ we have $(R, G) \equiv_X (R', G')$.*

Proof. From left to right we proceed as before. Let $Y \notin \mathfrak{F}_\Omega$. By the previous proposition there exists i with $Y = X \setminus G_i$ and $R_i \cap X \neq \emptyset$. We claim that also (R_i, G_i) is $\leq_X$-maximal, i.e. $(R_i, G_i) \in \Omega_X^{\max}$. Indeed, if there is j with $R_j \cap X \neq \emptyset$ and $X \cap G_j \subsetneq X \cap G_i$, then $X \setminus G_j \supsetneq X \setminus G_i = Y$, in contradiction to the maximality of Y. The claim $(R, G) \equiv_X (R', G')$ for all pairs which yield Y is trivial.

For the converse let $(R, G) \in \Omega_X^{\max}$ be a Streett with $Y = X \setminus G$. If Y' is such that $Y \subsetneq Y' \subseteq X$ with $Y' \notin \mathfrak{F}_\Omega$ then by the previous direction there exists $(R', G') \in \Omega_X^{\max}$ with $Y' = X \setminus G'$. This immediately implies $(R', G') >_X (R, G)$. A contradiction. □

Using the preceding propositions we can construct a Turing machine $\mathcal{M}_\lambda$ as used in algorithm 1 from the previous section. Recall that we required $\mathcal{M}_\lambda$ to be a Turing machine which takes tuples of the following form as input. $\mathcal{M}_\lambda$ either

[7] cf. sec. 5.1

receives a tuple (X, X') with two sets $X, X' \subseteq V$ or a tuple (X, n) with $X \subseteq V$ and $n \in \mathbb{N}$ as input. In the first case we assume $X' \subseteq X \subseteq V$, such that either $X \in \mathfrak{F}$ and $X' \in \max(\mathfrak{F} \restriction X)$ or $X \notin \mathfrak{F}$ and $X' \in \max(\mathfrak{F}^{\mathscr{C}} \restriction X)$. In the second case me may assume $n \leq |\max(\mathfrak{F} \restriction X)|$ if $X \in \mathfrak{F}$ or $n \leq \max(\mathfrak{F}^{\mathscr{C}} \restriction X)|$ if $X \notin \mathfrak{F}$.

Upon any input $(X, *)$, with $*$ being either a set or a number, $\mathcal{M}_\lambda$ first computes $\Omega_X^{\max}$ and for every $\equiv_X$-equivalence class it retains at most one representative. This takes time $\boldsymbol{\mathcal{O}}((k \cdot |V| \cdot \log_2(|V|))^2)$. We denote the resulting set by Ω'. It is an $\leq_X$-antichain. In the following we assume Ω' has been computed. Write $\Omega' = \{(R_{i_1}, G_{i_1}), \ldots, (R_{i_s}, G_{i_s})\}$.

Suppose now we are given an input (X, X') with $X \in \mathfrak{F}$. Then there exists i_j with $X' = X \setminus G_{i_j}$. Moreover, since Ω' is an $\leq_X$-antichain, this index i_j is unique. We compute $Z = X \setminus G_{(i+1 \bmod s)+1}$. If $Z \neq X'$, we output Z. Otherwise $i = (i + 1 \bmod s) + 1$ by the uniqueness of i. Then we simply output X' again. If we are given a natural number n instead of X' we compute the n-th set in this fashion meaning we choose the n-th index i_n.

If $X \notin \mathfrak{F}$ then the node with label X can have at most one child. Consequently we output X' if we are given an input of the form (X, X') or we must compute the unique maximal set $X' \subseteq X$ with $X' \in \mathfrak{F}$ if it exists. To this end we use the (unmodified) Streett condition Ω and for every pair $(R, G) \in \Omega$ with $G \cap X = \emptyset$ we remove all elements in $X \cap R$ from X. This requires time $\boldsymbol{\mathcal{O}}(|V| \cdot k \cdot \log_2(|V|))$. We have shown:

Proposition 12. *Let Ω be a Streett condition. Then there exists a machine $\mathcal{M}_\lambda$ computing λ with runtime $T_{\mathcal{M}_\lambda} \in \boldsymbol{\mathcal{O}}((|V| \cdot k \cdot \log_2(|V|))^2)$.*

As a consequence of the previous considerations and lemma 6 we obtain:

Theorem 4. *Let $\mathbf{G} = (\mathcal{A}, \Omega)$ be a Streett game. Then there exists a strategy machine of size, space requirement and latency polynomial in $|V|$ and $|\Omega|$. More precisely $\mathcal{M}$ is of size $\|\mathcal{M}\| \in \boldsymbol{\mathcal{O}}(|V|^2 + |V| \cdot |\Omega| \cdot \log_2(|V|))$, space consumption $S(\mathcal{M}) \in \boldsymbol{\mathcal{O}}(|V|^2(|V| + |\Omega|) \cdot \log_2(|V|))$ and latency $T(\mathcal{M}) \in \boldsymbol{\mathcal{O}}(|V|^2 \cdot (|V| \cdot |E| + |\Omega|^2 \cdot \log_2(|\Omega|)^2))$.*

Recall that Streett games are usually solved using *index appearance records* [Saf92,GTW02]. This memory structures yields Mealy machine with roughly $|\Omega|^2 \cdot |\Omega|!$ states. Recalling the construction from proposition 1 a straightforward conversion of such a machine to a strategy machine would yield one of size $\boldsymbol{\mathcal{O}}(|\Omega|^2 \cdot |\Omega|! \cdot |V|)$ and of latency $\boldsymbol{\mathcal{O}}(|\Omega|)$. We significantly reduce the size of the strategy machine while the latency remains polynomial in $|\Omega|$.

7 Conclusion and Future Work

We introduced the formal model of a strategy machine based on Turing machines. We showed how different new criteria of a strategy – latency, space requirement and size – fit into this model, providing general lower bounds for the classes of Muller, Streett and LTL games. Using this model one can obtain polynomial sized machines for Muller games which have polynomial space requirement. The runtime is linear in the size of the winning condition. This machine adapts the strategy as the play proceeds and critically relies on the fact that costly computations may be spread out over the course of several iterations. We were able to show that in the special case of a Streett game the very same algorithm can be made to work with a polynomial latency in the size of the arena and of the number of Streett pairs. The space requirement and size of all our strategy machines are polynomial in the size of the winning condition (given by a propositional formula) and of the arena. Altogether, this is an exponential improvement over the straightforward way of transforming a Mealy machine into a strategy machine via a case distinction over all inputs. This approach results in a machine of exponential size in general.

We plan to extend these results to different kinds of ω-regular games. We have partial results on Request-Response games. Another, more ambitious task, is to derive a theory of transforming arbitrary Mealy machines into strategy machines in such a way that the space consumption, latency and size obey certain bounds. Likewise one would like to find relations between the parameters indicating the nature of a trade-off. Infinite-state strategies fit neatly into our model and lend themselves to a closer study in this context.

References

[BGJ+07] Roderick Bloem, Stefan Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Specify, compile, run: Hardware from psl. *ENTCS*, 190:3–16, November 2007.

[DJW97] S. Dziembowski, M. Jurdzinski, and I. Walukiewicz. How much memory is needed to win infinite games? In *Proc. of the 12th Ann. IEEE Symp. on Logic in Comp. Sci.*, LICS '97, pages 99–, Washington, DC, USA, 1997. IEEE Computer Society.

[EJ99] E. Allen Emerson and Charanjit S. Jutla. The complexity of tree automata and logics of programs. *SIAM J. Comput.*, 29:132–158, September 1999.

[GH82] Yuri Gurevich and Leo Harrington. Trees, automata, and games. *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 60–65, 1982.

[GH11] Marcus Gelderie and Michael Holtmann. Memory reduction via delayed simulation. In *iWIGP*, pages 46–60, 2011.

[GHS09] Martin Grohe, André Hernich, and Nicole Schweikardt. Lower bounds for processing data with few random accesses to external memory. *J. ACM*, 56:12:1–12:58, May 2009.

