The Complexity of Games on Higher Order Pushdown Automata

Thierry Cachat, Igor Walukiewicz

The Complexity of Games on Higher Order Pushdown Automata ⋆

Thierry Cachat[1] and Igor Walukiewicz[2]

[1] LIAFA/CNRS UMR 7089 & Université Paris 7, France
Firstname.Lastname@liafa.jussieu.fr
[2] LaBRI, Université Bordeaux-1, France igw@labri.fr

Abstract We prove an n-EXPTIME lower bound for the problem of deciding the winner in a reachability game on Higher Order Pushdown Automata (HPDA) of level n. This bound matches the known upper bound for parity games on HPDA. As a consequence the μ-calculus model checking over graphs given by n-HPDA is n-EXPTIME complete.

1 Introduction

Higher Order Pushdown Automaton (HPDA) is a classical model of computation [6,7] that has recently regained attention. In [9] it has been proved that the MSO theory of the computation trees of HPDA is decidable. Then in [5] a new family of infinite graphs, also with a decidable MSO theory, has been introduced, which is closely related to HPDA (see [2,4]). See also other approaches in [1,3]. Up to now the Caucal hierarchy of [5] is essentially the largest class of graphs with a decidable MSO theory. But these decidability results have non-elementary complexity, even for a fixed level of the hierarchy. Considering μ-calculus model-checking and parity games allows to have better complexity bounds.

We consider the question of deciding a winner in a reachability game given by a HPDA. It was shown by the first author [2] that parity games on n-HPDA's can be solved in n-EXPTIME. This also gives n-EXPTIME algorithm for the μ-calculus model checking over such graphs. Here we complement the picture by showing that even reachability games are n-EXPTIME hard on n-HPDA's, thereby showing n-EXPTIME completeness for game solving and μ-calculus model checking over n-HPDA's.

It was already shown by the second author in [10] that pushdown games (on 1-HPDA) are EXPTIME-complete. We extend the technique with codding big counters, following the notation from [11], where the computation of space bounded Turing machines are written with the help of 1-counters of n-bits, 2-counters of 2^n bits and so on. The expressive power of HPDA is used to "copy" parts of the store and check equality of big counters.

⋆ This research has been partially supported by the European Community Research Training Network "Games and Automata for Synthesis and Validation" (GAMES), (contract HPRN-CT-2002-00283), see www.games.rwth-aachen.de.

In the next section we present the definitions of game and HPDA. In Section 3 we prove the lower bound using a reduction from the word problem for alternating HPDA and a result by Engelfriet. The rest of the paper is devoted to an alternative, self contained and hopefully simple, proof of the lower bound. Using HPDA we show in Section 4 how to handle counters of level 1 and 2, and then of higher levels. In Section 5 we use counters to encode configurations of Turing Machines and prove the lower bound.

We assume that the reader is familiar with the basic notions of games (see [8] for an overview).

2 Definitions: Game, HPDS

2.1 Game

An *arena* or *game graph* is a tuple (V_0, V_1, E), where $V = V_0 \uplus V_1$ is a set of vertices partitioned into vertices of Player 0 and vertices of Player 1, and $E \subseteq V \times V$ is a set of edges (directed, unlabeled). Starting in a given initial vertex $\pi_0 \in V$, a play in (V_0, V_1, E) proceeds as follows: if $\pi_0 \in V_0$, Player 0 makes the first 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 reachability: a subset $F \subseteq V$ is fixed, and

$$\text{Player 0 wins } \pi \text{ iff } \exists i : \pi_i \in F \ .$$

As soon as F is reached, the play stops. The play can also stop when a position is reached with no outgoing edges. In this case the player who is supposed to move loses. 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 $\pi_n E \pi_{n+1}$. 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.

