Fast Mu-calculus Model Checking when Tree-width is Bounded

Jan Obdržálek

Laboratory for Foundations of Computer Science
The University of Edinburgh
j.obdrzalek@ed.ac.uk

Abstract. We show that the model checking problem for μ-calculus on graphs of bounded tree-width can be solved in time linear in the size of the system. The result is presented by first showing a related result: the winner in a parity game on a graph of bounded tree-width can be decided in polynomial time. The given algorithm is then modified to obtain a new algorithm for μ-calculus model checking. One possible use of this algorithm may be software verification, since control flow graphs of programs written in high-level languages are usually of bounded tree-width. Finally, we discuss some implications and future work.

1 Introduction

The modal μ-calculus introduced by Kozen [Koz83] is a very expressive fixpoint logic capable of specifying a wide range of properties of (non-terminating) programs, such as safety, liveness, fairness etc. Moreover, many important temporal logics were shown to be fragments of the modal μ-calculus [EJS93,EL86].
Even though modal μ-calculus was extensively studied, the exact complexity of model checking problem for this logic is not known. The original result of Emerson and Lei [EL86] states the following: *The model checking problem for a formula of size m and alternation depth d on a system of size n is $\mathcal{O}(m \cdot n^{d+1})$.* So model checking is exponential in alternation depth of the formula. In [EJS93] the complexity was shown to be in NP$\cap$co-NP, though it is unlikely for the problem to be NP-complete. A substantial effort was undertaken to find a polynomial model checking algorithm. The only improvement since the original work came from Long et al. [LBC⁺94]. Their delicate result allowed the complexity to be decreased to $\mathcal{O}(m \cdot n^{\lceil d/2 \rceil + 1})$.
In contrast with the explicit computation of fixpoints in the original algorithm [EL86], the paper [EJS93] shows that the model checking problem for μ-calculus is equivalent to the non-emptiness problem for automata on infinite trees with parity acceptance condition. As a consequence of this fact it can be shown that this is equivalent to the problem of deciding a winner in parity games (there is a polynomial-time reduction). See also [Sti95]. Parity games were therefore extensively studied [Jur00,VJ00], but so far this research has not come up with a polynomial algorithm.

Tree-width is a graph theoretic concept introduced first by Robertson and Seymour [RS84] in their work on graph minors. Roughly speaking, tree-width measures how close is the given graph to being a tree. Graphs with low tree-width then allow a decomposition of the problem being solved into subproblems, decreasing the overall complexity – some in general NP-complete problems were show to be polynomial on these graphs. (Following the intuition that solving problems on trees is *much* easier than on general graphs. E.g. modal μ-calculus model checking is linear on trees.) See Bodlaender's paper [Bod97] for an excellent survey.
Even though the concept of tree-width is quite restrictive, in practice the systems considered are surprisingly often of a low tree-width. In [Tho98] it was shown that all C programs (resp. their control-flow graphs) are of tree-width at most 6, and Pascal programs of tree-width at most 3! This result does not hold for Java, as the labelled versions of `break` and `continue` can be as harmful as `goto` [GMT02]. In practice, however, programs with high tree-width do not appear (since they are written by sane humans).
In this paper we show that for graphs of bounded tree-width, the μ-calculus model checking problem can be solved in time $\mathcal{O}(n \cdot \alpha(m, k))$ on systems of tree-width k. In general this is a consequence of a general theorem of Courcelle [Cou90]: For a fixed MSO formula φ and a graph of bounded tree-width the model checking problem can be solved in time linear in the size of the graph. This result, however, does not provide any estimate on $\alpha(m, k)$ (except for being "large"). Recently it was shown [FG02] that α is not even elementary. Moreover the algorithm itself is quite complicated and does not provide any insight into what are the results/strategies in the underlying game.
In contrast, our algorithm does not require translation into MSO. Its complexity is clearly expressed in the parameters m, k and d and in addition one can easily follow the workings of the algorithm as well as the evolving strategies. We start by first proving a related important result: That a winner in a parity game can be determined in polynomial time (linear when the number of colors is fixed). This result is new and does not follow from [Cou90]. We then extend this to give a new μ-calculus model checking algorithm.

Acknowledgement. I am indebted to Colin Stirling for his invaluable support, guidance, and for suggesting me to work on this topic in the first place.

2 Parity Games and μ-calculus

2.1 Parity Games

The *Parity game* $\mathcal{G} = (V_0, V_1, E, \lambda)$ consists of a directed graph $D = (V, E)$, where V is disjoint union of V_0 and V_1, and a parity function $\lambda : V \to \mathbb{N}$ ($0 \notin \mathbb{N}$). For clarity we assume that for every vertex of V there is at least one outgoing edge in D.
The game is played by two players P_0 and P_1 (called also *Even* and *Odd*), who form an infinite path in D by moving a token along the edges. The game starts

