Higher Order Pushdown Automata, the Caucal Hierarchy of Graphs and Parity Games *

Thierry Cachat

Lehrstuhl für Informatik VII, RWTH, D-52056 Aachen
Fax: (49) 241-80-22215, cachat@informatik.rwth-aachen.de
http://www-i7.informatik.rwth-aachen.de/~cachat/publi.html

Abstract We consider two-player parity games played on transition graphs of higher order pushdown automata. They are "game-equivalent" to a kind of model-checking game played on graphs of the infinite hierarchy introduced recently by Caucal. Then in this hierarchy we show how to reduce a game to a graph of lower level. This leads to an effective solution and a construction of the winning strategies.

1 Introduction

Games on finite graphs have been intensively studied for many years and used for modeling reactive systems. In the last years, two-player games on simple classes of infinite graphs have attracted attention. Parity games on pushdown graphs were solved by Walukiewicz in [18] using a reduction to finite graphs and a refined winning condition involving claims for one player (see also [2]). Kupferman and Vardi used two-way alternating automata in [15,17] to give a solution of parity games on the more general class of prefix recognizable graphs (see also [3]). In this framework, a solution means that given the finite description of the game, an algorithm should determine the winner and compute a winning strategy. The model checking problem is equivalent to the question of determining the winner: given a graph and a μ-calculus formula, one can construct a parity game such that the first player wins if and only if the formula is satisfied in the graph. In this framework of game also weaker logics and winning conditions have been studied, see among others [1,6,10,14].

In the present paper we consider a generalization to higher order pushdown automata for defining the game graph, where the player and the priority of a configuration are determined by the control state. We consider also the infinite hierarchy of graphs defined recently by Caucal [5] from the finite trees using inverse mapping and unfolding. The paper has two main contributions: an equivalence via game-simulation between higher order pushdown automata and the Caucal graphs, and an effective solution of parity games on both of these types of graphs. Using this game-simulation we show how to translate a game on a higher order pushdown automaton to a kind of model-checking game on a Caucal graph; one can then reduce such a game to a game on a graph from a lower level of the hierarchy and finally to a parity game on a finite graph, which gives an effective solution. It is then possible to reconstruct a wining strategy for the original game. As far as we know this is the first result in this direction. So far only the decidability of MSO-properties of these graphs was known [5,13].

In the next section we define the different models of graphs and automata considered. Then we present in terms of game-simulation the reduction from higher order pushdown automata to the Caucal graphs and vice versa. In Section 4 we show that a game on a Caucal graph can be

* A preliminary version of this article was accepted at the conference ICALP 2003, Eindhoven. To appear in LNCS

reduced to an equivalent game on a graph of lower level. For this we use a generalization of ideas from [17] to trees of infinite degree: the construction of an alternating one-way tree automaton equivalent to a given two-way alternating automaton. The main result that we use without proof is the positional (memoryless) determinacy of parity games of [8]: from any configuration one of the players has a positional winning strategy. We assume that the reader is familiar with the basic notions of language theory, automata, graphs and games (see [11] for an overview).

2 The Models

We note $[max] = \{0, \cdots, max-1\}$ for an integer $max > 0$. We write regular expressions in the usual way, for example $(a+b)^*c$ for letters a, b, c from a (finite) alphabet Γ. The empty word is ε and $\Gamma^{\leqslant 3} := \varepsilon + \Gamma + \Gamma^2 + \Gamma^3 = \bigcup_{i \leqslant 3} \Gamma^i$.

2.1 Parity games

A *game structure* is a tuple (V_0, V_1, E, Ω), where $V = V_0 \uplus V_1$ is a set of vertices partitioned into vertices of Player 0 and vertices of Player 1, $E \subseteq V \times V$ is a set of edges (directed, unlabeled), and $\Omega : V \to [max]$ is a priority function assigning to each vertex an integer between 0 and $max - 1$, with $max > 0$. Starting in a given initial vertex $\pi_0 \in V$, a play in (V_0, V_1, E, Ω) proceeds as follows: if $\pi_0 \in V_0$, Player 0 picks the first transition (move) to π_1 with $\pi_0 E \pi_1$, else Player 1 does, and so on from the new vertex π_1. A play is a (possibly infinite) maximal sequence $\pi_0\pi_1\cdots$ of successive vertices. For the winning condition we consider the min-parity version: Player 0 wins the play $\pi_0\pi_1\cdots$ iff $\liminf_{k\to\infty} \Omega(\pi_k)$ is even, *i.e.*, iff the minimal priority seen infinitely often in the play is even. If the play is finite because of a deadlock, then by convention the player who should play loses immediately.

A *strategy* for Player 0 is a function associating to each prefix $\pi_0\pi_1\cdots\pi_n$ of a play such that $\pi_n \in V_0$ a "next move" π_{n+1} with $\pi_n E \pi_{n+1}$. A strategy is *positional* (or memoryless) if it depends only on the current vertex π_n. We say that Player 0 wins the game from the initial vertex π_0 if he has a *winning strategy* for this game: a strategy such that he wins every play.

A game structure (V_0, V_1, E, Ω) is *game-simulated* by another game structure $(V'_0, V'_1, E', \Omega')$ from initial vertices $\pi_0 \in V$ and $\pi'_0 \in V'$ if

- Player 0 wins the game $(V'_0, V'_1, E', \Omega')$ from π'_0 iff Player 0 wins the game (V_0, V_1, E, Ω) from π_0,
- from a winning strategy of Player 0 in $(V'_0, V'_1, E', \Omega')$ one can compute a winning strategy of Player 0 in (V_0, V_1, E, Ω).

2.2 Higher Order Pushdown System

We recall the definition from [13] (which is equivalent to the one from [9]), where we slightly change the terminology. A *level 1 store* (or *1-store*) over an alphabet Γ is an arbitrary sequence $[a_1, \cdots, a_\ell]$ of elements of Γ, with $\ell \geqslant 0$. A *level n store* (or *n-store*), for $n \geqslant 2$, is a sequence $[s_1, \cdots, s_\ell]$ of $(n-1)$-stores, where $\ell \geqslant 0$. We allow a store to be empty. The following operations can be performed on 1-store:

$$\begin{aligned}
push_1^a([a_1, \cdots, a_{\ell-1}, a_\ell]) &:= [a_1, \cdots, a_{\ell-1}, a_\ell, a] \text{ for all } a \in \Gamma \ , \\
pop_1([a_1, \cdots, a_{\ell-1}, a_\ell]) &:= [a_1, \cdots, a_{\ell-1}] \ , \\
top([a_1, \cdots, a_{\ell-1}, a_\ell]) &:= a_\ell \ .
\end{aligned}$$

If $[s_1, \cdots, s_l]$ is a store of level $n > 1$, the following operations are possible:

$$\begin{aligned}
push_n([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= [s_1, \cdots, s_\ell, s_\ell] \ , \\
push_k([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= [s_1, \cdots, push_k(s_\ell)] \text{ if } 2 \leqslant k < n \ , \\
push_1^a([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= [s_1, \cdots, push_1^a(s_\ell)] \text{ for all } a \in \Gamma \ , \\
pop_n([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= [s_1, \cdots, s_{\ell-1}] \ , \\
pop_k([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= [s_1, \cdots, s_{\ell-1}, pop_k(s_\ell)] \text{ if } 1 \leqslant k < n \ , \\
top([s_1, \cdots, s_{\ell-1}, s_\ell]) &:= top(s_\ell) \ .
\end{aligned}$$

The operation pop_k is undefined on a store, whose top store of level k is empty. Similarly *top* is undefined on a store, whose top 1-store is empty. Given Γ and n, the set Op_n of operations (on a store) of level n consists of:

$$push_k \text{ for all } 2 \leqslant k \leqslant n,\ push_1^a \text{ for all } a \in \Gamma, \text{ and } pop_k \text{ for all } 1 \leqslant k \leqslant n.$$

