Strategy Machines and their Complexity (with addendum)

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. Also, lower bound results are obtained.

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 [1], as it essentially preserves the size of the underlying Mealy machine. These machines are known to be large in general [2]. This is reflected in the complexity of solving ω-regular games [3–6]. Optimization of Mealy machines has been investigated [7, 8], 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 [9]. A strategy machine is a multi-tape Turing machine with two distinguished 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

* Supported by DFG research training group 1298, "Algorithmic Synthesis of Reactive and Discrete-Continuous Systems" (AlgoSyn)

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 of roughly equal size (the number of control states).

Introducing this model, we present initial results for two classes of ω-regular games, namely Muller and Streett games. Building on Zielonka's algorithm [10], 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[1], Madhusudan [11] 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 [11].

The paper is an extended abstract of [12]. It 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 [10], using some ideas from [2]. 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 promising, because Streett conditions are more succinct than explicit Muller conditions.

[1] See also the addendum at the end of this paper.

2 Strategy Machines – A Formal Model

The intuition of a *strategy*[2] *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.

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 $(q_O, b_1, \ldots, b_{k+2}) \in Q \times \hat{\mathbb{B}}^{k+2}$. By definition, no transition leads into q_I. The tapes of $\mathcal{M}$ are the* input-output-tape (IO-tape) t_{IO}, *the* computation-tapes $t_{com}^{(i)}$, $1 \leq i \leq k$, *and the* memory-tape t_{mem}.

Hereafter 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

[2] For a definition of a strategy, the reader may want to skip ahead to the next section.

and denoted by $\vdash$. Its transitive closure is denoted by $\vdash^*$. An *iteration* of $\mathcal{M}$ is a sequence $c_1, \dots, c_l$ of configurations, such that for $1 \leq i \leq l-1$ we have $c_i \vdash c_{i+1}$. It starts with $c_1 = (q_I, x, \epsilon, w_{mem}, 0, 0, 0)$ and ends with $c_l = (q_O, y, w, w'_{mem}, h, h', h'')$ for some elements $x, y \in \mathbb{B}^*$, arbitrary words $w, w_{mem}, w'_{mem} \in \mathbb{B}^*$ 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 from c to c' (since q_O has no outgoing transitions and q_I has no incoming transitions). If no such iteration exists, 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 $(c_1, c'_1), \dots, (c_n, c'_n)$, such that $t_{IO}(c_i) = x_i$ and $t_{mem}(c'_i) = t_{mem}(c_{i+1})$. By convention, $t_{mem}(c_1) = \epsilon$. 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 $f(x_1) = t_{IO}(c'_1)$. 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 $b \in A$.

A Mealy machine is a tuple $\mathfrak{M} = (M, \Sigma, m_0, \delta, \tau)$ with *states* M, *IO-alphabet* Σ, *initial state* m_0, *transition function* $\delta\colon M \times \Sigma \to M$ and *output function* $\tau\colon M \times \Sigma \to \Sigma$. By encoding Σ as a subset of $\mathbb{B}^*$ and maintaining a table of triples $(m, x, \delta(m, x))$ and $(m, x, \tau(m, x))$ in the state space, one obtains a strategy machine $\mathcal{M}$ equivalent to $\mathfrak{M}$:

Proposition 1 (Straightforward Simulation). *For every Mealy machine $\mathfrak{M} = (M, \Sigma, m_0, \delta, \tau)$ there exists an equivalent 1-tape strategy machine $\mathcal{M}_{\mathfrak{M}}$ of 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 $T(\mathcal{M}_{\mathfrak{M}}) \in \mathcal{O}(\log_2(|M| \cdot |\Sigma|))$.*

This illustrates the relationship between a strategy machine and a straightforward simulation of a Mealy machine by a Turing machine: The latter can be seen as a special case of the former. Note also that the complexity hidden in the size of Σ is exposed by this simulation.

3 Basics on Games

We fix the notation and terminology on ω-regular games. We assume the reader is familiar with these concepts. An introduction can be found in e.g. [13–15].