in an initial vertex and players play indefinitely in the following way. If the token is on vertex v of V_0, then player P_0 moves the token along some edge with tail v. If the token is on vertex of V_1, player P_1 moves the token. As a result, the players form an infinite path $\pi : \pi_1\pi_2\ldots$, which corresponds to the infinite sequence of priorities $Inf(\pi) : p(\pi_1)p(\pi_2)\ldots$ Player P_1 wins the path (play) π if the highest priority appearing infinitely often in $Inf(\pi)$ is odd, otherwise player P_0 wins.
Next we define the notion of strategy for player P_0 (it is dual for player P_1). A (memoryless) *strategy* S assigns to every vertex $v \in V_0$ one of the edges of E with a tail in v. More formally, S is a function $S : V_0 \rightarrow V$ s.t. $\forall v \in V_0.S(v) = w \implies (v, w) \in E$. A player plays *using a strategy* S, if in his vertex v he always chooses $S(v)$ as the next vertex. We say that a strategy S is winning for player P_0 (resp. player P_1) from $v \in V$, if he wins every play starting in v using this strategy.
From the well known determinacy result [EJ91] we know that the vertex set V can be divided into two sets W_0 and W_1, containing the vertices for which the player P_0 (P_1) has a winning strategy. Moreover, it is shown sufficient to take just memoryless strategies defined as above. Every strategy for player P_0 gives us a graph D^S:

Definition 1 (D^S). *For a graph $D = (V, E)$ and a strategy S we define $D^S = (V, E')$ to be a subgraph of D, where $E' = E \setminus \{(v, w) \in E \mid v \in V_0 \wedge S(v) \neq w\}$ In other words, we remove all the edges leaving the vertex v of V_0 except for the one corresponding to the strategy S.*

It is easy to show, that on this graph P_0 wins if there is no cycle reachable from the initial vertex such that P_1 wins this cycle.

2.2 Modal μ-calculus

The syntax of modal μ-calculus we use (positive normal form) is defined by:

$$\varphi ::= Z \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu Z.\varphi \mid \nu Z.\varphi$$

Z here ranges over a family of propositional variables. We do not give the semantics here, as we assume the reader is familiar with the modal μ-calculus (see [Sti01] for a good introduction). In the text we also deal only with closed formulas (i.e. not containing any free variables).
We evaluate μ-calculus formulas on labelled transition systems (LTSs), an LTS $\mathcal{T}$ being a triple $(P, Act, \longrightarrow)$. These LTS can for example represent a control flow graph of some program. In the rest of this text we use $p, q, \ldots$ to denote states (members of P) and write $p \stackrel{a}{\longrightarrow} q$ for $(p, a, q) \in \longrightarrow$.
There is a well known algorithm for constructing a parity game $\mathcal{G} = (V_0, V_1, E, \lambda)$ corresponding to model checking problem for $\mathcal{T} = (P, Act, \longrightarrow)$ and φ. This construction is basically by computing a synchronised product of the graphs of φ and $\mathcal{G}$ (see eg. [Sti01]). The following theorem holds:

Theorem 1 ([Sti01]). *Player V_0 has a winning strategy for (p, φ) in the parity game $\mathcal{G}$ iff $p \models_{\mathcal{T}} \varphi$.*

3 Tree Decompositions

Here we present the relevant facts about tree decompositions and tree-width, which will be needed later in the text.

Definition 2 (Tree decomposition). *A* tree decomposition *of an (undirected) graph $G = (V, E)$ is a pair $(\mathcal{X}, T)$, where $T = (I, F)$ is a tree (its vertices are called* nodes *throughout this paper) and $\mathcal{X} = \{X_i \mid i \in I\}$ family of subsets of V such that:*

- $\bigcup_{i \in I} X_i = V$,
- *for every edge $\{v, w\} \in E$ there exists an $i \in I$ s.t. $\{v, w\} \subseteq X_i$, and*
- *for all $i, j, k \in I$ if j is on the (unique) path from i to k in T, then $X_i \cap X_k \subseteq X_j$.*

The width *of a tree decomposition $(\mathcal{X}, T)$ is $\max_{i \in I} |X_i| - 1$. The* tree-width *of a graph G is the minimum width over all possible tree decompositions of G. Trees have tree-width one. One obtains an equivalent definition if the third condition is replaced by:*

> *For all $v \in V$, the set of nodes $\{i \in I \mid v \in X_i\}$ forms a connected subtree of T.*

Let i be a node of T in a tree decomposition $(\mathcal{X}, T)$ of D. Then we use the following notation:

T_i – subtree of T rooted in i
V_i – vertices of D appearing in T_i – i.e. $V_i = \bigcup_{j \in T_i} X_j$
D_i – subgraph of D induced by V_i – i.e. $V(D_i) = V_i$ and $E(D_i) = \{(s, t) \in E \mid s, t \in V_i\}$
$D \setminus W$ – subgraph of D induced by vertices $V(D) \setminus W$

The following fact about tree decompositions is one of the basic properties of graphs of bounded tree-width, which allows for all the interesting results.

Fact 1. *Let $(\mathcal{X}, T)$ be a tree decomposition and i a node of T. Then the only vertices in V_i adjacent to vertices $V \setminus V_i$ are those belonging to X_i. In other words, X_i is an* interface *between D_i and the rest of the graph.*

The following notion of *nice tree decomposition* allows us to significantly simplify the construction of our algorithm. This choice is justified by Lemma 1.