A *higher order pushdown system* of level n (or n-HPDS) is a tuple $H = (P, \Gamma, \Delta)$ where P is the finite set of control locations, Γ the finite store alphabet, and $\Delta \subseteq P \times \Gamma \times P \times Op_n$ the finite set of (unlabeled) transition rules. A *configuration* of an n-HPDS H is a pair (p, s) where $p \in P$ and s is an n-store. The set of n-stores is denoted $\mathscr{S}_n$. We do not consider HPDS as accepting devices, hence there is no input alphabet. A HPDS $H = (P, \Gamma, \Delta)$ defines a *transition graph* (V, E), where $V = \{(p, s) : p \in P, s \in \mathscr{S}_n\}$ is the set of all configurations, and

$$(p, s)E(p', s') \iff \exists (p, a, p', \sigma) \in \Delta : top(s) = a \text{ and } s' = \sigma(s) \ .$$

Note that if the top 1-store is empty, no transition is possible. If necessary one can add a "bottom store symbol" $\perp \in \Gamma$ and define explicitly the corresponding transitions, such that it cannot be erased. To define a parity game on the graph of a HPDS, we assign a priority and a player to each control state, and we consider an initial configuration: a *game structure on a HPDS H* is a tuple $\mathscr{G} = (H, P_0, P_1, \Omega, s_0)$, where $P = P_0 \uplus P_1$ is a partition of the control states of H, $\Omega : P \longrightarrow [max]$ is a priority function, and $s_0 \in V$. This extends naturally to a partition of the set of configurations and to a priority function defined on this set: with the notations of Section 2.1, $V_0 = P_0 \times \mathscr{S}_n$, $V_1 = P_1 \times \mathscr{S}_n$, $\Omega((p, s)) = \Omega(p)$, and E is defined above.

2.3 Caucal Hierarchy

We recall the definitions from [5]. Let L be a countable set of symbols for labeling arcs. A graph is here simple, oriented and arc labeled in a finite subset of L. Formally, a *graph* G is a subset of $V \times L \times V$, where V is an arbitrary set and such that its *label* set

$$\begin{aligned}
L_G &:= \{a \in L \mid \exists s, t : (s, a, t) \in G\} && \text{is finite, but its vertex set} \\
V_G &:= \{s \mid \exists a, t : (s, a, t) \in G \vee (t, a, s) \in G\} && \text{is finite or countable.}
\end{aligned}$$

We write also $t \xrightarrow[G]{a} s$ (or $t \xrightarrow{a} s$) for $(t, a, s) \in G$.

A *finite graph* is a graph whose vertex set is finite. A *tree* is a graph where each vertex has at most one predecessor, the unique root has no predecessor, and each vertex is accessible from the root. A vertex labeled tree is a tree, with a labeling function associating to each node a letter from a finite alphabet. The *unfolding* of a graph G is the following forest (set of trees):

$$Unf(G) := \{ws \xrightarrow{a} wsat : w \in (V_G \cdot L_G)^* \wedge s \xrightarrow[G]{a} t\} \ .$$

The unfolding $Unf(G,s)$ of a graph G from a vertex s is the restriction of $Unf(G)$ to the vertices accessible from s. Given a set of graphs $\mathscr{H}$, $Unf(\mathscr{H})$ is the set of graphs obtained by unfolding from the graphs of $\mathscr{H}$. Inverse arcs are introduced to move up and down in trees: we have a set $\overline{L} := \{\overline{a} \mid a \in L\}$ of fresh symbols in bijection with L. By definition, we have an arc $(s, \overline{a}, t)$ iff (t, a, s) is an arc of G. Note that in a tree there is at most one inverse arc from a given node.

In the usual way $s \xrightarrow[G]{w}{}^{*} t$ means that there is a *path* from s to t labeled by the word w. A *substitution* is a relation $h \subseteq L \times (L \cup \overline{L})^*$. It has finite domain if $Dom(h) := \{a \mid h(a) \neq \emptyset\}$ is finite. In this case, the inverse mapping of any graph G by h is

$$h^{-1}(G) = \{s \xrightarrow{a} t \mid \exists w \in h(a) : s \xrightarrow[G]{w}{}^{*} t\} .$$

The mapping h is rational if $h(a)$ is rational for every $a \in L$. Given a set of graphs $\mathscr{H}$, $Rat^{-1}(\mathscr{H})$ is the set of graphs obtained by inverse rational mapping from the graphs of $\mathscr{H}$. Let $Tree_0$ be the set of finite trees. The *Caucal Hierarchy* is defined in the following way:

$$\begin{aligned} Graph_n &:= Rat^{-1}(Tree_n) , \\ Tree_{n+1} &:= Unf(Graph_n) . \end{aligned}$$

Here $Graph_0$ is the set of finite graphs, $Tree_1$ is the set of regular trees of finite degree and $Graph_1$ is the set of prefix-recognizable graphs [4]. The other levels are mostly unknown.

Theorem 1 ([5]) $\bigcup_{n\geqslant 0} Graph_n$ *is a family of graphs having a decidable monadic theory.*

As a corollary, μ-calculus model checking on these graphs is decidable, and one can determine the winner of a parity game. But this result of decidability in [5] rely on the results from [4,7,19] whereas for the restricted framework of games we give here a direct algorithmic construction for determining the winner and a winning strategy.

2.4 Graph Automaton

An alternating parity graph automaton, or *graph automaton* for short, as defined in [15] is a tuple $\mathscr{A} = (Q, W, \delta, q_0, \Omega)$ where Q is a finite set of states, W is a *finite* set of edge labels, δ is the transition function to be defined below, $q_0 \in Q$ is the initial state, $\Omega : Q \to [max]$ is a priority function defining the acceptance condition: the minimal priority appearing infinitely often should be even. Let $next(W) = \{\varepsilon\} \cup \bigcup_{a\in W}\{[a], \langle a\rangle\}$, and $\mathcal{B}^+(next(W) \times Q)$ be the set of positive Boolean formulas built from the atoms in $next(W) \times Q$. The transition function is of the form $\delta : Q \to \mathcal{B}^+(next(W) \times Q)$. In the case of graphs, we will consider $W \subseteq L$, whereas in the case of trees, we will allow $W \subseteq L \cup \overline{L}$.

A run of a graph automaton $\mathscr{A} = (Q, W, \delta, q_0, \Omega)$ over a graph $G \subseteq V \times L \times V$ from a vertex $s_0 \in V$ is a labeled tree $\langle T_r, r\rangle$ in which every node is labeled by an element of $V \times Q$. This tree is like the unfolding of the product of the automaton by the graph. A node in T_r, labeled by (s, q), describes a "copy" of the automaton that is in state q and is situated at the vertex s of G. Note that many nodes of T_r can correspond to the same vertex of G, because the automaton can come back to a previously visited vertex and because of alternation. The label of a node and its successors have to satisfy the transition function. Formally, a run $\langle T_r, r\rangle$ is a Σ_r-vertex labeled tree, where $\Sigma_r := V \times Q$ and $\langle T_r, r\rangle$ satisfies the following conditions:

- $r(t_0) = (s_0, q_0)$ where t_0 is the root of T_r.
- Consider $y \in T_r$ with $r(y) = (s, q)$ and $\delta(q) = \theta$. Then there is a (possibly empty) set $Y \subseteq next(W) \times Q$, such that Y satisfies θ, and for all $\langle d, q'\rangle \in Y$, the following hold:
  - If $d = \varepsilon$ then there exists a successor y' of y in T_r such that $r(y') = (s, q')$.

- If $d = \langle a \rangle$ then there exists a successor y' of y in T_r, and a vertex s' such that $s \xrightarrow[G]{a} s'$ and $r(y') = (s', q')$.
- If $d = [a]$ then for each vertex s' such that $s \xrightarrow[G]{a} s'$, there exists a successor y' of y in T_r such that $r(y') = (s', q')$.