2.2 Higher Order Pushdown System

We recall the definition from [9] (which is equivalent to the one from [6]), where we slightly change the terminology. A *level 1 store* (or *1-store*) over an alphabet Γ is an arbitrary sequence $\gamma_1\cdots\gamma_\ell$ of elements of Γ, with $\ell \geqslant 0$. A *level k store* (or *k-store*), for $k \geqslant 2$, is a sequence $[s_1]\cdots[s_\ell]$ of $(k-1)$-stores, where $\ell \geqslant 0$. The following operations can be performed on 1-store:

$$\begin{aligned} push_1^{\gamma}(\gamma_1\cdots\gamma_{\ell-1}\gamma_\ell) &:= \gamma_1\cdots\gamma_{\ell-1}\gamma_\ell\gamma \text{ for all } \gamma \in \Gamma \ , \\ pop_1(\gamma_1\cdots\gamma_{\ell-1}\gamma_\ell) &:= \gamma_1\cdots\gamma_{\ell-1} \ , \\ top(\gamma_1\cdots\gamma_{\ell-1}\gamma_\ell) &:= \gamma_\ell \ . \end{aligned}$$

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

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

The operation pop_j is undefined on a store, whose top store of level j is empty. Similarly *top* is undefined on a store, whose top 1-store is empty. We will consider "bottom store symbols" $\bot_j\in\Gamma$ at each level $1\leqslant j\leqslant k$. When a j-store is empty, implicitly its top symbol is $\bot_j$. These symbols can neither be erased nor "pushed". Given Γ and k, the set Op_k of operations (on a store) of level k consists of:

$$push_j \text{ for all } 2\leqslant j\leqslant k,\ push_1^\gamma \text{ for all } \gamma\in\Gamma,\ pop_j \text{ for all } 1\leqslant j\leqslant k, \text{ and } skip \ .$$

The operations $push_j$, allowing to "copy" a part of the store, are responsible for the fact that the hierarchy of HPDS is strict. A *higher order pushdown system* of level k (or k-HPDS) is a tuple $H=(P,\Gamma,\Delta)$ where P is the finite set of control locations, Γ the finite store alphabet, and $\Delta\subseteq P\times\Gamma\times P\times Op_k$ the finite set of (unlabeled) transition rules. We do not consider HPDS as accepting devices, hence there is no input alphabet. The name HPDS is derived from Pushdown System (PDS), it is a HPDA with unlabeled transitions. A *configuration* of an k-HPDS H is a pair (p,s) where $p\in P$ and s is an k-store. The set of k-stores is denoted $\mathscr{S}_k$. A HPDS $H=(P,\Gamma,\Delta)$ defines a *transition graph* (V,E), where $V=\{(p,s):p\in P, s\in\mathscr{S}_k\}$ is the set of all configurations, and

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

For our constructions it would be simpler to assume that k-HPDS can work also on stores of lower levels, in particular on 1-stores. Of course we can always simulate a j-store, for $j<k$ with an k-store but in the notation it requires some additional parenthesis that make it less readable.

To define a game on the graph of a HPDS, we assign a player to each control state, and we consider an initial configuration: a *game structure on a HPDS H* is a tuple $\mathscr{G}=(H,P_0,P_1,s_0)$, where $P=P_0\uplus P_1$ is a partition of the control states of H, and $s_0\in\mathscr{S}_k$. This extends naturally to a partition of the set of configurations: with the notations of Section 2.1, $V_0=P_0\times\mathscr{S}_k$, $V_1=P_1\times\mathscr{S}_k$, and E is defined above.

3 Reduction from the Word Problem

Higher Order Pushdown Automata were originally designed to recognize languages. In the usual way transitions can be labeled by letters from an input alphabet A. A non-deterministic HPDA is defined like a HPDS above except that $\Delta \subseteq P \times \Gamma \times (A \cup \{\varepsilon\}) \times P \times Op_n$. A transition can "read" a symbol from the input word or stay on the same place. The edges of the transition graph are labeled accordingly, and a word is accepted iff there exist a path from an initial configuration to a final configuration. Here the initial configuration can be chosen arbitrarily and the final configurations are defined by the control state.

In an alternating (one-way) HPDA each control state is either existential (in P_0) or universal (in P_1). A computation is a tree, from which the root is $(p_0, s_0, 0)$ where p_0 is the initial control state, s_0 is the initial store content, and 0 represents the leftmost position of the input word. If the input word is $w = w_1 \ldots w_{|w|}$, then every non-leaf node (p, s, i) in the tree must satisfy the following.