Definition 3 (Nice tree decomposition). *Tree decomposition $(\mathcal{X}, T)$ is called* nice tree decomposition, *if the following three conditions are satisfied:*

1. *every node of T has at most two children,*
2. *if a node i has two children j and k, then $X_i = X_j = X_k$, and*
3. *if a node i has one child j, then either $|X_i| = |X_j| + 1$ and $X_j \subseteq X_i$ or $|X_i| = |X_j| - 1$ and $X_i \subseteq X_j$.*

Lemma 1 (See [Klo94]). *Every graph G of tree-width k has a nice tree decomposition of width k. Furthermore, if n is the number of vertices of G then there exists a nice tree decomposition with at most $4n$ nodes. Moreover, this decomposition can be constructed in $\mathcal{O}(n)$ time.*

In a nice tree decomposition $(\mathcal{X}, T)$ every node is one of four possible types. These types are:

Start If a node is a leaf, it is called a *start node*.
Join If a node has two children, it is called a *join node* (note that the subgraphs of its children are then disjunct except for X_i).
Forget If a node i has one child j and $|X_i| < |X_j|$, node i is called a *forget node*.
Introduce If a node i has one child j and $|X_i| > |X_j|$, node i is called an *introduce node*.

Moreover, we may assume that *start* nodes contain only a single vertex. If this is not the case, we can transform the nice tree decomposition into one having this property by adding a chain of *introduce* nodes in place of non-conforming *start* nodes. We will also need a notion of terminal graph, which is closely related to tree decompositions.

Definition 4 (Terminal graph). *A* terminal graph *is a triple $H = (V, E, X)$, where (V, E) is a graph and X an ordered subset of vertices of V called* terminals. *The operation $\oplus$ is defined on pairs of terminal graphs with the same number of terminals: $H \oplus H'$ is obtained by taking the disjoint union of H and H' and then identifying the i-th terminal of H with i-th terminal of H' for $i \in \{1, \ldots, l\}$. A terminal graph H is a* terminal subgraph *of a graph G iff there exists a terminal graph H' s.t. $G = H \oplus H'$. Finally we define H_i to be D_i taken as a terminal subgraph with X_i as a set of its terminals (the ordering of X_i is not important here).*

4 The Algorithm for Parity Games

In this section we give the algorithm for solving parity games on graphs of bounded tree-width in polynomial time. First let us fix a parity game $\mathcal{G} = (V_0, V_1, E, \lambda)$. In the text we will often use only $D = (V, E)$ (where $V = V_0 \cup V_1$) to denote the game $\mathcal{G}$ – sets V_0, V_1, and λ are then implied by context. Then we take G to be the undirected graph which arises from D by forgetting the orientation of edges (i.e. an edge $e = \{x, y\}$ of G can correspond to two edges (x, y) and (y, x) of D). We also assume that we have a tree decomposition $(\overline{\mathcal{X}}, \overline{T})$ of G of tree-width k and there is no cycle of length one. (If there is such a cycle, than the winner is known and we can safely remove the edge.) In the case of control flow graphs of programs, tree decomposition can be obtained by a simple syntactical analysis (see [Tho98]).
We start by converting a tree decomposition $(\overline{\mathcal{X}}, \overline{T})$ into a nice tree decomposition $(\mathcal{X}, T)$ of the same width – this can be done using Lemma 1. Our algorithm

then follows a general approach for solving problems on graphs of bounded tree-width (see [Bod97]). The crux of the algorithm lies in computing a bounded representation of the exponential set of strategies. Given node i of T, we only need to know effect of a given strategy for vertices in the interface X_i, size of which is bounded by a (small) constant. We compute the effects of strategies (called *full borders*) at nodes of T in a bottom-up manner. From a full border for the root we can then quickly decide the winner for vertices in the root node. Using force-sets or some similar technique, winners for the other vertices can be found as well (the complexity then increases by at most a factor of n).

4.1 Borders

In this section we will define the notion of *full border* for a graph G and show it is adequate. We need to have a means to record the "winner" for a set of paths. First, for a single finite path π we put $\mathcal{R}(\pi) = \max(\lambda(\pi_1), \ldots, \lambda(\pi_n))$. (Obviously, the highest vertex determines the winner.) Next, we need a "reward" ordering $\sqsubseteq$: Let $p_1, p_2 \in \mathbb{N}$. Then $p_1 \sqsubset p_2$ if p_1 is odd and p_2 is even, or $p_1 > p_2$ and p_1, p_2 are both odd, or $p_1 < p_2$ and p_1, p_2 are both even. We write $p_1 \sqsubseteq p_2$ if $p_1 \sqsubset p_2$ or $p_1 = p_2$. Finally for a set of paths Π we define $\overline{\mathcal{R}} = \min_{\sqsubseteq}\{\mathcal{R}(\pi) \mid \pi \in \Pi\}$.
The intuition behind this definition is the following: In the game where there is a fixed strategy S for player P_0 (i.e. every vertex in V_0 has a single outgoing edge), it is the player P_1 who decides which way to go if there are multiple choices available. To maximise her chances of winning, she chooses the path where the highest priority is odd, and the maximum of all such paths. If player P_0 wins all the paths, then P_1 tries to minimise the harm by selecting the one with the lowest winning priority. We will need also the following property, which is easy to prove:

Lemma 2. *Let D be a graph and $u, v, w \in V$. Let $\Pi_D(u, v)$ be the set of all paths from u to v in D, and $\Pi_D(u, v, w)$ set of those which pass through w. Then*

$$\overline{\mathcal{R}}(\Pi_D(u, v, w)) = \max(\overline{\mathcal{R}}(\Pi_D(u, w)), \overline{\mathcal{R}}(\Pi_D(w, v)))$$

A border of i tells us what happens *inside* the subgraph D_i – i.e. we take vertices of X_i as entry points for D_i, but not as its inner vertices. We start with some useful definitions.
An *i-path* in a graph D is path $\pi : \pi_1\pi_2 \ldots \pi_k$, where $\pi_1 = s, \pi_k = t \in V_i$ and $\{\pi_2 \ldots \pi_{k-1}\} \subseteq V_i \setminus X_i$. We allow $s = t$. For $s, t \in X_i$ we use $\Pi^i_D(s, t)$ to denote the set of all i-paths from s to t in D.
An *i-internal cycle* in a graph D is a cycle $\sigma = \sigma_1\sigma_2 \ldots \sigma_k$, where $\sigma_1 = \sigma_k$ and $\sigma_1, \ldots, \sigma_k \in V_i \setminus X_i$. Vertex $w \in X_i$ is called *i-losing*, if there is an i-internal cycle σ won by P_1 and a path $\pi : \pi_1\pi_2 \ldots \pi_k$ s.t. $\pi_1 = w, \pi_2 \ldots \pi_k \in V_i \setminus X_i$ and $\pi_k \in \sigma$.

Definition 5 (border). A border b *of i is a function $b : X_i \to \{\perp, \circ, 2^{X_i \times P}\}$. A border b (of i)* corresponds *to a strategy S, iff $\forall v \in X_i$. $b(v) =$*

- $\perp$ *iff* v *is* i*-losing in* D^S,
- $\circ$ *iff* $v \in V_0$ *has no outgoing edge in* D_i^S, *and*
- $\{(w,p) \mid w \in X_i \ \wedge \ \Pi^i_{D^S}(v,w) \neq \emptyset \ \wedge \ p = \overline{\mathcal{R}}(\Pi^i_{D^S}(v,w))\}$ *otherwise.*

First note that many strategies can correspond to a single border. This "compression" makes the algorithm work. Members of $b(s)$ are called *entries*. Note that for each $v, w \in X_i$ there is at most one entry $(w,p) \in b(v)$ when $b(v) \neq \perp$ or $\circ$. For the sake of clarity, we will overload the notation a little bit and write $b(s,t) = p$ as a shorthand for $(t,p) \in b(s)$ and $b(s,t) = 0$ when there is no pair $(t,p) \in b(s)$. In addition to border being a function, we can look at it as being a table of priorities with dimensions $(k+1) \times (k+1)$. In this table the rows and columns are labelled by vertices. Likewise if symbols $\perp$ or $\circ$ appear in a row, then the whole row must be marked by this symbol.
Next we want to show that border (corresponding to S) is well defined – i.e. that it contains all the information we need to know about D_i^S in order to check whether P_0 wins using the strategy S. To do this we first define a notion of *link*:

Definition 6 (link). *A* link *of a border* b *(of a node* i*) is a terminal graph* $H = (X_i \cup W \cup \{v_\perp\}, E \cup (v_\perp, v_\perp), X_i)$ *and priority function* λ *s.t.:*

- *for every pair* $s, t \in X_i$ *with* $b(s,t) \neq 0$ *we put a new vertex* w *into* W *and two edges* (s,w) *and* (w,t) *into* E. *We also set* $\lambda(w) = b(s,t)$.
- *for every* $s \in X_i$ *with* $b(s) = \perp$ *we insert an edge* $(s, v_\perp)$ *into* E.
- $p(v)$ *for* $v \in X_i$ *is the same as in original game,* $\lambda(v_\perp) = 1$.

We also write $Link(b)$ *for the link of the border* b.

In other words a link of b, where $b \in B(i)$ corresponds to a strategy S, is just a graph having the same properties w.r.t. winning the game as D_i^S does. The formal proof of this statement is subject of the following lemma:

Lemma 3 (Border is well defined). *Let* b *be a border of* i *corresponding to some strategy* S. *Let* H *be s.t.* $D^S = D_i^S \oplus H$ *and* v *a vertex in* $V \setminus (V_i \setminus X_i)$. *Then* P_1 *has a winning strategy for* v *in* D^S *iff she has a winning strategy for* v *in* $L = Link(b) \oplus H$.

Proof. $\Rightarrow$ Suppose P_0 does not win in D^S. Then there must be a cycle $\sigma = \sigma_1 \ldots \sigma_k$ reachable from v s.t. P_1 wins this cycle. Let π be the path from v to a vertex of σ. There are two cases to be considered:

1. $V(\sigma) \subseteq V_i \setminus X_i$
   Then there must be $i \in \mathbb{N}$ s.t. $\pi_i = w \in X_i$ and $b(w) = \perp$ by Fact 1. From definition of $Link(b)$ player P_1 has a winning strategy for v in L (she can force play to $v_\perp$ and then loop through this vertex).
2. Otherwise
   Let j be s.t. $\sigma_j \in X_i$ and $\forall i \leq j . \sigma_i \in V_i \setminus X_i$ (such a j must exist). Then σ_j is also reachable in L (by definition of border correspondence and $Link(b)$). Moreover, let σ' be a cycle obtained from σ by substituting every sequence $s = \sigma_i \ldots \sigma_{i+l} = t$, where $s, t \in X_i$ and $\{\sigma_{i+1} \ldots \sigma_{i+l}\} \subseteq V_i \setminus X_i$, by path $s w_{s,t} t$. Then σ' is a cycle of L and it is easy to check that P_1 also wins the cycle σ' of L.