The priority of a node $y \in T_r$, with $r(y) = (s, q)$, is $\Omega(q)$. A run is accepting if it satisfies the parity condition: along each infinite branch of T_r, the minimal priority appearing infinitely often is even.

When G is a tree, $\mathscr{A}$ is like an alternating two-way parity automaton of [15], because it can go up and down, but here the degree of the tree can be infinite. It is more general than the model of [12] which cannot distinguish between son and parent node. For the proofs we will also consider a tree automaton (defined as a graph automaton) that "reads" the labels of the vertices.

One can also consider the *model-checking game*: in a given graph Player 0 wants to prove that a formula is true and Player 1 has to challenge this. Similarly Player 0 wants to find an accepting run and Player 1 wants to refute it: a graph $G \subseteq V \times L \times V$ and a graph automaton $\mathscr{A}$ over G where $W = L_G$ define a parity game denoted by $(G, \mathscr{A})$. The configurations of the game are pairs $(s, q) \in V \times Q$, the initial configuration is (s_0, q_0). In general one needs also other configurations corresponding to subformulas of $\delta(q)$ to allow existential / universal choices: Player 0 makes the existential choices, Player 1 makes the universal ones. It is well known that a run is a strategy for Player 0, and an accepting run is a winning strategy for Player 0, see e.g. [11, ch. 4].

3 Game-Simulation between HPDS and Caucal Graphs

In this section we show an equivalence between a model based on graph transformations, where the vertex set is "abstract" —the Caucal graphs— and a model based on rewriting of "concrete" nodes —the HPDS. This game simulation should be compared to the notion of weak (bi)simulation. Moreover it seems that one can deduce from the following construction that each transition graph of a HPDS is a graph of the Caucal hierarchy of the same level.

3.1 From HPDS to Caucal Graphs

Theorem 2 *Given a game structure $\mathscr{G}$ on a HPDS H of level n, one can construct a graph automaton $\mathscr{A}$ and a tree $T \in Tree_n$ such that $\mathscr{G}$ is game-simulated by $(T, \mathscr{A})$.*

The tree $T \in Tree_n$ depends only on n and Γ.

Proof: (sketch) We describe the construction for $n = 1, 2$ and 3 before we give the generalization. Let $\mathscr{G} = (H, P_0, P_1, \Omega, s_0)$, $H = (P, \Gamma, \Delta)$ of level n.

Case $n = 1$. The idea here is similar to that of [15] and [4].
Let T_1 be the complete Γ-tree. It is the unfolding of a finite graph with a unique vertex and so $T_1 \in Tree_1$. See an example in Figure 1 where $\Gamma = \{a, b\}$. This tree is isomorphic to Γ^* in the sense that each node is associated to the label of the path from the root to it (we write the store from bottom to top, so we consider suffix rewriting in the application of the rules). It is easy to simulate a 1-store with this tree: each node corresponds to a word, which is a store content. Intuitively the effect of a transition $(p, a, p', push_1^b)$ on the store is simulated over T_1 by a path $\overline{a}ab$. Formally the state space of $\mathscr{A}$ is $Q = P \times (L \cup \overline{L})^{\leqslant 3}$, where a state (p, ε) on a node $v \in T_1$ represents a configuration $(p, [v])$ of the HPDS (by abuse v is associated to a word of Γ^*), whereas the states (p, x), $x \neq \varepsilon$ are intermediate states that simulate the behavior of the store. From these intermediate states, the transition is somehow "deterministic": $\forall a \in L \cup \overline{L},\ x \in (L \cup \overline{L})^{\leqslant 2}$:

$$\text{if } p \in P_0 \text{ then } \delta((p, ax)) = \langle \langle a \rangle, (p, x) \rangle \ ,$$
$$\text{if } p \in P_1 \text{ then } \delta((p, ax)) = \langle [a], (p, x) \rangle \ .$$

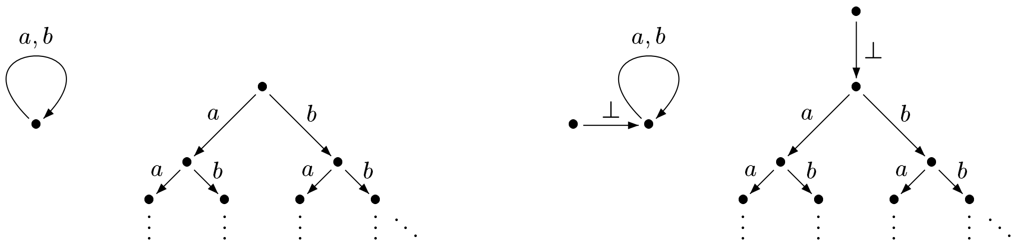

Figure 1. The complete $\{a,b\}$-tree T_1 obtained by unfolding of a finite graph, and the same with a "bottom stack symbol"

Note that here (on T_1), if $a \in L$ the "actions" $\langle a \rangle$ and $[a]$ are equivalent. From the states (p,ε) the corresponding player has to choose the move:

$$\text{if } p \in P_0 \text{ then } \delta((p,\varepsilon)) = \bigvee_{(p,a,p',pop_1)\in\Delta} \langle \varepsilon, (p',\overline{a})\rangle \quad \vee \bigvee_{(p,a,p',push_1^b)\in\Delta} \langle \varepsilon, (p',\overline{a}ab)\rangle \ ,$$
$$\text{if } p \in P_1 \text{ then } \delta((p,\varepsilon)) = \bigwedge_{(p,a,p',pop_1)\in\Delta} \langle \varepsilon, (p',\overline{a})\rangle \quad \wedge \bigwedge_{(p,a,p',push_1^b)\in\Delta} \langle \varepsilon, (p',\overline{a}ab)\rangle \ .$$

We see again that the convention is satisfied: when the play is in a deadlock, the player who should play loses immediately.

Case $n = 2$. For each letter $a \in \Gamma$, we assume that we have a fresh symbol $\dot{a}$ in L. We define the graph $G_1 \in \mathit{Graph}_1$ from the tree T_1:

$$G_1 = h_1^{-1}(T_1) \ ,$$

where the (finite) substitution h_1 is the following:

$$h_1(a) = a \text{ for all } a \in \Gamma \ , \qquad h_1(2) = \varepsilon \ ,$$
$$h_1(\dot{a}) = \overline{a} \text{ for all } a \in \Gamma \ .$$

Hence we suppose that $2 \in L$ is a fresh symbol. A part of the graph G_1 is pictured in Figure 2.

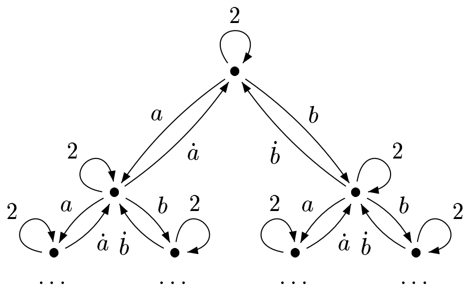

Figure 2. Graph G_1 for $\Gamma = \{a,b\}$

The loops labeled by 2 will be used to simulate the "copy" of the store content, *i.e.*, an operation $push_2$.