An *infinite two player game* (in this paper simply called a *game*) is a tuple $\mathbf{G} = (\mathcal{A}, \varphi)$ with an arena $\mathcal{A} = (V, E)$ and a *winning condition* $\varphi \subseteq V^\omega$. An

arena is a directed graph $\mathcal{A}$ 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. 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 a neighbor of v to each string $w \in V^+$ with $\mathrm{last}(w) = v \in V_i$ (where $\mathrm{last}(\cdot)$ denotes the last element of a sequence). π is *consistent* with σ if for every $n \in \mathbb{N}_0$ 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 if 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 $V = \mathcal{W}_0 \uplus \mathcal{W}_1$ [16]. If $v \in \mathcal{W}_i$, we say *player i wins from v*. A subarena is an induced subgraph $(V', E \cap V' \times V')$ which is again an arena. An i-trap is a subarena from which player i cannot escape. The *i-attractor on $S \subseteq V$* is the set of vertices from which player i can enforce to visit S and is denoted by $\mathrm{Attr}_i^{\mathcal{A}}(S)$. A corresponding strategy is called an *attractor strategy* (see [13, 14]). The complement of an i-attractor is a $(1-i)$-trap. We will need to use the fact that a Turing machine can compute an attractor on S in time polynomial in $|V|$ and $|E|$ (for details see [12]). In this paper we are concerned with only three kinds of winning conditions: Muller, Streett, and LTL conditions. A *Muller condition* is given by a propositional formula ϕ, using the set V as variables. A play $\pi \in V^\omega$ is won by player 0 if the *infinity set* $\mathrm{Inf}(\pi)$ of vertices seen infinitely often in π is a model of ϕ. The equivalent *explicit condition* is $\mathfrak{F} = \{F \subseteq V \mid F \models \phi\}$. It may be exponentially larger than $\|\phi\|$. A game $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$ with a Muller condition $\mathfrak{F}$ is called a *Muller game*. A *Streett condition* is 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 $\mathrm{Inf}(\pi)$ 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, 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[3]. If ϕ is an LTL-formula over the propositions V, a play π is won by player 0 iff $\pi \models \phi$. A game with an LTL condition is called an *LTL game*.

4 Lower Bounds

We investigate lower bounds on the space requirement and latency of any machine implementing a winning strategy for player 0 in certain classes of games. Proofs are omitted in the present abstract, but can be found in [12]. Let $f\colon \mathbb{N} \to$

[3] Introducing LTL is beyond the scope of this paper (see [13, 14]).

$\mathbb{N}$. 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*. A class $\mathcal{C}$ of games is *f-hard* if a *witnessing* f-hard family $(\mathbf{G}_n)_{n\geq 0} \subseteq \mathcal{C}$ exists. In general, this definition allows φ_n to be arbitrarily large. However:

Proposition 2. *The classes of Muller and Streett games are 2^n-hard. The conditions (propositional formulas, set of Streett pairs) of the witnessing families are no larger than $\boldsymbol{\mathcal{O}}(n)$. The class of LTL-games has complexity 2^{2^n} with winning condition in the witnessing family of size $\boldsymbol{\mathcal{O}}(n^2)$.*

A *solution scheme for $\mathcal{C}$* is a mapping $\mathcal{S}$ assigning a strategy machine $\mathcal{S}(\mathbf{G})$ to every $\mathbf{G} \in \mathcal{C}$ which implements a winning strategy for player 0 in $\mathbf{G}$. A function $f\colon \mathbb{N} \to \mathbb{N}$ is *sub-linear* if $f \in \boldsymbol{\mathcal{O}}(x^q)$ for some $q \in (0,1)$. It is well known that, for every $k \in \mathbb{N}$ and every $q \in (0,1)$, one has $\log(x)^k \in \boldsymbol{\mathcal{O}}(x^q)$. Likewise a *sub-exponential function* is a mapping $f\colon \mathbb{N} \to \mathbb{N}$ for which $f \in \boldsymbol{\mathcal{O}}((2^x)^q) = \boldsymbol{\mathcal{O}}(2^{qx})$ for some $q \in (0,1)$. All polynomial functions are sub-exponential. For a proof of the following theorem, see [12]:

Theorem 1 (Lower Bounds).