$\Leftarrow$ similar

$\square$

Definition 7 (full border). *A* full border $B(i)$ *(of node i) is just a set of all borders of i corresponding to some strategy S.*

The following important corollary says how we can derive the desired information from the full border for the root r of T.

Corollary 1. *Let $(\mathcal{X}, T)$ be a tree decomposition of D, r its root node and $v \in X_r$ a vertex of D. Then P_0 has a winning strategy for v in D iff there is $b \in B(r)$ s.t. P_0 has a winning strategy for v in $Link(b)$.*

It should be noted that the check in the corollary above can be done in constant time, which depends only on the tree-width of D.

4.2 Computing Full Border

Having a nice tree decomposition $(\mathcal{X}, T)$, we compute $B(i)$ for every node i of T in a bottom-up manner. Here we give an algorithm for each of the four node types. Detailed algorithms written in C-like pseudocode can be found in the appendix.

Start Node If i is a Start node, then it contains only a single vertex v. Two cases:

$v \in V_0$ - we set $B(i) = \{b\}$ where $b(v) = \circ$, since we have to postpone the choice.
$v \in V_1$ - we set $B(i) = \{b\}$ where $b(v) = \emptyset$ (no choice here).

Forget Node – *detecting cycles.* Let i be a forget node with a single child j and $X_j = X_i \cup \{v\}$. By definition of tree-width we know that there is no edge connecting v with $D \setminus V_i$, since v does not appear anywhere in the part of T yet to be explored. For $b \in B(j)$:

1. Create b' from b by copying all entries not containing v.
2. If $b(v) = \bot$ or $b(v, v) = p$ for p odd, set $b'(u) = \bot$ for all vertices $u \in X_i$ s.t. $b(u, v) \neq 0$, since u is i-losing.
3. Otherwise let s, t be a pair of vertices s.t. $b(s, v) = p_1$ and $b(v, t) = p_2$ and let $p = \max(p_1, p_2)$. Then $b'(s, t) = p$ if $b(s, t) = 0$, and $b'(s, t) = \overline{\mathcal{R}}(b(s, t), p)$ otherwise.
4. Put b' into $B(i)$.

Introduce Node – *adding new borders.* Let i be an introduce node with a single child j and $X_i = X_j \cup \{v\}$. For every border $b \in B(j)$:

1. Create a copy b' of b.
2. For every edge $(v, w) \in E$ s.t. $w \in X_i$ insert an entry $b'(v, w) = \max(\lambda(v), \lambda(w))$. If $v \in V_1$, then we add all such entries. Otherwise $v \in V_0$ and we create a new border for each of the edges (these borders correspond to different strategies) and also include the possibility $b'(v) = \circ$.
3. For every border b' created in the previous step and every edge $(u, v) \in E, u \in X_i \cap V_1$ we set $b'(u, v) = \max(\lambda(u), \lambda(v))$.
4. For every $W \subseteq \{u \in X_i \mid (u, v) \in E \wedge b(u) = \circ\}$ we create a new copy b'' of b' with $b''(u, v) = \max(\lambda(u), \lambda(v))$ for $u \in W$. Every such subset may correspond to some strategy S for D.
5. Remove every newly created b which can not correspond to a strategy. This happens when there is $v' \in X_i$ s.t. $b(v') = \circ$, but there is no $w \in V \setminus V_i$ s.t. $(v', w) \in E$. In other words - we have postponed the choice, but there are no remaining edges to choose from.
6. Add the resulting borders to $B(i)$.

Join Node Let i be a join node with j_1 and j_2 as its children and $B(j_1)$ and $B(j_2)$ their full borders. It is enough to define the join operation for every pair (b_1, b_2) where $b_1 \in B(j_1)$ and $b_2 \in B(j_2)$. Each such join will result in a new border b to be added into full border $B(i)$ of i.

Note on joining To get correct results, we must not apply the join operation to two borders which do not correspond to some common strategy. By definition of tree-width $V_{j_1} \cap V_{j_2} = X_i$ (i.e. D_{j_1} and D_{j_2} are disjoint except for their common interface), so we only have to check the choices made for P_0 vertices in X_i. To allow for the check, in every border we remember the choice made for these vertices. Look at the following table. For every $v \in X_i \cap V_0$ there is a single edge (v, w) in D^S. The possible cases are:

1) $w \in V \setminus V_i$ 2) $w \in X_i$
3) $w \in V_{j_1} \setminus X_i$ 4) $w \in V_{j_2} \setminus X_i$

To distinguish among these cases, we already can deduce some information from $b_1(v)$ and $b_2(v)$ (we use S, S' to mark the fact that $b(s)$ is a non-empty set of pairs):

$b_1(v) \backslash b_2(v)$	$\circ$	$\perp$	S'
$\circ$	1)	4)	4?
$\perp$	3)	–	–
S	3?	–	2?