Then the tree $T_2 \in Tree_2$ is the unfolding of G_1 from the vertex that was the root of T_1. In Figure 3 extra node-labels are added. They represent the corresponding 2-store. Note that several nodes can represent the same store content. The operations on 2-stores are simulated by paths in T_2. More precisely, the effect of a transition is simulated in the following way if $\Gamma = \{a, b\}$:

$$\begin{aligned}
(p, a, p', push_1^b) &\text{ corresponds to } \dot{a}ab \ , \\
(p, a, p', pop_1) &\text{ corresponds to } \dot{a} \ , \\
(p, a, p', push_2) &\text{ corresponds to } \dot{a}a2 \ , \\
(p, b, p', pop_2) &\text{ corresponds to } \dot{b}\left(\overline{a} + \overline{b} + \overline{\dot{a}} + \overline{\dot{b}}\right)^* \overline{2} \ .
\end{aligned}$$

Of course the expression $\dot{b}\left(\overline{a} + \overline{b} + \overline{\dot{a}} + \overline{\dot{b}}\right)^* \overline{2}$ is regular, and one can move along such a path using three states of $\mathscr{A}$. Because we are on a tree, there is no infinite upward path. Following a 2-arc allows to copy the top 1-store because we stay exactly in the same position in G_1. For popping the top 1-store, one has to find the last 2-arc that was used, and follow it in the reverse direction. Note that just after a $push_2$ (a 2-arc), we cannot move along an inverse arc $\overline{a}$ (to simulate a pop_1), that's why the arcs $\dot{a}$ are necessary.

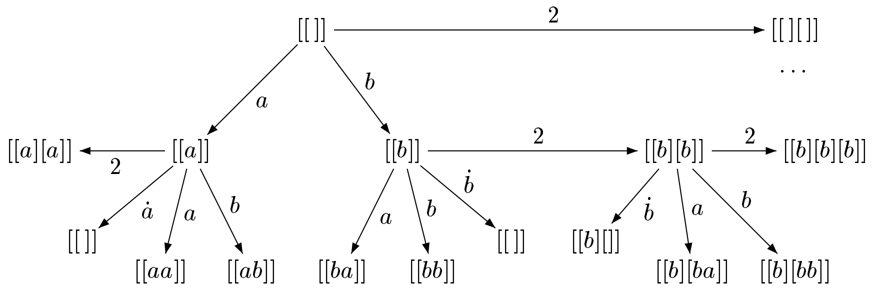

Figure 3. An initial part of the tree T_2

Case $n = 3$. We go on with $G_2 \in Graph_2$, defined from T_2:

$$G_2 = h_2^{-1}(T_2)$$

where the substitution h_2 is the following:

$$\begin{aligned}
&h_2(a) = a \text{ for all } a \in \Gamma \ , && h_2(2) = 2 \ , \\
&h_2(\dot{a}) = \dot{a} \text{ for all } a \in \Gamma \ , && h_2(\dot{2}) = \left\{\overline{a}, \overline{\dot{a}} \;\middle|\; a \in \Gamma\right\}^* \overline{2} \ , \\
&h_2(3) = \varepsilon \ .
\end{aligned}$$

Then $T_3 \in Tree_3$ is the unfolding of G_2 from the "root" (of T_2). On T_3 one can simulate a 3-store, almost the same way as a 2-store is simulated on T_2 (here $\Gamma = \{a, b\}$):

$$\begin{aligned}
(p, a, p', push_1^b) &\text{ corresponds to } \dot{a}ab \ , \\
(p, a, p', pop_1) &\text{ corresponds to } \dot{a} \ , \\
(p, a, p', push_2) &\text{ corresponds to } \dot{a}a2 \ , \\
(p, a, p', pop_2) &\text{ corresponds to } \dot{a}\dot{2} \ , \\
(p, a, p', push_3) &\text{ corresponds to } \dot{a}a3 \ , \\
(p, a, p', pop_3) &\text{ corresponds to } \dot{a}\left(\overline{2} + \overline{\dot{2}} + \overline{a} + \overline{b} + \overline{\dot{a}} + \overline{\dot{b}}\right)^* \overline{3} \ .
\end{aligned}$$

General case. It is easy to follow the construction: for $n \geqslant 3$, G_n is obtained from T_n using substitution h_n:

$$\begin{aligned}
h_n(a) &= a \text{ for all } a \in \Gamma \ , \qquad h_n(k) = k \text{ for all } 2 \leqslant k \leqslant n \ , \\
h_n(\dot{a}) &= \dot{a} \text{ for all } a \in \Gamma \ , \qquad h_n(\dot{k}) = \dot{k} \text{ for all } 2 \leqslant k < n \ , \\
h_n(\dot{n}) &= \left\{ \overline{a}, \overline{\dot{a}}, \overline{k}, \overline{\dot{k}} \ \middle| \ a \in \Gamma, 2 \leqslant k < n \right\}^* \overline{n} \ , \\
h_n(n+1) &= \varepsilon \ ,
\end{aligned}$$

and T_{n+1} is the unfolding of G_n from the "root". The automaton $\mathscr{A}$ has the same states as H plus auxiliary states for the regular expressions. It is clear that the winner of $\mathscr{G}$ is the winner of $(T, \mathscr{A})$, and a winning strategy in $(T, \mathscr{A})$ can be translated to a winning strategy in $\mathscr{G}$ (the other direction holds also here). ■

3.2 From Caucal Graphs to HPDS

Lemma 3 *Given a graph $G \in Graph_n$ and a graph automaton $\mathscr{A}$, one can construct a game structure $\mathscr{G}$ on a HPDS H of level n such that $(G, \mathscr{A})$ is game-simulated by $\mathscr{G}$.*

Proof: (sketch) The result is clear for $n = 0$, because G and $\mathscr{A}$ have a finite number of vertices and states.

Given $T_1 \in Tree_1$, $T_1 = Unf(G_0, s)$ for some $G_0 \in Graph_0$, we let $\Gamma = V_{G_0} \times L_{G_0}$. Letters from Γ will be pushed on a 1-store to remember the position in the unfolding, which is a path from s. Additionally the labels from L_{G_0} will allow to determine which inverse arc is possible from the current position. To simplify the notation we write $\langle a, q' \rangle \in \delta(q)$ if $\langle a, q' \rangle$ is an atom present in the formula $\delta(q)$. It is clear that the existential/universal choices in the formula can be expressed in the control states of a HPDS, so we skip this part and concentrate on the actual "moves":

$$\begin{aligned}
\langle a, q' \rangle \in \delta(q) &\text{ corresponds to } (q, (v, _), q', push_1^{(u,a)}) \text{ if } v \xrightarrow[G_0]{a} u \ , \\
\langle \overline{a}, q' \rangle \in \delta(q) &\text{ corresponds to } (q, (_, a), q', pop_1) \ .
\end{aligned}$$

A graph in $Graph_1$ can be simulated the same way using intermediate states for the rational substitutions.

Let $T_2 \in Tree_2$, $T_2 = Unf(G_1, s)$, T_2 can be simulated by a 2-store: each transition of G_1 is simulated on the top 1-store just like above, but the top 1-store has to be "copied" by a $push_2$ operation to keep track of the unfolding. It is also necessary to remember at each move the label of the arc of G_1 that was used. A solution is to use the following stack alphabet:

$$\Gamma = (V_{G_0} \times L_{G_0}) \uplus (\{2\} \times L_{G_1}) \ .$$

An action $\langle a, q' \rangle \in \delta(q)$ is simulated by the following sequence of operations:

$$\begin{array}{l} push_2 \\ < \text{simulation of an } a\text{-arc of } G_1 \text{ on the top 1-store} > \\ push_1^{(2,a)} \ . \end{array}$$

And an action $\langle \overline{a}, q' \rangle \in \delta(q)$ in the following way:

$$\begin{array}{l} < \text{check that the top symbol is } (2,a) > \\ pop_2 \ . \end{array}$$

And so on for $n \geqslant 3$. This construction is more natural if we use the model of higher order pushdown automata from [9], but both models are clearly equivalent [13]. ∎

4 Reducing the hierarchy level

In this section we present our main result: an algorithmic solution of parity games on the graphs of the Caucal hierarchy, and hence on HPDS. The proof is by induction on the definition of the hierarchy, using the next two lemmas to obtain graphs of lower levels.

Lemma 4 *Given $G \in Graph_n$ and a graph automaton $\mathscr{A}$, the game $(G, \mathscr{A})$ can be effectively simulated by a game $(T, \mathscr{B})$, where $T \in Tree_n$, such that $G = h^{-1}(T)$, and $\mathscr{B}$ is a graph automaton.*

The proof uses similar techniques as in [2] or [15] for the case of prefix-recognizable graphs.