[GTW02] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, editors. *Automata logics, and infinite games: a guide to current research.* Springer-Verlag, New York, NY, USA, 2002.

[HD05] Paul Hunter and Anuj Dawar. Complexity bounds for regular games (extended abstract). In *Symposium on Mathematical Foundations of Computer Science (MFCS)*, 2005.

[HD08] Paul Hunter and Anuj Dawar. Complexity bounds for muller games. *Theoretical Computer Science (TCS)*, 2008. Submitted.

[HL07] Michael Holtmann and Christof Löding. Memory reduction for strategies in infinite games. In *CIAA*, pages 253–264, 2007.

[Hor08] Florian Horn. Explicit muller games are ptime. In *FSTTCS*, pages 235–243, 2008.

[Löd11] Christof Löding. Infinite games and automata theory. In Krzysztof R. Apt and Erich Grädel, editors, *Lectures in Game Theory for Computer Scientists*. Cambridge UP, 2011.

[Mad11] Parthasarathy Madhusudan. Synthesizing reactive programs. In *Comp. Sci. Log., CSL 2011, Proceedings*, pages 428–442. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[McN93] Robert McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149 – 184, 1993.

[PP04] D. Perrin and J.É. Pin. *Infinite words: automata, semigroups, logic and games.* Pure and applied mathematics. Elsevier, 2004.

[Saf92] Shmuel Safra. Exponential determinization for ω-automata with strong-fairness acceptance condition (extended abstract). *STOC '92*, pages 275–282, 1992.

[Zie98] Wiesław Zielonka. Infinite games on finitely coloured graphs with applications to automata on infinite trees. *Theor. Comput. Sci.*, 200:135–183, June 1998.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2009-01 * Fachgruppe Informatik: Jahresbericht 2009

2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications

2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices

2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs

2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete

2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I

2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs

2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem

2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm

2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs

2009-12 Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes

2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games

2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV'09)

2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhäußer: Compositional Abstraction for Stochastic Systems

2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs

2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata

2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies

2010-01 * Fachgruppe Informatik: Jahresbericht 2010

2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time

2010-03 Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering

2010-04 René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme

2010-05 Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme

2010-06 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata

2010-07 George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms

2010-08 Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting

2010-09 George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs

2010-10 Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut

2010-11 Martin Zimmermann: Parametric LTL Games

2010-12 Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut

2010-13 Ashraf Armoush: Design Patterns for Safety-Critical Embedded Systems

2010-14 Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination

2010-15 Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode

2010-16 Christian Berger: Automating Acceptance Tests for Sensor- and Actuator-based Systems on the Example of Autonomous Vehicles

2010-17 Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten

2010-18 Ibrahim Armaç: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit

2010-19 Felix Reidl: Experimental Evaluation of an Independent Set Algorithm

2010-20 Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games

2011-01 * Fachgruppe Informatik: Jahresbericht 2011

2011-02 Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting

2011-03 Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems

2011-04 Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars

2011-07 Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV

2011-08 Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog

2011-09 Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing

2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11 Nils Jansen, Erika Ábrahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
2011-19 Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode
2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 * Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.

A Turing Machine Complexity Proofs

Throughout this section, let $\mathcal{A} = (V, E)$ be an arena with $V \subseteq \mathbb{B}^*$. We make this assumption so as to not have to deal with encoding vertices as binary strings. This encoding does not influence the results mentioned below except where explicitly mentioned.

A.1 Representing the Arena

We first describe a two-tape Turing machine $\mathcal{M}_{\mathcal{A}}$ which represents the arena $\mathcal{A}$. The idea is to call $\mathcal{M}_{\mathcal{A}}$ as a subroutine, which is why we allow it to have several initial states. A call to the machine is then implemented as moving the calling machine writing its state onto the tape and moving to one of the input states of $\mathcal{M}_{\mathcal{A}}$. The representation will resemble an adjacency list.

Formally $\mathcal{M}_{\mathcal{A}} = (Q, \mathbb{B}, \hat{\mathbb{B}}, q_{next}, q_E, q_{tE}, \delta, q_{return}, q_{fail}, q_{EOF})$. We first describe the meaning of the respective input state. Then we turn to describe the semantics in detail.