As we see, there are some cases which need further checking (these are marked "?" in the table). Note that $b(v, w) \neq 0$ when w is in X_i, so we can remember the choice by selecting a single pair from $b(v)$ for every $v \in X_i \cap V_0$. No pair selected then corresponds to w being in $V_i \setminus X_i$. Our algorithms can be easily extended to keep track of this information.

Algorithm For a pair of borders b_1 and b_2 we first check whether they can be both borders of a same strategy as outlined above. Now we have to "merge" b_1 and b_2 into b by going through all vertices in X_i. The only interesting case is when $b_1(s,t) = p_1$ and $b_2(s,t) = p_2$, in which case we set $b(s,t) = \overline{\mathcal{R}}(p_1, p_2)$. Once finished, we must remove borders which cannot correspond to a strategy (see the Introduce node).

4.3 Correctness and Complexity

The following theorem states that our algorithm is correct.

Theorem 2 (correctness). *Let $\mathcal{G} = (V_0, V_1, E, \lambda)$ be a parity game and $(\mathcal{X}, T)$ a nice tree decomposition of D. Let i be a node of T and $B(i)$ the set computed using our algorithm for a node i. Then $B(i)$ is a full border for i.*

Proof (Sketch). The proof goes by induction on structure of T. For every node we have to prove that

1. For every strategy S there is $b \in B(i)$ corresponding to S
2. Every $b \in B(i)$ corresponds to some strategy S.

The proof itself is straightforward, since all the necessary facts were mentioned in Sect. 4.2 alongside the algorithms. □

Theorem 3 (complexity). *Let $\mathcal{G} = (V_0, V_1, E, \lambda)$ be a parity game with tree decomposition $(\mathcal{X}, T)$ of $D = (V = V_0 \cup V_1, E)$ of width k. Let $\mathbf{p} = |\{\lambda(v) \mid v \in V\}|$ be the number of priorities. The algorithm described in this section runs in time $\mathcal{O}(n \cdot k^2 \mathbf{p}^{2(k+1)^2})$ where n is the number of vertices of D.*

Proof. There are at most $\mathbf{p}^{(k+1)^2}$ different borders in a full border. That is because the dimensions of a single border are at most $(k+1) \times (k+1)$, and a border is nothing else than a table of priorities. It can be easily seen that full border for each of the four node types can be computed in constant time depending only on k. A precise analysis shows that this time has an upper bound of $k^2 \cdot \mathbf{p}^{2(k+1)^2}$.
According to Lemma 1 we know that a nice tree decomposition has at most $4n$ nodes and can be constructed in $\mathcal{O}(n)$ time. This gives the bound $\mathcal{O}(n \cdot k^2 \mathbf{p}^{2(k+1)^2})$. It remains to mention that in the general case the number of priorities $\mathbf{p}$ is from the range $\langle 1, n \rangle$, and therefore our algorithm is *polynomial* in n. □

We have been able to identify examples of parity games for which the standard algorithm based on computing the approximants needs exponential time, but which are of very low tree-width. In [Mad97] there is an example of such a parity game. This example is parametrized by n – the number of vertices. The game graph in Fig. 1 shows an instance of size 10. In our notation vertices of $P_0(P_1)$ are shown as circles (squares) and the number associated with each vertex shows its priority. Note that the tree-width of this game graph is only 2 (this value does not depend on n).

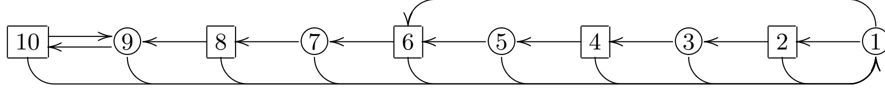

Fig. 1. Parity game example

5 Adaptation to μ-calculus

In this section we explain how to adapt the algorithm for parity games to μ-calculus model checking. As an instance of a model checking problem, we are given an LTS $\mathcal{T}$ of size n and a μ-calculus formula φ of size m. Moreover, we assume that $\mathcal{T}$ has a tree decomposition of tree-width k and therefore also a nice tree decomposition $(\mathcal{Y}, T)$, where $T = (I, F), \mathcal{Y} = \{Y_i \mid i \in I\}$, of the same size. For D we take the corresponding game graph created using the translation from Sect. 2.2.

It is important to realize that a direct approach by taking a parity game graph D and using the previous algorithm to solve D does not work as intended. This is because of the fact that D can be of much higher tree-width than $\mathcal{T}$, because the graph of the formula φ can contain several loops. Actually, the following fact holds:

Fact 2. *The graph of a μ-formula φ is of tree-width at most n, where n is the number of variables in the formula (and not any greater than alternation depth). Moreover for every n there exists a formula φ with n variables such that the tree-width of the graph of φ is n.*

Instead of computing the game graph first, we can just use the algorithm for parity games on the graph of the system being checked. The only change we make is that instead of adding/removing a single vertex p, we will add/remove m vertices (p, ψ) at a time – one for every $\psi \in Sub(\varphi)$ $(m = |Sub(\varphi)|)$.