Proof: By definition $G = h^{-1}(T)$. The aim is to "simulate" an a-transition of $\mathscr{A}$ along an arc of G by a path in T: a sequence of transitions of $\mathscr{B}$ labeled by a word of $h(a)$. The automaton $\mathscr{B} = (Q_{\mathscr{B}}, W_{\mathscr{B}}, \delta_{\mathscr{B}}, q_0, \Omega_{\mathscr{B}})$ will have the same states as $\mathscr{A}$ plus auxiliary states for this simulation. For each $a \in L$, $h(a)$ is regular. If $h(a) \neq \emptyset$, let

$$\mathscr{C}_a = (Q_a, W_a, \Delta_a, q_{0a}, F_a)$$

be a (non-deterministic) finite automaton on finite words recognizing $h(a)$. Here F_a is the set of final states. We consider $\mathscr{C}_a$ as a finite graph, and note the transitions $q_a \xrightarrow[\mathscr{C}_a]{b} q'_a$ for $q_a, q'_a \in Q_a$. The new auxiliary states of $\mathscr{B}$ are of the form $(q_a, [q])$ and $(q_a, \langle q \rangle)$ for $q \in Q_{\mathscr{A}}$, $q_a \in Q_a$. To obtain the transitions of $\mathscr{B}$ from the transitions of $\mathscr{A}$,

$$\begin{array}{c} \text{each atom } \langle [a], q \rangle \text{ is replaced by } \langle \varepsilon, (q_{0a}, [q]) \rangle \ , \\ \text{and each } \langle \langle a \rangle, q \rangle \text{ is replaced by } \langle \varepsilon, (q_{0a}, \langle q \rangle) \rangle \end{array}$$

in the body of a transition $\delta_{\mathscr{A}}(q')$. Of course the atoms $\langle \varepsilon, q \rangle$ remain unchanged. Then the new transitions of $\mathscr{B}$ are

$$\delta_{\mathscr{B}}((q_a, [q])) = \bigwedge_{q_a \xrightarrow[\mathscr{C}_a]{b} q'_a} \langle [b], (q'_a, [q]) \rangle \wedge \bigwedge_{q_a \in F_a} \langle \varepsilon, q \rangle \ ,$$

$$\delta_{\mathscr{B}}((q_a, \langle q \rangle)) = \bigvee_{q_a \xrightarrow[\mathscr{C}_a]{b} q'_a} \langle \langle b \rangle, (q'_a, \langle q \rangle) \rangle \vee \bigvee_{q_a \in F_a} \langle \varepsilon, q \rangle \ ,$$

for each $a \in L$ such that $h(a) \neq \emptyset$.

To avoid the game to stay forever in the intermediate nodes of $\mathscr{B}$, we assign to these nodes a priority that is losing for the corresponding player. Suppose that the priority function $\Omega_{\mathscr{A}}$ of $\mathscr{A}$ ranges from 0 to $2c$, $c \geqslant 0$, then we fix

$$\Omega_{\mathscr{B}}((q_a, [q])) = 2c \ , \quad \Omega_{\mathscr{B}}((q_a, \langle q \rangle)) = 2c + 1 \ .$$

And dualy if the maximal priority of $\mathscr{A}$ is $2c+1$, then the new priorities are $2c+1$ and $2c+2$. They do not interfere with the "real" game $(G, \mathscr{A})$. So one has one new priority and in the worst case the number of states of $\mathscr{B}$ is $|Q_{\mathscr{B}}| = |Q_{\mathscr{A}}| \left(1 + \sum_{h(a) \neq \emptyset} |Q_a|\right)$. ■

Lemma 5 *Given $T \in Tree_{n+1}$ and a graph automaton $\mathscr{A}$, the game $(T, \mathscr{A})$ can be effectively simulated by a game $(G, \mathscr{B})$, where $G \in Graph_n$, such that $T = Unf(G, s)$, and $\mathscr{B}$ is a graph automaton.*

This result is related to the $k-$*covering* of [7], where k is the number of states of $\mathscr{A}$. The proof is based here on the construction of a *one-way* tree automaton that is equivalent to $\mathscr{A}$. This construction was presented in [17] only in the case of (deterministic) trees of finite degree. The idea is that if Player 0 has a winning strategy in $(T, \mathscr{A})$, then he has also a positional winning strategy [8]: choosing always the same transition from the same vertex. This strategy can be encoded as a labeling of T using a (big) finite alphabet. Then several conditions have to be checked to verify that this strategy is winning, but it can be done by a one-way automaton. Finely this strategy can be non-deterministicaly guessed by the automaton. And a one-way automaton cannot distinguish T and G.

We give here a flavor of this proof, details are in Appendix. Formally a *strategy for $\mathscr{A}$ and a given tree T* is a mapping

$$\tau : V_T \longrightarrow \mathscr{P}(Q \times next(W) \times Q) \ .$$

An element $(q, d, q') \in \tau(x)$ means that when arriving at node $x \in V_T$ in state q, the automaton should send a copy in state q' to the node in direction d (and maybe other copies in other directions). A strategy must satisfy the transition of $\mathscr{A}$, and a strategy has to be followed:

$\forall x \in V_T, \forall (q, d, q') \in \tau(x)$:
$\{(d_2, q_2) \mid (q, d_2, q_2) \in \tau(x)\}$ satisfies $\delta(q)$ and:
- if $d = \varepsilon$ then $\exists d_1, q_1, (q', d_1, q_1) \in \tau(x)$ or $\emptyset$ satisfies $\delta(q')$,
- if $d = [a]$ then $\forall y : x \xrightarrow[T]{a} y \Rightarrow \exists d_1, q_1, (q', d_1, q_1) \in \tau(y)$ or $\emptyset$ satisfies $\delta(q')$,
- if $d = \langle a \rangle$ then $\exists y : x \xrightarrow[T]{a} y \wedge \exists d_1, q_1, (q', d_1, q_1) \in \tau(y)$ or $\emptyset$ satisfies $\delta(q')$.

For the root $t_0 \in V_T$ we have:

$$\exists d_1, q_1, (q_0, d_1, q_1) \in \tau(t_0) \text{ or } \emptyset \text{ satisfies } \delta(q_0) \ . \tag{1}$$

Considering $St := \mathscr{P}(Q \times next(W) \times Q)$ as an alphabet, a St-labeled tree (T, τ) defines a positional strategy on the tree T. One can construct a *one-way* automaton that checks that this strategy is correct according to the previous requirements.

The second step of the reduction from two-way to one-way is concerned with the priorities seen along (a branch of) the run, when one follows a strategy τ. To check the acceptance condition, it is necessary to follow each path of $\mathscr{A}$ in T up and down, and remember the priorities appearing. Such a path can be decomposed into a *downwards* path and several finite *detours* from the path, that come back to their origin (in a loop). Because each node has a unique parent and $\mathscr{A}$ starts at the root, we consider only downwards detour (each move $\overline{a}$ is in a detour). That is to say, if a node is visited more than once by a run, we know that the first time it was visited, the run came from above. This idea of detour is close to the idea of subgame in [18]. To keep track of these finite detours, we use the following annotation. An *annotation* for $\mathscr{A}$, a given tree T and a strategy τ is a mapping

$$\eta : V_T \longrightarrow \mathscr{P}(Q \times [max] \times Q) \ .$$

Intuitively $(q, f, q') \in \eta(x)$ means that from node x and state q there is a detour that comes back to x with state q' and the smallest priority seen along this detour is f. Again η can be considered as a labeling of T, and a $one-way$ automaton can check that the annotation is consistent with respect to the strategy in reading both labelings. A typical requirement is:

$$\begin{aligned}(q, [a], q_1) \in \tau(x) \ \Rightarrow \ \forall y \in V_T : x \xrightarrow{a} y \Rightarrow \\ \big[(q_1, \overline{a}, q') \in \tau(y) \Rightarrow (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta(x)\big] \ .\end{aligned}$$

The last step is to check every possible branch of the run by using the detours: it is easy to define a one-way automaton $\mathscr{E}$ that "simulates" (follow) a branch of the run of $\mathscr{A}$. One can change the acceptance condition of $\mathscr{E}$ such that it accepts a tree labeled by τ and η iff *there exists* a branch in the corresponding run of $\mathscr{A}$ that *violates* the acceptance condition of $\mathscr{A}$. Then using techniques from [16] one can determinize and complement $\mathscr{E}$. Finally the product of the previous automata has to be build, to check all conditions together, and a "projection" is necessary to nondeterministicaly guess the labels, *i.e.*, the strategy and the annotation.