- *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 latency or space requirement sub-linear in $|V|$. In particular, for no $k \in \mathbb{N}$ can there be such a scheme with latency or space requirement in $\log_2(|V|)^k$.*
- *There is no solution scheme for LTL-games, assigning a strategy machine with latency or space requirement sub-exponential in $|V|$ to every LTL-game $\mathbf{G} = ((V,E),\varphi)$. In particular, there can be no such scheme with latency or space requirement polynomial in $|V|$.*

5 Muller and Streett Games

In this section we outline (for formal proofs see [12]) how to construct a polynomial sized strategy machine implementing a winning strategy in Muller and Streett games. For Streett games we even get a machine with a latency polynomial in the winning condition. We will consider a Muller game $\mathbf{G} = (\mathcal{A}, \phi)$, $\mathcal{A} = (V,E)$, with a propositional formula ϕ and explicit condition $\mathfrak{F}$.

Our construction is based on Zielonka's algorithm [10], while also using ideas from [2]. We will define some notation, which will be used subsequently. If $V \in \mathfrak{F}$, we define the set $\max(\mathfrak{F}) = \{V' \subseteq V \mid V' \notin \mathfrak{F} \wedge \forall V'' \supsetneq V' :\ V'' \in \mathfrak{F}\}$ of all maximal subsets of V *not* in $\mathfrak{F}$. If $X \subseteq V$, let $\mathfrak{F} \restriction X := \mathfrak{F} \cap \mathscr{P}(X)$ be the *restriction* of $\mathfrak{F}$ to subsets contained in X. The *Zielonka tree* $\mathcal{Z} := \mathcal{Z}_{\mathfrak{F}} = (N, \lambda)$ of $\mathfrak{F}$ is a labeled tree $N \subseteq \mathbb{N}^*$ with labels $\lambda(x) \in \mathscr{P}(V)$ for all $x \in N$. ε is the root, 1 is the first child of the root, 12 is the second child of 1, and so forth[4]. The root has label $\lambda(\epsilon) = V$. Let $x \in N$ with label $X = \lambda(x)$. If $X \in \mathfrak{F}$ then x

[4] We assume that, if $n \cdot a \in N$, then $n \cdot b \in N$ for all $1 \leq b \leq a$.

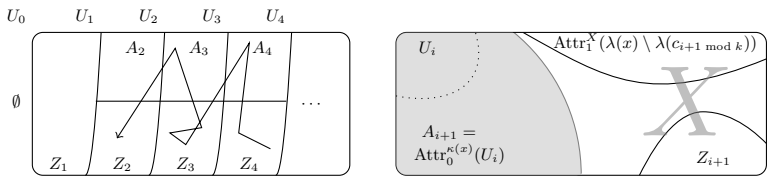

Fig. 1. A play visiting U_i stays in Z_i or is "pulled" into U_{i-1}

has $k = |\max(\mathfrak{F} \restriction X)|$ children $c_0, \ldots, c_{k-1}$, each labeled with a distinct element from $\max(\mathfrak{F} \restriction X)$. If $X \notin \mathfrak{F}$, we apply the same construction with respect to $\mathfrak{F}^{\mathscr{C}}$. The tree obtained in this way is of depth $\leq |V|$. A node labeled with a set $X \in \mathfrak{F}$ is called a 0-level node. The remaining nodes are 1-level nodes.

We will use an additional labeling function κ, which assigns (possibly empty) subarenas to the nodes in N. The idea of attaching the subarenas to the nodes in $\mathcal{Z}$ is elaborated in [2]. Given $x \in N$, $\kappa(x)$ is a subarena with the property that player 0 can win the subgame $(\kappa(x), \mathfrak{F} \restriction \lambda(x))$ from every vertex in $\kappa(x)$. The computation of κ is quite involved and only sketched here (see [2]). If x is 0-level with label $\kappa(x)$ and if c is a child of x in $\mathcal{Z}$, the construction is an attractor computation $\kappa(c) = \kappa(x) \setminus \mathrm{Attr}_0^{\kappa(x)}(\lambda(x) \setminus \lambda(c))$. If x is 1-level, one constructs an increasing sequence $(U_i^x)_{i \geq 0}$ of sets of "finished vertices" with $U_0^x = \emptyset$. Let $c_0, \ldots, c_{k-1}$ be the children of x in $\mathcal{Z}$. To compute U_{i+1}^x we remove $A_{i+1} = \mathrm{Attr}_0^{\kappa(x)}(U_i)$ from $\kappa(x)$ and obtain X. Then we remove $\mathrm{Attr}_1^X(\lambda(x) \setminus \lambda(c_{i+1 \bmod k}))$ obtaining Y. Now Z_{i+1}^x is taken to be the winning region in the subgame $(Y, \mathfrak{F} \restriction \lambda(c_{i+1 \bmod k}))$. Then $U_{i+1}^x = A_{i+1} \uplus Z_{i+1}^x$. The label of c_i is $\kappa(c_i) = \bigcup_{j \equiv i \pmod k} Z_j^x$. The intuition is that, once a play reaches U_i^x for some i, it must either stay in Z_i^x or will be forced into U_{i-1}^x. This can only happen finitely many times, so the play will eventually stay in one $Z_i^x \subseteq \lambda(c_{i \bmod k})$ (i.e. in the subtree below $c_{i \bmod k}$) or it must leave the entire subarena $\kappa(x)$ (i.e. the subtree under x). The situation is depicted in Fig. 1.