Taking $(\mathcal{Y}, T)$ and D as above, we define $X_i = \{(p, \psi) \mid p \in Y_i, \psi \in Sub(\varphi)\}$. It is easy to check that using these X_i's and D in the definition of border makes all the results in Sect. 4.1 hold. Now we must modify the algorithm a little bit. We will progress on the structure of $\mathcal{Y}$:

Introduce node A vertex v of T is being introduced. We will add vertices (v, ψ) one by one, as a sequence of vertices in the original algorithm. The order is not important here.

Start node A leaf containing only v is created. In this case we create a border with vertex (v, φ) and then add vertices $(v, \psi), \psi \in Sub(\varphi)$ one by one, as in the original introduce node algorithm.

Forget node Vertex v of T is being removed. We remove the vertices (v, ψ) one by one as a sequence of vertices in the original algorithm.

Join node The only different bit is checking whether two borders can correspond to some common strategy. The choices made in the vertices of type $(v, \langle a \rangle \psi)$ are checked by the original algorithm, as the edge goes to some vertex (w, ψ), where $v \neq w$. The choices made for vertices $(v, \psi_1 \vee \psi_2)$ can be checked easily, as there is an edge to either (v, ψ_1) or (v, ψ_2) in both the borders.

5.1 Complexity

Theorem 4 (complexity). *Let $\mathcal{T}$ be a LTS of size n with a tree decomposition of tree-width k, and φ a formula of size m. Then the previous algorithm runs in time $\mathcal{O}(n \cdot (km)^2 d^{2((k+1)m)^2})$.*

Proof. We start with the complexity estimate for parity games. In the μ-calculus case, the size of borders has grown from $k+1$ to $(k+1) \cdot m$. However, we do not increase the number of nodes in tree-decomposition. The number of priorities $\mathbf{p}$ is bounded by m (actually, we can bound it by d, the alternation depth of formula). The rest follows from Theorem 3. □

Comparing to the result of [LBC+94], our algorithm is *linear* in the size of the system, no matter what the formula is. It should be also noted, that the estimated running time is really the upper bound and the algorithm may benefit from further optimisation.

5.2 Application to Software Model Checking

The algorithm presented above looks suitable for model checking software programs. For structured programs have a low tree-width and, moreover, we can find the tree decomposition just by performing a simple syntactic analysis [Tho98]. In practice it is usually the case that the size of the system itself is huge, whereas the formula is quite small. This is where the fact that our algorithm is linear in the size of the system may give better results compared to previous algorithms.

6 Conclusions and Future Work

We have shown that parity games can be solved in polynomial time for the important class of systems of bounded tree-width. This result was then used to present μ-calculus model checking algorithm which is linear in the size of the system. We hope that these results can bring another insight into what is the exact complexity of solving parity games and μ-calculus model checking. In addition software model checking may benefit from this work, since control flow graphs of structured programs have bounded tree-width.
Following the approach presented here, there is a hope to obtain even better results. For example, our algorithm is not optimal on directed acyclic graphs (DAGs). Even though μ-calculus model checking (or solving parity games) on DAGs can be done in linear time, DAGs can have a high tree-width. This is because we decompose the underlying undirected graph, and therefore do not take into account some knowledge we have about the system. Finding some right structural property of directed graphs might prove useful.

References

[Bod97] Hans L. Bodlaender. Treewidth: Algorithmic techniques and results. In *MFCS'97*, volume 1295 of *LNCS*, pages 19–36, 1997.

[Cou90] B. Courcelle. Graph rewriting: An algebraic and logic approach. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 193–242. Elsevier, Amsterdam, 1990.

[EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 5th IEEE Foundations of Computer Science*, pages 368–377, 1991.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of mu-calculus. In *CAV 93*, volume 697 of *LNCS*, pages 385–396. Springer-Verlag, 1993.

[EL86] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional μ–calculus. In *Symposion on Logic in Computer Science*, pages 267–278. IEEE Computer Society Press, June 1986.

[FG02] M. Frick and M. Grohe. The complexity of first-order and monadic second-order logic revisited. In *LICS'02*, pages 215–224. IEEE Computer Society, 2002.

[GMT02] J. Gustedt, O. Mæhle, and J. A. Telle. The treewidth of Java programs. In *Proceedings of ALENEX'02*, volume 2409 of *LNCS*. Springer-Verlag, 2002.

[Jur00] M. Jurdziński. Small progress measures for solving parity games. In *STACS 2000*, volume 1770 of *LNCS*, pages 290–301. Springer-Verlag, 2000.

[Klo94] T. Kloks. *Treewidth – computations and approximations*, volume 842 of *LNCS*. Springer-Verlag, 1994.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[LBC+94] D. E. Long, A. Browne, E. M. Clarke, S. Jha, and W. R. Marrero. An improved algorithm for the evaluation of fixpoint expressions. In *CAV '94*, volume 818 of *LNCS*, pages 338–350. Springer-Verlag, 1994.

[Mad97] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. Edition versal 8, Bertz Verlag, Berlin, 1997.

[RS84] N. Robertson and P. D. Seymour. Graph Minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36:49–63, 1984.