Theorem 6 *Parity games on higher order pushdown systems are solvable: one can determine the winner and compute a winning strategy.*

As a corollary we get a new proof that the μ-calculus model checking of these graphs is decidable (it was known as a consequence of the MSO-decidability).

Proof: Given a game structure $\mathscr{G}$ on a HPDS H of level n, one obtains from Theorem 2 a graph automaton $\mathscr{A}$ and a tree $T \in Tree_n$ such that $(T, \mathscr{A})$ is a game simulation of $\mathscr{G}$. By successive reductions using Lemmas 4 and 5, one can obtain a game on a finite graph which is equivalent to the initial game. Using classical techniques (see [11, ch. 7]), one can solve this game, and compute a positional strategy for the winner. Then one can step by step reconstruct the strategy for the graphs of higher levels. ■

5 Complexity, Strategy

The one-step reduction of Lemma 5 is in exponential time in the description of $T \in Tree_{n+1}$ and $\mathscr{A}$, and the size of the output is also exponential. For this reason the complexity of the complete solution of a parity game on a Caucal graph or on a HPDS is a tower of exponentials where the height is the level of the graph. The classical translation from parity game *to* μ-calculus *to* MSO and the corresponding decision procedure is already non-elementary (in the number of priorities) for level 1 graphs. And following [19] the (one-step) transformation of an MSO-formula from the unfolding to the original graph is also non-elementary.

Using the reductions presented here, one can compute a winning strategy for a 1-HPDS game which is a finite automaton that reads the current configuration and outputs the "next move", like in [15,3]. But it is more natural to consider a *pushdown strategy* as introduced in [18]. It is a pushdown transducer that reads the moves of Player 1 and outputs the moves of Player 0. It needs additional memory (the stack), but the computation of the "next move" can be done in constant time. When we recompose the game, a strategy for an n-HPDS game is an n-HPDS with input and output which possibly needs to execute several transitions to compute the "next move" from a given configuration.

Acknowledgment

Great thanks to Didier Caucal, Christof Löding, Wolfgang Thomas, Stefan Wöhrle and to the referees of ICALP'03 for useful remarks.

References

1. T. Cachat, *Symbolic strategy synthesis for games on pushdown graphs*, ICALP'02, LNCS 2380, pp. 704-715, 2002.
2. T. Cachat, *Uniform solution of parity games on prefix-recognizable graphs*, INFINITY 2002, ENTCS 68(6), 2002.
3. T. Cachat, *Two-way tree automata solving pushdown games*, ch. 17 in [11].
4. D. Caucal, *On infinite transition graphs having a decidable monadic theory*, ICALP'96, LNCS 1099, pp. 194–205, 1996.
5. D. Caucal, *On infinite terms having a decidable monadic theory*, MFCS'02, LNCS 2420, pp. 165–176, 2002.
6. D. Caucal, O. Burkart, F. Moller and B. Steffen, *Verification on infinite structures*, Handbook of process algebra, Ch. 9, pp. 545-623, Elsevier, 2001.
7. B. Courcelle and I. Walukiewicz, *Monadic second-order logic, graph converings and unfoldings of transition systems*, Annals of Pure and Applied Logic 92-1, pp. 35-62, 1998.
8. E. A. Emerson and C. S. Jutla, *Tree automata, mu-calculus and determinacy*, FoCS'91, IEEE Computer Society Press, pp. 368–377, 1991.
9. J. Engelfriet, *Iterated push-down automata*, 15th STOC, pp. 365–373, 1983.
10. J. Esparza, D. Hansel, P. Rossmanith and S. Schwoon, *Efficient algorithm for model checking pushdown systems*, Technische Universität München, 2000.
11. E. Grädel, W. Thomas and T. Wilke eds., *Automata, Logics, and Infinite Games*, A Guide to Current Research, LNCS 2500, 2002.
12. E. Grädel and I. Walukiewicz, *Guarded fixed point logic*, LICS '99, IEEE Computer Society Press, pp. 45–54, 1999.
13. T. Knapik, D. Niwinski and P. Urzyczyn *Higher-order pushdown trees are easy*, FoSSaCS'02, LNCS 2303, pp. 205–222, 2002.
14. O. Kupferman, M. Y. Vardi and N. Piterman, *Model checking linear properties of prefix-recognizable systems*, CAV'02, LNCS 2404, pp. 371–385.
15. O. Kupferman and M. Y. Vardi, *An automata-theoretic approach to reasoning about infinite-state systems*, CAV'00, LNCS 1855, 2000.
16. W. Thomas, Languages, Automata, and Logic, *Handbook of formal language theory*, vol. III, pp. 389–455, Springer-Verlag, 1997.
17. M. Y. Vardi, *Reasoning about the past with two-way automata.*, ICALP'98, LNCS 1443, pp. 628–641, 1998.
18. I. Walukiewicz, *Pushdown processes: games and model checking*, CAV'96, LNCS 1102, pp. 62–74, 1996. Full version in Information and Computation 164, pp. 234-263, 2001.
19. I. Walukiewicz, *Monadic second order logic on tree-like structures*, STACS'96, LNCS 1046, pp. 401–414. Full version in TCS 275 (2002), no. 1–2, pp. 311–346.

Appendix

Proof: (Lemma 5) We adapt the construction from [17] to the case of infinite degree. Recall that $W \subseteq L \cup \overline{L}$. As remarked above, the question whether Player 0 wins the game $(T, \mathscr{A})$ is equivalent to the question whether there is an accepting run of $\mathscr{A}$ on T. We want to check this condition with a *one-way* tree automaton.

We know from [8] that if Player 0 has a winning strategy, then he has a memoryless winning strategy. In other words, if $\mathscr{A}$ has an accepting run, then it has an accepting run using a memoryless strategy: choosing always the same "transitions" from the same node and state. We decompose the run of $\mathscr{A}$ using this strategy.