Lemma 1 ([10, 2]). *In the above notation, if $\mathcal{W}_0 = V$ then $\kappa(x) = \bigcup_i U_i^x$. Furthermore, player 0 wins from every $v \in Z_i^x$ in the subgame $(Z_i^x, \mathfrak{F} \restriction \lambda(c_{i \bmod k}))$.*

The sequence $(U_i^x)_{i \geq 1}$ becomes stable after $\leq k \cdot |V|$ steps. We use the superscript x to indicate the (1-level) node for which the sequence has been computed.

We will now describe an *adaptive* strategy machine $\mathcal{M}$ implementing a winning strategy for Muller games. By "adaptive" we mean that the machine continuously refines the memory state it maintains, until eventually a sufficient amount of information is computed to win the play. As a consequence, we must prevent player 0 from inadvertently leaving $\mathcal{W}_0$ during the adaption phase. Hence, we assume $\mathcal{W}_0 = V$, which can be achieved by preprocessing. Since we are interested in representing a winning strategy succinctly, not in deciding the game, this is no restriction. $\mathcal{M}$ will store paths in $\mathcal{Z}$ on its memory tape. By a path we mean one that begins in the root and ends in a leaf. In addition, $\mathcal{M}$ will store

the labels $\lambda(x)$ and $\kappa(x)$ for a given node x on the path. Say the current path is $p = p(0), \dots, p(m)$. Given an input $v \in V$ the machine will then traverse the path and find the unique lowest node $p(i)$ such that $v \in \kappa(p(i))$. The idea is to then play according to the classical Zielonka strategy: If $p(i)$ is 0-level, $\mathcal{M}$ plays an attractor strategy on the set $\lambda(p(i)) \setminus \lambda(p(i+1))$. Otherwise, it will compute the minimal t with $v \in U_t^{p(i)}$ (which exists by Lem. 1) and either update the memory state (if $v \in Z_t^{p(i)}$ to a path through the $(t \bmod k)$-th child of $p(i)$, where k is the number of children of $p(i)$) or it will attract to $U_{t-1}^{p(i)}$.

Unfortunately, since $\kappa(c) = \bigcup_{i \equiv j \pmod k} Z_i^x$ is expensive to compute if c is the j-th child of a 1-level node x, the memory update in this approach is costly. Computing the sequence $(Z_i^x)_{i \geq 1}$ requires solving a subgame for every i. Also, to retain all of this information, either in the control states or on the memory tape, would yield a controller of exponential size (or exponential space requirement) in $|V|$. Therefore the idea is to spread the computation of the sets $(U_i^x)_{i \geq 0}$ and $(Z_i^x)_{i \geq 1}$ across multiple iterations of $\mathcal{M}$. This is achieved using an auxiliary Turing machine $\mathcal{M}_\kappa$, which on input $\lambda(x) \in \mathfrak{F}$ and $\kappa(x)$ computes the number k_x of children of x and $(U_i^x)_{i \geq 0}$, $(Z_i^x)_{i \geq 1}$, or rather a compact representation of the same (see below). We attach configurations c of $\mathcal{M}_\kappa$ to the parent of those 0-level nodes x, for which $\kappa(x)$ is currently being computed. We will store the labels $\kappa(x)$ where they have been computed. For 1-level nodes we also store $(U_i^x)_{i \geq 0}$, $(Z_i^x)_{i \geq 1}$ (compactly represented), once available.