- If $p \in P_0$ then *there is* a transition $(p, \gamma, a, p', \theta) \in \Delta$ such that $top(s) = \gamma$ and
  - either $a = w_{i+1}$ and the node (p, s, i) has one child $(p', \theta(s), i + 1)$,
  - or $a = \varepsilon$ and the node (p, s, i) has one child $(p', \theta(s), i)$.
- If $p \in P_1$ then
  - *for each* transition $(p, \gamma, a, p', \theta) \in \Delta$ such that $top(s) = \gamma$ and $a = w_{i+1}$, the node (p, s, i) has a child $(p', \theta(s), i + 1)$,
  - and *for each* transition $(p, \gamma, \varepsilon, p', \theta) \in \Delta$ such that $top(s) = \gamma$, the node (p, s, i) has a child $(p', \theta(s), i)$.

A word w is accepted if there exists a computation tree such that every leaf is (labeled by) an accepting state.

It is well known that there is strong connections between alternation and games (see e.g. [8]) but these connections depends very much on the context (finite/infinite words, epsilon-transitions allowed or not, ...).

Let $Tower$ stand for the "tower of exponentials" function, *i.e.*, $Tower(0, n) = n$ and $Tower(k+1, n) = 2^{Tower(k,n)}$. One of the results of [7] is that given $k > 0$, the class of languages of alternating level k HPDA is the class

$$\bigcup_{d>0} DTIME(Tower(k, dn)) \tag{1}$$

where n is the length of the input word.

Given a k-HPDA $H = (P, \Gamma, \Delta)$ and a word w, our aim is to define a k-HPDS G and a game structure on G such that Player 0 wins if and only if w is accepted by H. Because in the game there is no input word, the idea is to encode w in the control states and in the transitions of G. Let $Q = P \times [0, |w|]$ and $G = (Q, \Gamma, \Delta')$ where

$$\begin{aligned} \Delta' = \{&((p, i), \gamma, (p', i + 1), \theta) : (p, \gamma, a, p', \theta) \in \Delta \text{ and } w_{i+1} = a\} \cup \\ \{&((p, i), \gamma, (p', i), \theta) : (p, \gamma, \varepsilon, p', \theta) \in \Delta\} \end{aligned}$$

The set Q_0 of control states where Player 0 moves is $P_0 \times [0, |w|]$, corresponding to existential states. The set Q_1 where Player 1 moves is $P_1 \times [0, |w|]$, corresponding to universal states. The goal set F is given by the final state(s) of H.

Proposition 1 *Given an alternating (one-way) HPDA H and an input word w one can construct in polynomial time a game structure on a HPDS of the same level and whose size is linear in $|H|.|w|$.*

Note that this proposition can be easily extended to alternating two-way HPDA. From the results of [7] (see (1) above) it follows that for every $k > 0$ and $d > 0$ there is a HPDA H of level k such that the word problem for H cannot be decided in less than $DTIME(Tower(k, dn))$. It follows from this fact and the previous proposition that a game on a HPDS G of level k and size $|G|$ cannot be solved in less than $DTIME(Tower(k, |G|))$.

Theorem 2 *Reachability games on k-HPDS are k-EXPTIME hard.*

Note that given an alternating HPDA H, one can simply remove the transition labels and the input alphabet, keeping the same set of control states. The game structure G obtained is such that: if some word is accepted by H then the game is won by Player 0, but the converse is not true. So there is no clear link between the emptiness problem and the game problem. The situation is different if one considers infinite words (a Büchi acceptance condition), a unary alphabet and no epsilon-transitions.

4 Counters

In the rest of the paper we give an alternative proof of Theorem 2. Our final aim will be to encode computation of k-EXPSPACE bounded alternating Turing machines using k-HPDS. As a preparatory step we will show that using k-HPDS we can manipulate numbers of up to $Tower(k, n)$.

4.1 Alphabets

For each index $i \geq 1$ we consider the alphabet $\Sigma_i = \{a_i, b_i\}$, where a_i and b_i are associated to a and b when regarded as letters of the Turing machine, and to 0 and 1 when regarded as bits (respectively). This conventions will be used through-out the rest of the paper.

4.2 2-counters

As an introductory step we will show that we can count up to 2^{2^n} using 2-store.

Definition 3 *Given $n > 0$, a* 1-counter *of length n is a word*

$$\sigma_{n-1} \cdots \sigma_1 \sigma_0 \in (\Sigma_1)^n \ ,$$

it represents the number $\sum_{i=0}^{n-1} \sigma_i 2^i$ (recall that the letter a_1 represents 0 and the letter b_1 represents 1.)