Definition: A strategy for $\mathscr{A}$ and a given tree T is a mapping

$$\tau : V_T \longrightarrow \mathscr{P}(Q \times next(W) \times Q) \ .$$

A strategy must satisfy the transitions of $\mathscr{A}$, and a strategy has to be followed:

$$\forall x \in V_T, \forall (q, d, q') \in \tau(x) :$$

$$\{(d_2, q_2) \mid (q, d_2, q_2) \in \tau(x)\} \text{ satisfies } \delta(q) \text{ and:} \tag{2}$$

$$\text{- if } d = \varepsilon \text{ then } \exists d_1, q_1, (q', d_1, q_1) \in \tau(x) \text{ or } \emptyset \text{ satisfies } \delta(q') \ , \tag{3}$$

$$\text{- if } d = [a] \text{ then } \forall y : x \xrightarrow[T]{a} y \Rightarrow \exists d_1, q_1, (q', d_1, q_1) \in \tau(y) \text{ or } \emptyset \text{ satisfies } \delta(q') \ , \tag{4}$$

$$\text{- if } d = \langle a \rangle \text{ then } \exists y : x \xrightarrow[T]{a} y \wedge \exists d_1, q_1, (q', d_1, q_1) \in \tau(y) \text{ or } \emptyset \text{ satisfies } \delta(q') \ . \tag{5}$$

For the root $t_0 \in V_T$ we have:

$$\exists d_1, q_1, (q_0, d_1, q_1) \in \tau(t_0) \text{ or } \emptyset \text{ satisfies } \delta(q_0) \ . \tag{6}$$

Considering $St := \mathscr{P}(Q \times next(W) \times Q)$ as an alphabet, a St-labeled tree (T, τ) defines a memoryless strategy on the tree T. We will construct a *one-way* automaton $\mathscr{C}$ that checks that this strategy is correct according to the previous requirements. For $(q, \langle a \rangle, q') \in \tau(x)$, if $a \in L$ it has just to check in the direction a downwards that the strategy is well defined for q', but if $(q, \langle \overline{a} \rangle, q') \in \tau(x)$, $\overline{a} \in \overline{L}$ he must have *remembered* that the strategy *was* defined for q' in the parent-node, and that the arc from the parent node is labeled by a. The states of $\mathscr{C}$ are tuples $\langle Q_1, Q_2, \ell \rangle \in \mathscr{P}(Q) \times \mathscr{P}(Q) \times (W \cap L)$, where ℓ is the label of the arc from the parent node, $q' \in Q_1$ means that $\mathscr{C}$ has to check (down) that the strategy can be followed for q', and $q'' \in Q_2$ means that q'' is already allowed at the parent node. Note that if a is the label of the arc from the parent node, then the actions $[\overline{a}]$ and $\langle \overline{a} \rangle$ are equivalent, because in a tree there is no more than one parent node. So we shall write simply $\overline{a}$ for both cases when we know from the context that it is right. If a is *not* the label from the parent node, then $[\overline{a}]$ means immediate "wins", and $\langle \overline{a} \rangle$ means immediate "lost" (for Player 0). Let

$$\mathscr{C} := (\mathscr{P}(Q) \times \mathscr{P}(Q) \times \ell, St, \delta_{\mathscr{C}}, \langle \{q_0\}, \emptyset, b \rangle, true) \text{ where} \tag{7}$$

$$\delta_{\mathscr{C}}(\langle Q_1, Q_2, \ell \rangle, \tau_1) :=$$

$$\text{IF } \forall q \in Q_1,\ \{(d_2, q_2) : (q, d_2, q_2) \in \tau_1\} \text{ satisfies } \delta(q), \text{ and} \tag{8}$$

$$\forall (q', \varepsilon, q) \in \tau_1,\ \{(d_2, q_2) : (q, d_2, q_2) \in \tau_1\} \text{ satisfies } \delta(q), \text{ and} \tag{9}$$

$$\forall (q, [\overline{a}], q') \in \tau_1 : a = \ell \Rightarrow q' \in Q_2, \text{ and} \tag{10}$$

$$\forall (q, \langle \overline{a} \rangle, q') \in \tau_1 : a = \ell \wedge q' \in Q_2 \tag{11}$$

$$\text{THEN} \bigwedge_{d \in next(W \cap L) \setminus \{\varepsilon\}} \big(d, \langle \{q' : \exists\ (q, d, q') \in \tau_1\}, Q_2' \rangle\big) \tag{12}$$

$$\text{with } Q_2' := \{q'' :\ \exists\ d_1, q_1,\ (q'', d_1, q_1) \in \tau_1 \text{ or } \emptyset \text{ satisfies } \delta(q'')\}, \tag{13}$$

$$\text{ELSE } false. \tag{14}$$

Condition (9) is related to (3), (10) and (11) to (4) and (5) where $a \in \overline{L}$, and (12) to (4) and (5) where $a \in L$. In condition (8) there is no requirement on the $q \notin Q_1$, that's why the condition (2) above is stronger. This is not a problem for the following, as we are searching *some* winning strategy (one could define the *minimal* valid strategy as in [17]).

The acceptance condition is easy to enunciate: it just requires that the run can be followed, *i.e.*, the transition is possible at each node. In the initial state $\langle \{q_0\}, \emptyset, b\rangle$, the letter b is not relevant. $\mathscr{C}$ is a one-way automaton with $4^{|Q|}$ states.

Proposition: A two-way alternating parity automaton accepts an input tree iff it has an accepting strategy over the input tree.

With the help of a so called annotation, we will check in the following whether a strategy is accepting.

Annotation

The previous automaton $\mathscr{C}$ just checks that the strategy can be followed (ad infinitum) but forgets about the priorities of $\mathscr{A}$. To check the acceptance condition, it is necessary to follow each path of $\mathscr{A}$ up and down, and remember the priorities appearing. Such a path can be decomposed into a *downwards* path and several finite *detours* from the path, that come back to their origin (in a loop). Because each node has a unique parent and $\mathscr{A}$ starts at the root, we consider only downwards detour (each move $\overline{a}$ is in a detour). That is to say, if a node is visited more than once by a run, we know that the first time it was visited, the run came from above. This idea of detour is close to the idea of subgame in [18]. To keep track of these finite detours, we use the following annotation.

Definition: An annotation for $\mathscr{A}$ and a given tree T is a mapping

$$\eta : V_T \longrightarrow \mathscr{P}(Q \times [max] \times Q) \ .$$

Intuitively $(q, f, q') \in \eta(x)$ means that from node x and state q there is a detour that comes back to x with state q' and the smallest priority seen along this detour is f. By definition, the following conditions are required for the annotation η of a given strategy τ, indeed a detour can stay in the node x (15,16), go down to a child y and come back immediately to x (17,19), or go down to y, use another detour from y and then come back to x (18,20).

$$\forall\, q, q' \in Q,\ x \in V_T,\ a \in W \cap L,\ f, f' \in [max]:$$

$$(q, \varepsilon, q') \in \tau(x) \Rightarrow (q, \Omega(q'), q') \in \eta(x)\ , \tag{15}$$

$$(q_1, f, q_2) \in \eta(x) \wedge (q_2, f', q_3) \in \eta(x) \Rightarrow (q_1, \min(f, f'), q_3) \in \eta(x)\ , \tag{16}$$

$$(q, [a], q_1) \in \tau(x) \ \Rightarrow\ \forall y \in V_T : x \xrightarrow{a} y \Rightarrow \big[(q_1, \overline{a}, q') \in \tau(y) \Rightarrow (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta(x)\big]\ , \tag{17}$$

$$(q, [a], q_1) \in \tau(x) \ \Rightarrow\ \forall y \in V_T : x \xrightarrow{a} y \Rightarrow \big[(q_1, f, q_2) \in \eta(y) \wedge (q_2, \overline{a}, q') \in \tau(y) \Rightarrow (q, \min(\Omega(q_1), f, \Omega(q')), q') \in \eta(x)\big]\ . \tag{18}$$

The next conditions have to be "synchronized" with the automaton $\mathscr{C}$ that checks the strategy. The a-successor y of x is "chosen" by $\mathscr{C}$:

$$(q, \langle a\rangle, q_1) \in \tau(x) \ \Rightarrow\ \exists y \in V_T : x \xrightarrow{a} y \wedge \big[(q_1, \overline{a}, q') \in \tau(y) \Rightarrow (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta(x)\big], \tag{19}$$

$$(q, \langle a\rangle, q_1) \in \tau(x) \ \Rightarrow\ \exists y \in V_T : x \xrightarrow{a} y \wedge \big[(q_1, f, q_2) \in \eta(y) \wedge (q_2, \overline{a}, q') \in \tau(y) \Rightarrow (q, \min(\Omega(q_1), f, \Omega(q')), q') \in \eta(x)\big]. \tag{20}$$