[Sti95] C. Stirling. Local model checking games. In *CONCUR '95*, volume 962 of *LNCS*, pages 1–11. Springer-Verlag, 1995.

[Sti01] Colin Stirling. *Modal and Temporal Properties of Processes*. Texts in Computer Science. Springer-Verlag, 2001.

[Tho98] M. Thorup. All structured programs have small tree-width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[VJ00] J. Vöge and M. Jurdziński. A discrete strategy improvement algorithm for solving parity games. In *CAV 2000*, volume 1855 of *LNCS*, pages 202–215. Springer-Verlag, 2000.

Appendix: Algorithms from Sect. 4.2

Forget node

Function `Forget(`j,v`)`

$B(i) = \emptyset$;
for *all* $b \in B(j)$ **do**
  $b' = b \setminus b(v,*), b(*,v)$;
  if $b(v) == \perp \| \ b(v,v) == p,\ p$ *odd* **then**
    for *all* $u \in X_i$ *s.t.* $b(u,v)\ != 0$ **do**
      $b'(u) = \perp$
    $B(i) = B(i) \cup \{b'\}$;
  else
    for *all* s,t *s.t.* $b(s,v) == p_1$ *and* $b(v,t) == p_2$ **do**
      $p = \max(p_1, p_2)$;
      if $b(s,t)! = 0$ **then**
        $b'(s,t) = \overline{\mathcal{R}}(p, b(s,t))$;
      else
        $b'(s,t) = p$;
  $B(i) = B(i) \cup \{b'\}$;

Introduce node

Function `Introduce`(j,v)

$B(i) = \emptyset$;
for $b \in B(j)$ **do**
  // add edges $X_i \cap V_1 \rightarrow v$;
  for $u \in (X_i \cap V_1), (u,v) \in E$ **do** $b =$ `add` (b,u,v);

  // add edges $v \rightarrow X_i$;
  if $v \in V_1$ **then**
    for $w \in X_i, (v,w) \in E$ **do** b'=`add` (b,v,w);
    $S = \{b'\}$;
  else
    $S = \{b_{\{v \mapsto \circ\}}\}$;
    for $w \in X_i, (v,w) \in E$ **do**
      $b' =$ `add` (b,v,w);
      $S = S \cup \{b'\}$

  for $b' \in S$ **do**
    // all possibilities of unresolved P_0 vertices ;
    for $W \subseteq \{u \in X_i \mid (u,v) \in E \wedge b(u) == \circ\}$ **do**
      $b'' = b'$;
      for $u \in W$ **do** $b'' =$ `add` (b'',u,v);
      $B(i) = B(i) \cup \{b''\}$;
// remove borders which cannot correspond to a strategy ;
for $b \in B(i), v' \in X_i$ **do**
  if $b(v') == \circ$ *and there is no* $w \in V \setminus V_i$ *s.t.* $(v',w) \in E$ **then**
    $B(i) = B(i) \setminus \{b\}$

Procedure `add`(b,s,t)

$p = \max(p(s), p(t))$;
$b(s,t) = p$;
return b;

Join node

As mentioned in **Note on joining** (Sect. 4.2), we must remember which edge was selected for every vertex in $V_0 \cap X_i$. In the algorithm we use notation $\tilde{b}(v)$, where $\tilde{b}(v) = w$ means the edge (v, w) $(w \in X_i)$ was selected, and $\tilde{b}(v) = 0$ when w is in $V_i \setminus X_i$.

Function `join`(j_1, j_2)

```
    B(i) = ∅;
out for all b1 ∈ B(j1), b2 ∈ B(j2) do
        new b;
        for all s ∈ Xi do
            if s ∈ V1 then
                switch (b1(s), b2(s)) do
                    case (⊥,): b(s) = ⊥; break ;
                    case (,⊥): b(s) = ⊥; break ;
                    case (S, S'): joinPair (b, b1, b2, s);
            else
                switch (b1(s), b2(s)) do
                    case (⊥, ⊥): ;
                    case (S, ⊥): ;
                    case (⊥, S): break out;
                    case (○, ○): b(s) = ○; break ;
                    case (○, ⊥): b(s) = ⊥; break ;
                    case (⊥, ○): b(s) = ⊥; break ;
                    case (S, ○):
                        if b̃1(s) == 0 then
                            b(s) = S ; break ;
                        else
                            break out;
                    case (○, S):
                        if b̃2(s) == 0 then
                            b(s) = S ; break ;
                        else
                            break out;
                    case (S, S'):
                        if b̃1(s) = b̃2(s) then
                            break out;
                        else
                            case (S, S'): joinPair (b, b1, b2, s);
        B(i) = B(i) ∪ {b}
    //remove borders which cannot correspond to a strategy ;
    for b ∈ B(i), v ∈ Xi do
        if b(v) == ○ and there is no w ∈ V ∖ Vi s.t. (v, w) ∈ E then
            B(i) = B(i) ∖ {b}
```

Procedure `joinPair`(b, b_1, b_2, s)

```
if b̃_1(s) = 0 then return ;
else for all t ∈ X_i do
    p_1 = b_1(s, t);
    p_2 = b_2(s, t);
    b(s, t) = R̄(p_1, p_2);
```