There are two obstacles: First, the path may need to be updated, whereby some sets $\kappa(p(i))$ already computed are lost. In Lem. 2 we show that this is tolerable. The second problem is storing the sequence $(U_i^x)_{i \geq 0}$. It may contain exponentially many sets (recall that it becomes stationary after $\leq k \cdot |V|$ steps, where the number k of children of x may be exponential in $|V|$). In the next paragraph, we give a compact representation of the sequence $(U_i^x)_{i \geq 0}$ which allows us to recompute the sets Z_i^x and U_i^x efficiently for any i.

If $D_i^x = U_i^x \setminus U_{i-1}^x$ then $D_i^x \neq \emptyset$ for at most $|V|$ indices i. Given $(D_i^x)_{i \geq 1}$, one obtains Z_i^x by computing $U_{i-1}^x = \bigcup_{j=1}^{i-1} D_i^x$ and $A = \mathrm{Attr}_0^{\kappa(x)}(U_{i-1}^x)$. Then $Z_i^x = U_i^x \setminus A$. If c is the j-th child out of k_x children of x, then $\kappa(c) = \bigcup_{i \equiv j \pmod{k_x}} Z_i$. Thus, it is sufficient to store the number k_x of children of x as well as $\mathcal{D}^x = \{(D_t^x, \mathsf{bin}(t)) \mid D_t^x \neq \emptyset\}$, where $\mathsf{bin}(t)$ is the binary representation of t, in order to compute Z_i^x and $\kappa(c)$ for any child c of x and any i. This representation requires space in $\mathsf{poly}(|V|)$. Computing Z_i^x and the label $\kappa(c)$ requires time in $\mathsf{poly}(|V|, |E|)$.

The memory tape will contain a labeled path p in $\mathcal{Z}$, written as a sequence of triples $p(i) = (n_i, \lambda(p,i), \theta(p,i))$. The i-th node is $n_1 \cdots n_i \in N \subseteq \mathbb{N}^*$, which, by abuse of notation, is denoted by $p(i)$ as well. $p(0)$ is always the root. By convention, $n_0 = 0$. We let $\lambda(p,i) = \lambda(n_1 \cdots n_i)$. For $\theta(p,i)$ we have three possibilities. $\theta(p,i) = \kappa(p,i)$ if this label has already been computed and if $p(i)$ is 0-level. If $p(i)$ is 1-level with $k_{p(i)}$ children, then either $\theta(p,i) = (\kappa(p,i), \mathcal{D}^{p(i)}, k_{p(i)})$, if the set $\mathcal{D}^{p(i)}$ has been computed, or $\theta(p,i) = (\kappa(p,i), c)$ for some configuration c of $\mathcal{M}_\kappa$, otherwise. Finally, $\theta(p,i) = \emptyset$ if the computation has not proceeded this far down the path (note that invoking $\mathcal{M}_\kappa$ on any node $n_1 \cdots n_i$ requires

$\kappa(n_1 \cdots n_{i-1})$). We will refer to p as a *memory path* (see Fig. 2). Storing p requires space in $\mathsf{poly}(|V|, S_{\mathcal{M}_\kappa})$, where $S_{\mathcal{M}_\kappa}$ is the space requirement of $\mathcal{M}_\kappa$.

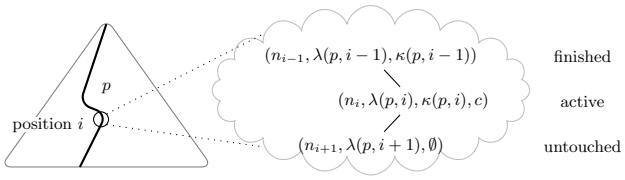

Fig. 2. A Memory Path p

A node with $\theta(p, i) = \emptyset$ is called *untouched*. One with $\theta(p, i) = (\kappa(p, i), c)$ is called *active* (note that it must be 1-level). Nodes which are neither active nor untouched are *finished* and necessarily have a κ-label. We require that, for any memory path p, either all nodes are finished or there exists precisely one node which is active. Since $\kappa(p, i)$ can only be computed when $\kappa(p, i-1)$ is available, any memory path p with an active node $p(i)$ satisfies that all $p(j)$ with $j < i$ are finished and all $p(r)$ with $r > i$ are untouched.