Considering $An := \mathscr{P}(Q \times [max] \times Q)$ as an alphabet, the aim is now to construct a *one-way* automaton $\mathscr{D}$ on $(An \times St)$-labeled trees that checks that the annotation satisfies these requirements. Conditions 15 and 16 above can be checked in each node (independently) without memory. For conditions 17 to 20, the automaton has to remember the whole $\eta(x)$ from the parent node x, and the part of $\tau(x)$ leading to the current node. Let

$$\mathscr{D} := (An \times \mathscr{P}(Q \times Q), An \times St, \delta_{\mathscr{D}}, \langle \emptyset, \emptyset \rangle, true),$$

where

$$\begin{aligned}
&\delta_{\mathscr{D}}(\langle \eta_0, \alpha \rangle, \langle \eta_1, \tau_1 \rangle) := \\
&\text{IF conditions 15 and 16 hold for } \eta_1 \text{ and } \tau_1 \text{ AND} \\
&\forall (q, q_1) \in \alpha : (q_1, \overline{a}, q') \in \tau_1 \;\Rightarrow\; (q, \min(\Omega(q_1), \Omega(q')), q') \in \eta_0 \qquad (21) \\
&\forall (q, q_1) \in \alpha : (q_1, f, q_2) \in \eta_1 \wedge (q_2, \overline{a}, q') \in \tau_1 \\
&\qquad \Rightarrow\; (q, \min(\Omega(q_1), f, \Omega(q')), q') \in \eta_0 \qquad (22) \\
&\text{THEN} \bigwedge_{d \in next(W \cap L) \setminus \{\varepsilon\}} \big(d, \langle \eta_1, \{(q, q_1) : \; exists \; (q, d, q_1) \in \tau_1\} \rangle\big) \\
&\text{ELSE } false \, .
\end{aligned}$$

Condition (21) is related to (17,19), and (22) to (18,20). We recall that in the case that $(q, \langle a \rangle, q_1) \in \tau_1$, the $\langle a \rangle$-transition has to be synchronized with the one of the automaton $\mathscr{C}$: the product automaton that we will define later has to chose an a-successor and follow the run of $\mathscr{C}$ and $\mathscr{D}$ from this successor.

Similarly to $\mathscr{C}$, $\mathscr{D}$ is a one-way automaton with $2^{|Q|^2 m} \cdot 2^{|Q|^2} = 2^{|Q|^2(m+1)}$ states, and the acceptance condition is again trivial. Note that if a part of the tree is not visited by the original automaton $\mathscr{A}$, the strategy and annotation can be empty on this part. The automaton $\mathscr{D}$ does not check that the annotation is minimal, but this is not a problem. With the help of the annotation one can determine if a path of $\mathscr{A}$ respects the acceptance condition or not, as showed in the next part of the proof.

Parity Acceptance

Up to now the automata $\mathscr{C}$ and $\mathscr{D}$ together just check that the strategy and annotation for the run of $\mathscr{A}$ are correct, but do not verify that the run of $\mathscr{A}$ is accepting, *i.e.*, that each path is valid. With the help of the annotation we can "simulate" (follow) a path of $\mathscr{A}$ with a one-way automaton, and determine the parity condition for this path. This one-way automaton does not go into the detours, but reads the smallest priority appearing in them. Let

$$\begin{aligned}
&\mathscr{E} := (Q \times [max], An \times St, \delta_{\mathscr{E}}, \langle q_0, 0 \rangle, \Omega_0), \\
&\delta_{\mathscr{E}}(\langle q, i \rangle, \langle \eta_1, \tau_1 \rangle) := \bigvee_{(q,d,q') \in \tau_1, d = [a] \vee d = \langle a \rangle} (\langle a \rangle, \langle q', \Omega(q') \rangle) \;\vee \bigvee_{(q,f,q') \in \eta_1} (\varepsilon, \langle q', f \rangle).
\end{aligned}$$

Once again if $d = \langle a \rangle$, the automaton has to be synchronized with $\mathscr{C}$ and $\mathscr{D}$ to choose the right a-successor. At each step $\mathscr{E}$ either goes *down* following the strategy, or simulates a detour with an ε-move and the corresponding priority. The second component ($[max]$) of the states of $\mathscr{E}$ just remembers the last priority seen. We can transform $\mathscr{E}$ into a nondeterministic one-way automaton $\mathscr{E}'$ without ε-moves with the same state space. Note that $\mathscr{E}$ can possibly stay forever in the same node by using ε-transitions, either in an accepting run or not. This possibility can be checked by $\mathscr{E}'$ just by reading the current annotation, with a transition *true* or *false*.

We will use $\mathscr{E}$ and $\mathscr{E}'$ to find the *invalid* paths of the run of $\mathscr{A}$, just by changing the acceptance condition: $\Omega_0(\langle q, i \rangle) := i + 1$.

Proposition: The one-way tree automaton $\mathscr{E}'$ accepts a $(An \times St)$-labeled tree iff the corresponding run of $\mathscr{A}$ is *not* accepting.

But $\mathscr{E}'$ is not deterministic, and accepts a tree if $\mathscr{E}'$ has *some* accepting run. We can view $\mathscr{E}'$ as a word automaton: it follows just a branch of the tree. For this reason it is possible to co-determinize it: determinize and complement it in a singly exponential construction (see [16]) to construct the automaton $\overline{\mathscr{E}}$ that accepts those of the $(An \times St)$-labeled trees that represent the accepting runs of $\mathscr{A}$.

We will define the product $\mathscr{B}' := \mathscr{C} \times \mathscr{D} \times \overline{\mathscr{E}}$ of the previous automata, that accepts a $(An \times St)$-labeled input tree iff the corresponding run of $\mathscr{A}$ is accepting. Let

$$\begin{aligned} \mathscr{B}' &:= (Q_{\mathscr{C}} \times Q_{\mathscr{D}} \times Q_{\overline{\mathscr{E}}}, An \times St, \delta_{\mathscr{B}'}, q_{0,\mathscr{B}'}, Acc), \\ \delta_{\mathscr{B}'}(\langle q_{\mathscr{C}}, q_{\mathscr{D}}, q_{\overline{\mathscr{E}}} \rangle, \langle \eta_1, \tau_1 \rangle) &:= \langle \delta_{\mathscr{C}}(q_{\mathscr{C}}, \langle \tau_1 \rangle), \delta_{\mathscr{D}}(q_{\mathscr{D}}, \langle \eta_1, \tau_1 \rangle), \delta_{\overline{\mathscr{E}}}(q_{\overline{\mathscr{E}}}, \langle \eta_1, \tau_1 \rangle) \rangle , \end{aligned}$$

where $Q_{\mathscr{C}}$ is the state space of $\mathscr{C}$, and so on. The acceptance condition Acc of $\mathscr{B}'$ is then exactly the one of $\overline{\mathscr{E}}$: $\Omega_{\mathscr{B}'}(\langle q_{\mathscr{C}}, q_{\mathscr{D}}, q_{\overline{\mathscr{E}}} \rangle) = \Omega_{\overline{\mathscr{E}}}(q_{\overline{\mathscr{E}}})$.

We define the automaton $\mathscr{B}$ to be the "projection" of $\mathscr{B}'$: $\mathscr{B}$ nondeterministicaly guesses the labels from $An \times St$, $\mathscr{B}$ has no input alphabet. Finally $\mathscr{B}$ is a one-way tree-automaton that is equivalent to $\mathscr{A}$: it accepts the same trees. The strategy and annotation depended on the input tree, now after the projection, $\mathscr{B}$ can search the run of $\mathscr{A}$ for each input tree. The automaton $\mathscr{B}$ has (like $\mathscr{B}'$) $4^{|Q|} \cdot 2^{|Q|^2(m+1)} \cdot 2^{c|Q|m} = 2^{|Q|^2(m+1)} \cdot 2^{|Q|(2+cm)}$ states. As a one-way automaton, $\mathscr{B}$ cannot distinguish T and G. In other words, it has an accepting run on T iff it has one on G. Positional strategy on G gives a "regular" run on T. ■