So we use counters of n bits, and the parameter n is now fixed for the rest of this section without further mentioning.

Definition 4 *A* 2-counter *is a word*

$$\sigma_k \ell_k \cdots \sigma_1 \ell_1 \sigma_0 \ell_0 \ ,$$

where $k = 2^n - 1$*, for all* $i \in [0, 2^n - 1]$ *we have* $\sigma_i \in \Sigma_2$ *and* $\ell_i \in (\Sigma_1)^n$ *is a* 1*-counter representing the number* i*. This* 2*-counter represents the number* $\sum_{i=0}^{2^n-1} \sigma_i 2^i$*.*

We will see how to force Player 0 to write down a proper counter on the store. More precisely we will define states that we call tests. From these states it will be possible to play only a finite game which will be designed to test some properties of the stack. For example, Player 0 will win from $(counter_i, u)$ iff a suffix of u is an i-counter.

From a configuration $(counter_1, u)$ we want Player 0 to win iff on the top of the stack there is a 1-counter; more precisely when u has a suffix $\sigma_2 v \sigma_2'$ for $v \in (\Sigma_1)^n$ and $\sigma_2, \sigma_2' \in \Sigma_2$. To obtain this we let Player 1 pop $n+2$ letters and win if inconsistency is discovered; if no inconsistency is found then Player 0 wins. Similarly we can define $first_1$ and $last_1$ from which Player 0 wins iff on the top of the stack there is a 1-counter representing 0 and $2^n - 1$ respectively.

In a configuration $(equal_1, u)$ we want Player 0 to win iff the two topmost 1-counters have the same value; more precisely when a suffix of the stack u is of the form $\sigma_2 v \sigma_2' v \sigma_2''$ with $v \in (\Sigma_1)^n$, $\sigma_2, \sigma_2', \sigma_2'' \in \Sigma_2$. In the state $equal_1$ Player 1 has the opportunity either to check that there are no two 1-counters on the top of the stack (which is done with $counter_1$), or to select a position where he thinks that the counters differ. To do this he removes from the stack up to n letters in order to reach a desired position. The bit value of this position is stored in the control state and then exactly $n+1$ letters are taken from the stack. Player 1 wins iff the letter on the top of the stack is different from the stored bit value; otherwise Player 0 is the winner.

Similarly, in a configuration $(succ_1, u)$ Player 0 wins iff the two topmost 1-counters represent successive numbers; more precisely when u has a suffix of the form $\sigma_2 v \sigma_2' v' \sigma_2''$ with $v, v' \in (\Sigma_1)^n$ representing consecutive numbers, and $\sigma_2, \sigma_2', \sigma_2'' \in \Sigma_2$. As before Player 1 has an opportunity to check if the stack does not end with two 1-counters. The other possibility is that Player 1 can select a position where he thinks that the value is not right. First he can "pop" any number of letters. During this process, the control state remembers whether the letter b_1 (which represents 1) has already been seen: because lowest bits are popped first, as long as a_1 are popped, we know the corresponding letter in the other counter should be a b_1. After the first b_1, the letters should be the same in the other counter. Then exactly $n+1$ letters or popped (including σ_2') and Player 1 wins if the letter is not right; otherwise Player 0 wins.

Starting from a configuration $(counter_2, u)$ we want Player 0 to win iff on the top of the stack there is a 2-counter; more precisely when u has a suffix $\sigma_3 v \sigma_3'$

with $\sigma_3, \sigma'_3 \in \Sigma_3$ and v a 2-counter. A 2-counter is a sequence of 1-counters, and the task of Player 1 is to show that u has no suffix of the right form. One way to do this is to show that u does not end with a 1-counter or that this last counter does not have value $2^n - 1$. This Player 1 can do with $last_1$ test. Otherwise Player 1 can decide to show that there is some part inside the hypothetical 2-counter that is not right. To do this he is allowed to take letters from the stack up to some Σ_2 letter at which point he can check that the two topmost counters have wrong values (using test $succ_1$). This test can be performed only if Player 0 does not claim that the counter on the top represents 0. If Player 0 claims this then Player 1 can verify by using test $first_1$. It should be clear that if u does not end with a 2-counter then Player 1 can make the right choice of a test and win. On the other hand if u indeed ends with a 2-counter then Player 0 wins no matter what Player 1 chooses. Similarly we can define $first_2$ and $last_2$ from which Player 0 wins iff the top of the store is a 2-counter representing values 0 and $2^{2^n} - 1$ respectively.