In addition to the auxiliary machine $\mathcal{M}_\kappa$ we will also make use of an auxiliary machine $\mathcal{M}_\lambda$, which computes the labeling function λ. Both auxiliary machines $\mathcal{M}_\lambda$ and $\mathcal{M}_\kappa$ are used implicitly, as subroutines, by $\mathcal{M}$. Their runtime and space requirement affect the bound on the latency and space requirement of $\mathcal{M}$ we give below[5]. We will assume $\mathcal{M}$ has access to a representation of $\mathcal{A}$ and of the winning condition ϕ in its state space. We do not give the exact construction here (see [12]), but we point out that this requires a number of control states and query time in $\mathsf{poly}(|V|, \|\phi\|)$.

Formally, $\mathcal{M}$ proceeds as follows. The initial memory path p_I is obtained by setting $n_0 = \cdots = n_r = 0$. We choose r, such that $n_1 \cdots n_r \in N$ 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 $i = 0$, 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 path 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 p_I satisfies this invariant. Additionally, we assume that for any memory path p either all sets $\kappa(p, i)$ are computed or there exists an active node. The memory update will also preserve this invariant.

Suppose the input to $\mathcal{M}$ is $v \in V$ and the current path is p of length $\|p\|$. By assumption, the highest node in p for which κ has not been computed is 0-level. Let i be the maximal index with $\theta(p, i) \neq \emptyset$ and $v \in \kappa(p(i))$, i.e. $\kappa(p, i)$ has been computed and contains v. We call this node the *NOI (node of interest)*. Finding

[5] Note that $\mathcal{M}_\lambda$ may need $\mathbf{\Omega}(2^{|V|})$ steps to produce a label. Depending on ϕ, this can be improved (see results on Streett games, Thm. 3). We cannot spread out this computation over several iterations (c.f. Rem. 1 on page 11).

the NOI is in $\mathsf{poly}(|V|)$, as the space requirement of a triple $(n_i, \lambda(p,i), \theta(p,i))$ is independent of $S_{\mathcal{M}_\kappa}$ if $p(i)$ is finished. 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 a number of steps in $\mathsf{poly}(|V|, |E|)$.
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. We then compute $\kappa(p, i+1)$ from $\mathcal{D}^{p(i)}$. If $i+1 < \|p\|$, then we also compute $\kappa(p, i+2)$. This is an attractor on the set $\lambda(p, i+1) \setminus \lambda(p, i+2)$, as $p(i+1)$ is 0-level. Finally, if even $i+2 < \|p\|$, we set $p(i+2)$ active, i.e. we replace $\theta(p, i+2) = \emptyset$ by $(\kappa(p, i+2), c_I)$. All these steps need time in $\mathsf{poly}(|V|, |E|)$.
For the next move, we distinguish $v \notin \kappa(p, i+1)$ and $v \in \kappa(p, i+1)$. If $v \notin \kappa(p, i+1)$, $\mathcal{M}$ computes the minimal t with $v \in U_t^{p(i)}$ (note that this can be done while computing $\kappa(p, i+1)$ 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, $\mathcal{M}$ outputs according to the corresponding attractor strategy. Otherwise, $\mathcal{M}$ updates its memory state to a path through the corresponding child of $p(i)$. This requires $\boldsymbol{\mathcal{O}}(|V| \cdot T_{\mathcal{M}_\lambda})$ steps, where $T_{\mathcal{M}_\lambda}$ is the runtime of $\mathcal{M}_\lambda$. If $v \in \kappa(p, i+1)$, $\mathcal{M}$ outputs an arbitrary neighbor of v in $\kappa(p, i+1)$.

(b) $\kappa(p, i+1)$ has been computed. We proceed as the usual Zielonka strategy indicates: If $p(i)$ is 0-level, we play an attractor strategy on the set $\lambda(p,i) \setminus \lambda(p, i+1)$, updating the memory path if necessary. 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 we update the memory to the leftmost path through the unique child c of $p(i)$ with $Z_t \subseteq \kappa(c)$ and output an arbitrary neighbor of v in $\kappa(c)$. Again, all of this is feasible in time in $\mathsf{poly}(|V|, |E|, T_{\mathcal{M}_\lambda})$.