First, q_{next} will write the next vertex (in some fixed order) from V on the second tape if a vertex is given initially. Otherwise it will output the first vertex. The state q_E is intended for *edge lookup*. Roughly speaking it will give the first vertex (in some fixed order) which is adjacent to the input vertex. If there are two (adjacent) vertices given, it will produce the next neighbor or signal that there is no next neighbor. In a similar fashion q_{tE} (the t is for *transpose*) is intended to give the next adjacent vertex in the *transpose arena* $\mathcal{A}^T = (V, E^T)$, where $E^T = \{(y, x) \mid (x, y) \in E\}$.

More formally, let $v = v_0 \cdots v_k \in \mathbb{B}^k$ be the input. We assume this input to be on the first tape of the machine. From state q_{next} the machine will read v. For this we have states $q^n_{b_0 \cdots b_t}$ for $t \leq k$, $b_i \in \mathbb{B}$, such that $(q_{next}, v) \vdash^* (v, q^n_v)$. If $v \notin V$ it will terminate with state q_{fail}. Otherwise it will output the next vertex v' in some fixed internal order of V. If there is no next vertex the machine will terminate in state q_{EOF}. Otherwise it will terminate in state q_{return}. If no input is given then the first vertex in this order is written onto the second tape. The machine terminates in state q_{return}.

From state q_E the machine will inspect an adjacency-list representation it has in memory. Suppose the input is v. If the second tape also contains a vertex v' the machine will verify that both v and v' are in V and that $(v, v') \in E$. If one of these assertions is not met, the machine will terminate in state q_{fail}. Otherwise the machine will output the next vertex v'' neighboring v if it exists. In this case it will terminate in q_{return}. Otherwise it will terminate in q_{EOF}. If no vertex v' is given on the second tape (i.e. the second tape is empty), the machine will simply output the first neighbor of v. The meaning of q_{tE} is analogous except that we consider $\mathcal{A}^T$ instead of $\mathcal{A}$.

One easily verifies that $\mathcal{M}_{\mathcal{A}}$ requires $\mathcal{O}(|V|^2)$ states. If L is the maximal length of a string $v \in V$, the runtime is bounded by $\mathcal{O}(L)$.

A.2 Attractor Computation

Now we turn to attractor computation. Let $\mathcal{A} = (V, E)$ be an arena, given as a subprogram $\mathcal{M}_{\mathcal{A}}$ as described above. We assume that $\mathrm{last}(v) = 1$ iff $v \in V_1$. Note that we can ensure this by encoding the vertices with one bit overhead. We denote the length of the longest word $v \in V$ by L.

Proposition 13. *There exists a two tape Turing machine with $\mathcal{O}(|V|^2)$ states using $\mathcal{M}_{\mathcal{A}}$ as a subroutine, which computes the attractor of a given set $S \subseteq V$ in time $\mathcal{O}(|V| \cdot |E| \cdot L)$.*

Proof. Let S be given on the first tape as a sequence of vertices $s_1\#s_2\#\cdots\#s_k$. Then our machine first moves this description on the second tape. This requires $|S| \cdot L$ steps. Next it writes a pair (v_i, n_i) for every player 1 node $v_i \in V_1$ onto the first tape. Here n_i is the number of edges going out of v_i. This requires time $\mathcal{O}(|V| \cdot |E|)$. To see this consider the machine $\mathcal{M}_{\mathcal{A}}$ from the previous section. We iteratively call $\mathcal{M}_{\mathcal{A}}$ with initial state q_{next}, discarding all V_0 vertices. For every V_1 vertex encountered we switch into state q_E to produce a neighbor until the machine terminates with q_{EOF}. For every neighbor we increment the counter associated by one.