Next we want to describe $equal_2$ test for which we will need the power of 2-stores. We want Player 0 to win from a configuration $(equal_2, u)$ iff there is a suffix of u consisting of two 2-counters with the same value; more precisely a suffix of the form $\sigma_3 v \sigma'_3 v \sigma''_3$ with v a 2-counter. If u does not end with two 2-counters then Player 1 can check this with $counter_2$ test and win. If u indeed ends with two 2-counters then Player 1 needs to show that the values of these counters differ. For this he selects, by removing letters from the store, a position in the topmost counter where he thinks that the difference occurs. So the store now finishes with $\sigma v \sigma'$, where $\sigma, \sigma' \in \Sigma_2$ and v is a 1-counter. Next Player 1 performs $push_2$ operation which makes a "copy" of 1-store. The result is:

$$[u'\sigma v \sigma'][u'\sigma v \sigma'] \ .$$

It is then the turn of Player 0 to pop letters from the copy of the store in order to find in the second counter the position with number v. We can be sure that Player 0 stops at some position of the second counter by demanding that in the process he pops precisely one letter from Σ_3. After this the store has the form:

$$[u'\sigma v \sigma'][u''\rho w \rho'] \ .$$

From this configuration Player 0 wins iff $v = w$ and $\sigma' = \rho'$. This test can be done in the same way as $equal_1$ test.

Using similar techniques, it is also possible to define a test $succ_2$ checking that the two topmost 2-counters represent successive numbers (from $[0, 2^{2^n} - 1]$).

4.3 Counters of Higher Levels

As expected k-counters are defined by induction.

Definition 5 *For all $k > 1$ a* k-counter *is a sequence of $(k-1)$-counters of the form:*

$$\sigma_j \ell_j \cdots \sigma_1 \ell_1 \sigma_0 \ell_0 \ ,$$

where $j = Tower(k-1, n)-1$*, for all* $i \in [0, j] : \sigma_i \in \Sigma_k$ *and* ℓ_i *is a* $(k-1)$*-counter representing the number* i*. This* k*-counter represents the number* $\sum_{i=0}^{j} \sigma_i 2^i$*.*

To cope with k-counters, k-HPDS are needed. We want to define for all $k \geqslant 2$ a k-HPDS with the control states with the following properties:

- from $(counter_k, u)$ Player 0 wins iff u ends with a k-counter;
- from (first_k, u), $(last_k, u)$ Player 0 wins iff u ends with a k-counter representing 0 and the maximal value respectively;
- from $(equal_k, u)$ Player 0 wins iff the two last k-counters in u have the same value;
- from $(succ_k, u)$ Player 0 wins iff the two topmost k-counters represent successive numbers.

This is done by induction on k, using hypotheses for lower levels as subprocedures. For $k = 1$ and $k = 2$, we have shown the constructions in the previous subsection. In the following we consider some $k > 2$ and explain now the construction by induction.

Starting from a configuration $(counter_k, u)$ we want Player 0 to win iff on the top of the stack there is a k-counter; more precisely that u has a suffix $\sigma_{k+1} v \sigma'_{k+1}$ with $\sigma_{k+1}, \sigma'_{k+1} \in \Sigma_{k+1}$ and v a k-counter. A k-counter is a sequence of $(k-1)$-counters, and the task of Player 1 is to show that u has no suffix of the right form. One way to do this is to show that u does not end with a $(k-1)$-counter or that this last counter does not have value $Tower(k-1, n) - 1$. This Player 1 can do with $last_{k-1}$ test. Otherwise Player 1 can decide to show that there is some part inside the hypothetical k-counter that is not right. To do this he is allowed to take letters from the stack up to some Σ_k letter at which point he can check that the two consecutive topmost $(k-1)$-counters have wrong values (using test $succ_{k-1}$). This test can be performed only if Player 0 does not claim that the counter on the top represents 0. If Player 0 claims this then Player 1 can verify by using test first_{k-1}. Similarly we can define first_k and $last_k$ test.