The proofs of the following two lemmas can be found in [12]:

Lemma 2. *$\mathcal{M}$, as described above, implements a winning strategy for player 0.*

Note that the size of $\mathcal{M}$ depends only on the representations of $\mathcal{A}$ and ϕ it uses, as well as on $\|\mathcal{M}_\lambda\|$ and $\|\mathcal{M}_\kappa\|$. Those representations can be implemented with a polynomial number of control states. All other parts of the machine are independent of the size of $\mathcal{A}$ or ϕ. Let $T_{\mathcal{M}_\lambda}$ and $S_{\mathcal{M}_\kappa}$ denote the runtime of $\mathcal{M}_\lambda$ and the space requirement of $\mathcal{M}_\kappa$.

Lemma 3. *Let $\mathbf{G} = (\mathcal{A}, \phi)$ be a Muller game, where $\mathcal{A} = (V, E)$ and ϕ is a propositional formula. There exists a strategy machine $\mathcal{M}$ of size $\|\mathcal{M}\| \in$*

$\mathsf{poly}(|V|, \|\phi\|)$, *space consumption* $S(\mathcal{M}) \in \mathsf{poly}(|V|, S_{\mathcal{M}_\kappa})$, *and latency* $T(\mathcal{M}) \in \mathsf{poly}(|V|, |E|, T_{\mathcal{M}_\lambda})$ *which implements a winning strategy for player 0.*

Using that solving Muller games is in PSPACE [3, 4, 17] we can show $S_{\mathcal{M}_\kappa} \in \mathsf{poly}(|V|, \|\phi\|)$ (see [12]). Also, $T_{\mathcal{M}_\lambda} \in \boldsymbol{\mathcal{O}}\big(|V| \cdot \log_2(|V|) \cdot \max\{|\mathfrak{F}|, |\mathfrak{F}^{\mathscr{C}}|\}\big)$, where $\mathfrak{F}$ is the explicit condition for ϕ. For a proof of the next theorem, see [12]:

Theorem 2. *For any Muller game* $\mathbf{G} = (\mathcal{A}, \mathfrak{F})$*, where* $\mathcal{A} = (V, E)$ *and* $\mathfrak{F}$ *is given by a propositional formula* ϕ*: There exists a strategy machine* $\mathcal{M}$ *of size* $\|\mathcal{M}\| \in \mathsf{poly}(|V|, \|\phi\|)$*, space consumption* $S(\mathcal{M}) \in \mathsf{poly}(|V|, \|\phi\|)$*, and latency* $T(\mathcal{M})$ *polynomial in* $|V| + |E|$ *and linear in* $\max\{|\mathfrak{F}|, |\mathfrak{F}^{\mathscr{C}}|\}$*.*

Remark 1. Unlike the situation of computing κ, we cannot spread out the computation of $\mathcal{M}_\lambda$ across the infinite play, because we do not have a compact representation of the sets $\lambda(x_0), \ldots, \lambda(x_k)$ (if $x_0, \ldots, x_k$ are siblings). For $\kappa(x_i)$ the set $\mathcal{D}^x$ provides such a compact representation.

Muller games are usually solved via *latest appearance records* [18, 17], yielding a Mealy machine of size $|V| \cdot |V|!$. The straightforward simulation (c.f. Prop. 1) of this machine is of size $|V|^2 \cdot |V|!$. The size we obtain is exponentially lower, at the price of an exponentially longer latency. For Streett games we can improve the bound on $T_{\mathcal{M}_\lambda}$. Every Streett condition Ω is equivalent to $\mathfrak{F}_\Omega = \{X \subseteq V \mid X \text{ violates no pair in } \Omega\}$. We can show (for details, see [12]):

Proposition 3. *Let* Ω *be a Streett condition. Then there exists a machine* $\mathcal{M}_\lambda$ *computing* λ *with runtime polynomial in* $|V|$ *and* $|\Omega|$*.*

With Prop. 3 and Lem. 3 one shows (see [12]):

Theorem 3. *Let* $\mathbf{G} = (\mathcal{A}, \Omega)$ *be a Streett game. Then there exists a strategy machine* $\mathcal{M}$ *of size* $\|\mathcal{M}\| \in \mathsf{poly}(|V|, |\Omega|)$*, space consumption* $S(\mathcal{M}) \in \mathsf{poly}(|V|, |\Omega|)$ *and latency* $T(\mathcal{M}) \in \mathsf{poly}(|V|, |\Omega|)$*.*

Streett games are usually solved using *index appearance records* [19, 13]. This yields a Mealy machine with $|\Omega|^2 \cdot |\Omega|!$ states. The straightforward simulation (Prop. 1) gives a strategy machine of size $\boldsymbol{\mathcal{O}}(|\Omega|^2 \cdot |\Omega|! \cdot |V|)$ and latency $\boldsymbol{\mathcal{O}}(|\Omega|)$. We significantly reduce the size while the latency remains polynomial in $|\Omega|$.