We can now turn to the attractor computation. Assume level $A = A_k^{(0)}$ has already been computed and is available on the second tape. For every vertex v from A we perform the following computation. We copy v to the first tape. Then we move the head of the second tape to the first position after A (indicating the boundary by $\#\#$, say). Now we invoke $\mathcal{M}_{\mathcal{A}}$ with q_{tE}, i.e. we look at the predecessors v' of v ($\mathcal{O}(L)$ for each predecessor). We now scan the content to the left (on the second tape) for an occurrence of v' ($\mathcal{O}(|V|)$). If we find one, we have two options. Either $v' \in V_0$, in which case we advance the head to its right proceeding with the next predecessor, or it is in V_1. Then we decrement the corresponding counter on the first tape ($\mathcal{O}(|V|_1 \cdot L)$). If it reaches 0 we leave v' and proceed with the next predecessor. Otherwise we overwrite v' with the next predecessor. In total these operation require $\mathcal{O}(|V| \cdot L)$. Note that we consider each edge at most one, which means that in effect we need $\sum_{e \in E} |V| \cdot L = |E| \cdot |V| \cdot L$. □

A.3 Representing Winning Conditions

In this section we describe a Turing machine representing the winning condition in the two classes of ω-regular games studied in this paper – Muller and Streett games. We will use this Turing machine as a subroutine in more complex machine, as was done with the machine representing the arena or the machine computing the attractor.

Muller Conditions We now turn to Muller games. Let $\mathfrak{F}$ be given as a propositional formula ϕ in negation normal form (NNF) over the set V of variables. We are going to describe a two-tape Turing machine which decides whether $\{v_1, \ldots, v_k\} \models \phi$ for a given input $\{v_1, \ldots, v_k\} \subseteq V$.

For rigor's sake, define $\|\phi\|$ by induction on the construction of NNF formulas ϕ. If $\phi = v$ or $\phi = \neg v$ for $v \in V$ then $\|\phi\| = 1$. Then define $\|\phi \vee \psi\| = \|\phi\| + \|\psi\| = \|\phi \wedge \psi\|$. The machine $\mathcal{M}$ will have $\mathcal{O}(\|\phi\| \cdot \log_2(|V|))$ states. It will have two tapes. Suppose the input on the first tape is $v_1\#v_2\#\cdots\#v_k$ encoding $\{v_1, \ldots, v_k\} \in V$ (notice that here $v_i \in V \subseteq \mathbb{B}^*$ is a sequence of bits). Then the machine will write a copy of ϕ onto the second tape. To do this it requires $\mathcal{O}(\|\phi\| \cdot \log_2(|V|))$ states. It will then iterate over the input sequence $v_1, \ldots, v_k$ and replace their occurrence with 1 or 0 depending on whether they appear positively or not. If the machine encounters a formula of the form $\psi \vee 1$ it replaces it by 1 and likewise it replaces $\psi \wedge 0$ by 0. In this way it will terminate with the result 1 or 0 of the query depending on whether the set $\{v_1, \ldots, v_k\}$ is a model of ϕ or not. For this it requires $\mathcal{O}(k \cdot \|\phi\| \cdot \log_2(|V|))$ steps.

Streett Conditions Considering Streett games, we take an approach that amounts to writing the Streett pairs on tape. The machine $\mathcal{M}$ does not take any input and simply writes all Streett pairs on tape. This requires $\mathcal{O}(|\Omega| \cdot |V| \cdot \log_2(|V|))$ memory states and the same amount of computation steps.

B Hardness Claims

Streett and Muller Games Consider the arena $\mathcal{A}_n$ as depicted for $n = 6$ in figure 1. Define a propositional formula $\phi_n = \bigwedge_i y_i = 1 \leftrightarrow x_i = 1$ and a Streett condition $\Omega_n = \{(\{y_i = 1\}, \{x_i = 1\}), (\{y_i = 0\}, \{x_i = 0\}) \mid i = 1, \ldots, n\}$. Evidently both conditions have linear size in n. It is now straightforward to verify proposition 3.