Next we want to describe $equal_k$ test for which we will need the power of k-stores. We want Player 0 to win from a configuration $(equal_k, u)$ iff there is a suffix of u consisting of two k-counters with the same value; more precisely a suffix of the form $\xi z \xi' z \xi''$ with z a k-counter and $\xi, \xi, \xi'' \in \Sigma_{k+1}$. If u does not end with two k-counters then Player 1 can check this with $counter_k$ test and win. If u indeed ends with two k-counters then Player 1 needs to show that the values of these counters differ. For this he selects, by removing letters from the store, a position in the topmost counter where he thinks that the difference occurs. So the store now finishes with $\sigma v \sigma'$, where $\sigma, \sigma' \in \Sigma_k$ and v is a $(k-1)$-counter. Next Player 1 performs $push_2$ operation which makes a "copy" of 1-store. The result is of the form:

$$[u' \xi z \xi' z' \sigma v \sigma'][u' \xi z \xi' z' \sigma v \sigma'] \ .$$

This is a 2-store with two elements where z is a k-counter and z' is a prefix of a k-counter.

It is then the turn of Player 0 to pop letters from the copy of the store in order to find in the second counter the position with number v. We can be sure that Player 0 stops at some position of the second counter by demanding that in the process he pops precisely one letter from Σ_{k+1}. After this the store has the form:

$$[u'\xi z\xi' z'\sigma v\sigma'][u'\xi z''\rho w\rho'] \ .$$

From this configuration Player 1 wins iff $v \neq w$ or $\sigma' \neq \rho'$. Checking $\sigma' \neq \rho'$ is easy. The test whether $v = w$ can be done in a similar way as $equal_{k-1}$ test. The difference is that now we have 2-store and $equal_{k-1}$ works on 1-stores. We elaborate the construction as this is the place where the power of k-stores really comes into play.

We will construct states $same_k^i$, for $i < k$, with the property that Player 0 wins in a configuration with a $(k-i+1)$-store of the form

$$s[u\langle r\sigma v\sigma'\rangle][u'\langle r'\rho w\rho'\rangle] \ .$$

iff $\sigma' = \rho'$ and $v = w$ is a i-counter. Here $\sigma, \sigma', \rho, \rho' \in \Sigma_{i+1}$, r, r' are sequences of letters, u, u' are $(k-i)$-stores and s is a $(k-i+1)$-store. The notation $\langle\sigma v\sigma'\rangle$ is to denote the first 1-store in the given store, hence $\langle\ \rangle$ stand for some number of nested [ ] parentheses. The verification we need in the last paragraph is precisely $same_k^{k-1}$ as there we have a 2-store and compare $(k-1)$-counters.

It is quite straightforward to construct $same_k^1$. Player 1 has the right to declare that either $\sigma' \neq \rho'$ or that the counters are not equal. Checking the first case is straightforward. To show that the counters are different, Player 1 chooses $j \leq n$ and pops j letters from w using pop_1. Then j and the top letter are remembered in the control state. Afterward pop_{k-1} is performed and once more j letters are popped. Player 1 wins if the top letter is different from the one stored in the finite control.

To construct $same_k^i$ for $i > 1$ we proceed as follows. Player 1 has the possibility to check if $\sigma' = \rho'$ as before. The other possibility is that he can pop_1 some number of letters finishing on a letter from Σ_i and without popping a letter from Σ_{i+1} in the process. The resulting configuration is of the form:

$$s[u\langle r\sigma v\sigma'\rangle][u'\langle r'\rho w'\tau x\tau'\rangle] \ .$$

The intuition is that Player 1 declares that at position x in v the value is different than τ'. Now $push_{k-i+2}$ is performed giving the configuration

$$\big[s[u\langle r\sigma v\sigma'\rangle][u'\langle r'\rho w'\tau x\tau'\rangle]\big]\ \big[s[u\langle r\sigma v\sigma'\rangle][u'\langle r'\rho w'\tau x\tau'\rangle]\big] \ .$$

As we had $(k-i+1)$-store before, now we have $(k-i+2)$-store consisting of two elements.

Next we let Player 0 to do pop_{k-i} and some number of pop_1 operations to get to the situation

$$\big[s[u\langle r\sigma v\sigma'\rangle][u'\langle r'\rho w'\tau x\tau'\rangle]\big]\ \big[s[u\langle r\sigma v'\gamma y\gamma'\rangle]\big] \ .$$

where he claims that $x = y$ and $\tau' = \gamma'$. This can be checked from $same_k^{i-1}$ state.