6 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 with polynomial space requirement implementing winning strategies in Muller games. 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 other kinds of ω-regular games. We have partial results on Request-Response games. Considering latency, space requirement, and size, we would like to find relations between the parameters indicating the nature of a trade-off. Also, infinite-state strategies lend themselves to a closer study based on our model. Finally, using an adaptive strategy raises the question of how long the adaption takes. We try to address this in current research.

Acknowledgments The author would like thank Wolfgang Thomas and Christof Löding for their helpful comments and suggestions.

Addendum In this note, we repair an omission of the paper above and address related work that appears in the last section of [2]. There *p-automata*, a kind of synchroznied product of finite automata, are used to model strategies, and a procedure to find polynomial sized p-automata, based on Zielonka's construction, is outlined. If the latency of a strategy machine is bounded, there is a translation of strategy machines to p-automata preserving polynomial size. While both approaches use Zielonka's construction, our contribution is a fine grained analysis of all complexities involved (latency, space, requirement, and size, all parameterized by representations of arena and winning condition), an adaptive strategy machine, and a differentiation between *static* memory (control states) and *dynamic* memory (memory tape) of a strategy.

References

1. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from psl. ENTCS **190** (November 2007) 3–16
2. Dziembowski, S., Jurdzinski, M., Walukiewicz, I.: How much memory is needed to win infinite games? In: LICS 97, Washington, DC, USA, IEEE Computer Society
3. Hunter, P., Dawar, A.: Complexity bounds for regular games (extended abstract). In: Symposium on Mathematical Foundations of Computer Science (MFCS). (2005)
4. Hunter, P., Dawar, A.: Complexity bounds for muller games. Theoretical Computer Science (TCS) (2008) Submitted.
5. Horn, F.: Explicit muller games are ptime. In: FSTTCS. (2008) 235–243
6. Emerson, E.A., Jutla, C.S.: The complexity of tree automata and logics of programs. SIAM J. Comput. **29** (September 1999) 132–158
7. Holtmann, M., Löding, C.: Memory reduction for strategies in infinite games. In: CIAA. (2007) 253–264

8. Gelderie, M., Holtmann, M.: Memory reduction via delayed simulation. In: iWIGP. (2011) 46–60
9. Grohe, M., Hernich, A., Schweikardt, N.: Lower bounds for processing data with few random accesses to external memory. J. ACM **56** (May 2009) 12:1–12:58
10. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theor. Comput. Sci. **200** (June 1998) 135–183
11. Madhusudan, P.: Synthesizing reactive programs. In: Comp. Sci. Log., CSL 2011, Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2011) 428–442
12. Gelderie, M.: Strategy machines and their complexity. Technical Report AIB-2012-04, RWTH Aachen University, (2012) http://sunsite.informatik.rwth-aachen.de/Publications/AIB/2012/2012-04.pdf
13. Grädel, E., Thomas, W., Wilke, T., eds.: Automata logics, and infinite games: a guide to current research. Springer-Verlag, New York, NY, USA (2002)
14. Löding, C.: Infinite games and automata theory. In Apt, K.R., Grädel, E., eds.: Lectures in Game Theory for Computer Scientists. Cambridge UP (2011)
15. Perrin, D., Pin, J.: Infinite words: automata, semigroups, logic and games. Pure and applied mathematics. Elsevier (2004)
16. Büchi, J.R., Landweber, L.H.: Solving Sequential Conditions by Finite-State Strategies. Trans. of the AMS **138** (1969) 295–311
17. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic **65**(2) (1993) 149 – 184
18. Gurevich, Y., Harrington, L.: Trees, automata, and games. Proceedings of the fourteenth annual ACM symposium on Theory of computing (1982) 60–65
19. Safra, S.: Exponential determinization for ω-automata with strong-fairness acceptance condition (extended abstract). STOC '92 (1992) 275–282