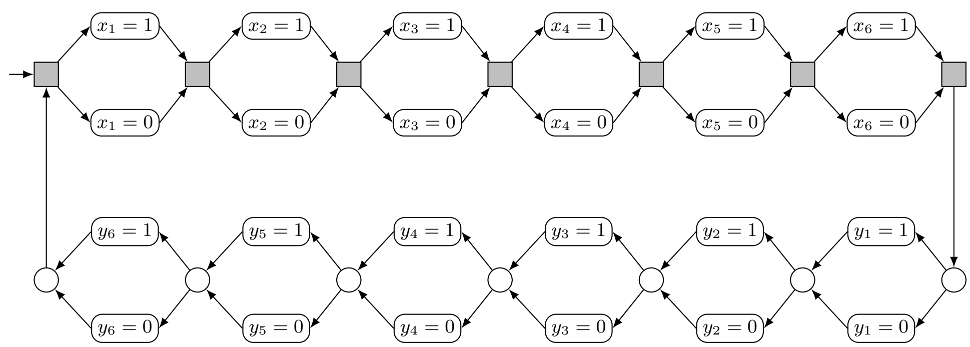

Fig. 1. Arena $\mathcal{A}_6$ for Muller and Streett games.

LTL Games We show the following claim:[8]

Proposition 14. *The class of LTL games is* 2^{2^n}*-hard. The winning conditions of the witnessing family are of size* $\mathcal{O}(n^2)$.

Consider the arena $\mathcal{B}_n$ for $n \in \mathbb{N}$ as depicted in figure 2 for $n = 6$. Player 1 moves from shaded, square vertices. Player 0 moves from all remaining vertices. The idea is that player 1 will play the loop back to a from d a finite number of times. If he plays this loop infinitely many times player 0 wins. Every such sequence between a and d (without any occurrence of a or d in between) is called a *round*. Every round corresponds to an assignment of $x_1, \ldots, x_n$. Finally player 1 moves to b and picks another assignment of $x_1, \ldots, x_n$. Then player 0 must make a decision: If player 1 played an assignment after b which did not occur in any of the rounds before, player 0 must move to 'no'. Otherwise he must move to 'yes'.

We give an LTL formula, which describes this winning condition. Consider the following LTL formulas:

$$\psi_n = \mathbf{F}\big(a \wedge \bigwedge_{i=1}^{n} \mathbf{X}^i x_i = 1 \leftrightarrow \mathbf{F}(b \wedge \mathbf{X}^i x_i = 1)\big)$$

[8] For a definition of the terminology see section 4.

The formula $\varphi_n = \mathbf{G}\neg b \vee ((\mathbf{F}\text{yes}) \leftrightarrow \psi_n)$ defines the winning condition just outlined.

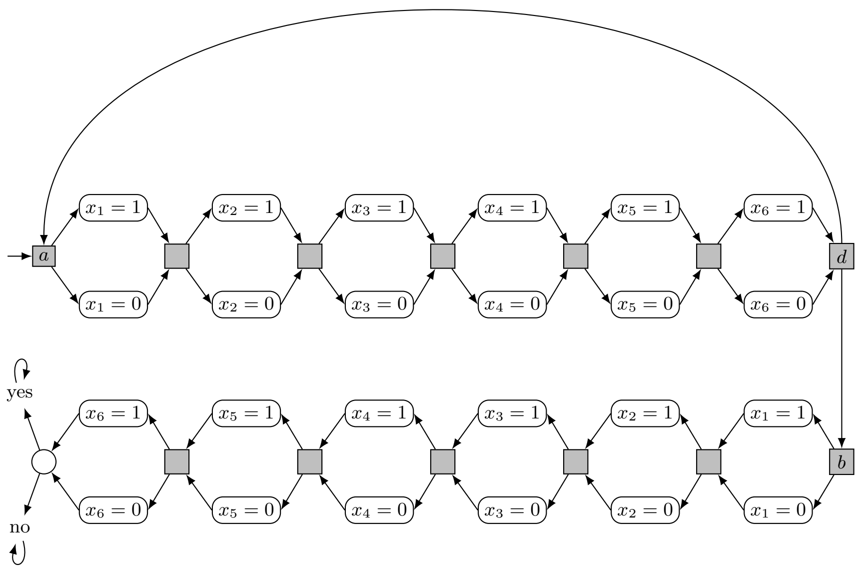

Fig. 2. Arena $\mathcal{B}_6$ for LTL games.