The procedure $succ_k$ is implemented similarly to $equal_k$. Here it is not the case that at each position in the counters bits should be the same. Nevertheless the rule for deducing which bit it should be is easy and the difficult part of comparing the positions is done using $same_k^{k-1}$.

5 Encoding Turing Machines

In this section we will show how to encode computations of an EXPSPACE-bounded Turing machine using 2-store. Then we will claim that the construction generalizes to alternating k-EXPSPACE and $(k+1)$-stores.

Fix M, an EXPSPACE-bounded alternating Turing machine (TM), as well as an input word of length n. The set of control states of the TM is denoted Q. A *configuration* of M is a word over $\Delta_2 = \{a_2, b_2\} \cup Q \cup \{\vdash, \dashv\}$ of the form

$$\vdash u_1 \cdots u_i q u_{i+1} \cdots u_j \dashv$$

where $q \in Q$, $\forall k : u_k \in \{a_2, b_2\}$. Here the TM is in state q, reading letter u_{i+1}.

We will encode configurations of M almost in the form of 2-counters to write them in the store of a HPDS. Let $k = 2^n$. A configuration $\sigma_0\sigma_1 \cdots \sigma_{k-1} \in (\Delta_2)^k$ is represented by a word

$$\xi\sigma_{k-1}\ell_{k-1} \cdots \sigma_1\ell_1\sigma_0\ell_0\xi \ ,$$

where for all $i \in [0, 2^n - 1]$: $\sigma_i \in \Delta_2$, $\ell_i \in (\Sigma_1)^n$ is a 1-counter representing the number i, and $\xi \in \Sigma_3$ is a separator.

A computation is represented as a string obtained by concatenation of configurations. The game will proceed as follows: departing from the initial configuration of the Turing machine (the input word), Player 0 is in charge of building an accepting run and Player 1 is in charge of checking that no error occurs. Player 0 simply writes letter by letter a configuration. If the state of the configuration is existential then after writing down the configuration Player 0 writes also a transition he wants to perform. Otherwise it is Player 1 who writes the transition. Then Player 0 continues with writing a next configuration that he claims is the configuration obtained by the transition that was just written down. This process continues until a configuration with a final state is reached. At the end of writing each configuration Player 1 has the opportunity to check if the last two configurations on the stack indeed follow from each other by the transition that is written between them.

Let us describe some details of this construction. Applying a transition rule of the Turing Machine consists in rewriting only three letters: u_i, q and u_{i+1} in the notation of the example above. To check that the transition is legal, we will proceed in several steps. After writing a configuration, ended by a separator $\xi \in \Sigma_3$, Player 0 has to write again the three letters $u_i q u_{i+1}$. Then, depending whether state q is existential or universal in the TM, Player 0 or Player 1 writes three other letters of Δ_2, say $q'ac$, such that $(u_i q u_{i+1}, q'ac)$ is a transition rule of the TM. The other player can test that this transition rule is indeed in the TM.

After that Player 0 has to write down the configuration obtained by the chosen transition, and Player 1 has the opportunity to test whether this is correct. To do this he has several possibilities. First he can check that the newly written configuration is of a correct form, using a test similar to $counter_2$, replacing Σ_2 by Δ_2.

Otherwise he can check that this two last configurations are identical, except for the part involved in the transition rule. The store at this point is:

$$s\ \xi c_1 \xi\ u_i q u_{i+1} q' ac\ \xi c_2 \xi\ ,$$

where s is a prefix of computation, c_1 and c_2 are the last two configurations separated by the chosen transition. We describe a game from a state $trans_2$ such that Player 0 wins from $trans_2$ and the store as above iff the two topmost configurations obey the transition rule written between them. The test $trans_2$ has the same structure as the test $equal_2$. Player 1 has first to pop letters to select a position in the configuration, that is a 1-counter. Each time he wants to pop next 1-counter he asks Player 0 if this position is the rightmost position involved in the transition or not. If yes then Player 1 has to pop three counters at the time, if not he pops one counter. Finally, Player 1 stops at a position where he thinks that an error occurs. He asks Player 0 if this position is the rightmost position of the transition. If Player 0 says that it is not then it is tested that at the same position in the preceding configuration there is the same letter; this is done in the same way as $equal_2$ test.

If Player 0 claims that the chosen position is the rightmost position of the ones involved in the transition then the test is slightly more complex. A $push_2$ is performed and the store becomes

$$[s\ \xi c_1 \xi\ u_i q u_{i+1} q' ac\ \xi c_2'\ \rho v \rho' v' \rho'' v''][s\ \xi c_1 \xi\ u_i q u_{i+1} q' ac\ \xi c_2'\ \rho v \rho' v' \rho'' v'']\ ,$$

where c_2' is a prefix of c_2, $\rho, \rho', \rho'' \in \Delta_2$ and v, v', v'' are 1-counters. Player 1 has the opportunity to check that $q'ac = \rho\rho'\rho''$, which is easy to implement. Player 1 has also the opportunity to let Player 0 find the position in c_1 corresponding to v'' and then test that the corresponding letters from Δ_2 are exactly $u_i q u_{i+1}$; this is implemented in a similar way as in $equal_2$ test.

The game is won by Player 0 iff he can write an accepting configuration of the TM without Player 1 ever challenging him, or if Player 1 fails in some test. In other words the game is won by Player 1 iff he can prove that Player 0 was cheating somewhere or if Player 0 never reaches an accepting configuration of the TM. Examining the construction one can see that we need $\mathcal{O}(n^2 + |M|)$ states in 2-HPDS to carry out the described constructions. So we have a poly-time reduction of the acceptance problem of alternating EXPSPACE Turing Machines to the problem of determining the winner in a reachability game over a 2-HPDS.

Theorem 6 *Reachability games on 2-HPDS are 2-EXPTIME hard.*

Together with the double exponential time solution of the more general parity games from [2], we have:

Corollary 7 *Reachability/parity games on* 2*-HPDS are complete for* 2-EXPTIME.

Using the constructions of Section 4.3, it is easy to extend the encoding above and show that alternating k-EXPSPACE Turing Machines can be simulated by $(k+1)$-HPDS. Together with the results from [2] we get:

Theorem 8 *Reachability/parity games on k-HPDS are complete for k-*EXPTIME.

This result gives also a new proof that the hierarchy of HPDA is strict, and together with [2], that the Caucal hierarchy is also strict.

6 Conclusion

The k-EXPTIME lower bound that we have proved in this paper shows that games are difficult on HPDA, even the simplest ones : reachability games. Surprisingly the complexity for solving parity games is the same as for reachability games. It is open to find algorithms or lower bounds for the model checking of other logics like CTL or LTL, that are weaker than the μ-calculus.

Acknowledgment

Many thanks to Luke Ong and Olivier Serre for interesting discussions.

References

1. A. Bouajjani and A. Meyer, *Symbolic Reachability Analysis of Higher-Order Context-Free Processes*, FSTTCS'04, LNCS 3328, pp. 135–147, 2004.
2. T. Cachat, *Higher order pushdown automata, the Caucal hierarchy of graphs and parity games*, ICALP'03, LNCS 2719, pp. 556–569, 2003.
3. A. Carayol, *Regular Sets of Higher-Order Pushdown Stacks*, MFCS'05, LNCS 3618, pp. 168–179, 2005.
4. A. Carayol and S. Wöhrle, *The Caucal hierarchy of infinite graphs in terms of logic and higher-order pushdown automata*, FSTTCS'03, LNCS 2914, pp. 112–123, 2003
5. D. Caucal, *On infinite terms having a decidable monadic theory*, MFCS'02, LNCS 2420, pp. 165–176, 2002.
6. J. Engelfriet, *Iterated push-down automata*, 15th STOC, pp. 365–373, 1983.
7. J. Engelfriet, *Iterated stack automata and complexity classes* , Information and Computation, 95(1), pp. 21–75, 1991.
8. E. Grädel, W. Thomas and T. Wilke eds., *Automata, Logics, and Infinite Games*, A Guide to Current Research, LNCS 2500, 2002.
9. T. Knapik, D. Niwinski and P. Urzyczyn, *Higher-order pushdown trees are easy*, FoSSaCS'02, LNCS 2303, pp. 205–222, 2002.
10. 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.
11. I. Walukiewicz, *Difficult configurations — On the complexity of $LTrL$*, ICALP'98, LNCS 1443, pp. 140–151, 1998.