It is left to the reader to verify that any Mealy machine strategy for player 0 in the game $\mathbf{G}_n = (\mathcal{B}_n, \varphi_n)$ must have at least 2^{2^n} states. Since $\mathcal{A}_n$ has $6n + 4 \in \boldsymbol{\mathcal{O}}(n)$ vertices, the family $(\mathbf{G}_n)_{n\geq 1}$ is 2^{2^n}-hard. Note also that $\|\varphi_n\| \in \boldsymbol{\mathcal{O}}(n^2)$.

C The Space Complexity of Solving Muller Games

It is well known that solving Muller games is PSPACE complete [HD05,HD08]. However, for our purposes we want a specific space bound. Below we will give a coarse estimate using Zielonka's algorithm [Zie98].

Proposition 15. *There exists a 2-tape Turing machine $\mathcal{M}$ which on input $\mathcal{A} = (V, E)$ (given as an adjacency matrix), propositional formula ϕ and a vertex $v \in V$ decides in space $\boldsymbol{\mathcal{O}}(|V| \cdot (|V| + \|\phi\|) \cdot \log(|V|))$ whether $v \in \mathcal{W}_0$ in the game $\mathbf{G} = (\mathcal{A}, \phi)$.*

Proof. The machine $\mathcal{M}$ first tests whether $V \models \phi$. This requires space $\|\phi\| \cdot \log(|V|)$. If $V \models \phi$ we continue to work with $\neg\phi$. Hence, we assume that $V \not\models \phi$.

On the second tape, $\mathcal{M}$ now constructs the sequence $(U_i)_{i\geq 0}$ of subsets of V described in section 5.1, starting with $U_0 = \emptyset$. In [Zie98] it is shown that v is in the winning region of player 0 iff $v \in U_i$ for some $i \in \mathbb{N}$. Hence it is sufficient to retain the unions $\bigcup_{j\leq i} U_i$ over all sets U_j computed so far. This requires space $\boldsymbol{\mathcal{O}}(|V| \cdot \log_2(|V|))$. We need the labels for the children $c_0, \ldots, c_{k-1}$ of the root in the Zielonka tree for ϕ. To compute the next such label requires space $|V| + \|\phi\| \cdot \log_2(|V|)$. We start with the label of c_0. Finally we need a counter r, counting the number of times since we last added a vertex to $\bigcup_{j\le i} U_j$. Note that r is bounded by k and so $\log_2(r) \le |V|$.

After i steps (during which the set $U = \bigcup_{j\le i} U_j$ has been computed), we compute the attractor $X = \mathrm{Attr}_0^{\mathcal{A}}(U)$ (requiring additional space of $\boldsymbol{\mathcal{O}}(|V| \cdot \log_2(|V|)))$. Then we consider the 0-trap $\mathcal{B} = \mathcal{A} \setminus X$. We compute $Y = \mathrm{Attr}_1^{\mathcal{B}}(V \setminus \lambda(c_{i \bmod k}))$, again using space $\boldsymbol{\mathcal{O}}(|V| \cdot \log_2(|V|))$. Finally we solve the subgame $(\mathcal{B} \setminus Y, \phi(c_{i \bmod k}))$. Here $\phi(c_{i \bmod k})$ denotes the formula obtained from ϕ by replacing all variables not in $\lambda(c_{i \bmod k})$ by 0.

Observe that the space requirement without the recursive call is in $\boldsymbol{\mathcal{O}}((|V| + \|\phi\|) \cdot \log_2(|V|))$. Observe that if the Zielonka tree for $\phi(c_{i \bmod k})$ has height 0, the space requirement for this step is $\|\phi\| \cdot \log_2(|V|)$, since we only have to check if the (only) node in this tree is labeled with a model of $\phi(c_{i \bmod k})$ or not. Observe furthermore that the tree has height $\le |V|$. Our claim thus follows by induction on the height of the Zielonka tree. $\square$