Decidable classes of documents for XPath*

Vince Bárány[1,2], Mikołaj Bojańczyk[2], Diego Figueira[2,3], and Paweł Parys[2]

1 **TU Darmstadt, Darmstadt, Germany**
2 **University of Warsaw, Warsaw, Poland**
3 **University of Edinburgh, Edinburgh, United Kingdom**

Abstract

We study the satisfiability problem for XPath over XML documents of bounded depth. We define two parameters, called *match width* and *braid width*, that assign a number to any class of documents. We show that for all k, satisfiability for XPath restricted to bounded depth documents with match width at most k is decidable; and that XPath is undecidable on any class of documents with unbounded braid width. We conjecture that these two parameters are equivalent, in the sense that a class of documents has bounded match width iff it has bounded braid width.

1998 ACM Subject Classification F.1.1 Models of Computation, F.4.1 Mathematical Logic, F.4.3 Formal Languages, H.2.3 Languages

Keywords and phrases XPath, XML, class automata, data trees, data words, satisfiability

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2012.99

1 Introduction

This paper is about satisfiability of XPath over XML documents, modelled as data trees. A data tree is a tree where every position carries a *label* from a finite set, and a *data value* from an infinite set. The data values can only be tested for equality.

XPath satisfiability. XPath can be seen as a logic for expressing properties of data trees. Here are some examples of properties of data trees that can be expressed in XPath: "every two positions carry a different data value", "if x and y are positions that carry the same data value, then on the path from x to y there is at most one position that has label b". Our interest in XPath stems from the fact that it is arguably the most widely used XML query language. It is implemented in XSLT and XQuery and it is used in many specification and update languages. Query containment and query equivalence are important static analysis problems, which are useful to query optimization tasks. These problems reduce to checking for *satisfiability*: Is there a document on which a given XPath query has a non-empty result? By answering this question we can decide at compile time whether the query contains a contradiction and thus the computation of the query (or subquery) on the document can be avoided. Or, by answering the query equivalence problem, one can test if a query can be safely replaced by another one which is more optimized in some sense (e.g., in the use of some resource). Moreover, the satisfiability problem is crucial for applications on security, type checking transformations, and consistency of XML specifications.

Our point of departure is that XPath satisfiability is an undecidable problem [9]. There are two main approaches of working around this undecidability.

* Partially supported by FET-Open Project FoX, grant agreement 233599.


© V. Bárány, M. Bojańczyk, D. Figueira, and P. Parys;
licensed under Creative Commons License NC-ND
32nd Int'l Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012).
Editors: D. D'Souza, J. Radhakrishnan, and K. Telikepalli; pp. 99–111

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1. **Restrict the formulas.** The first way is to consider fragments of XPath that have decidable satisfiability. For example, fragments without negation or without recursive axes [1]; or fragments whose only navigation can be done downwards [7] or downwards and rightwards [6] or downwards and upwards [8]. However, even though all these restrictions yield decidable fragments, the most expressive ones have huge, non-primitive-recursive, complexity bounds.
2. **Restrict the models.** When proving undecidability of XPath satisfiability, for each Minsky machine one constructs an XPath formula φ, such that models of φ describe computations of the Minsky machine. XML documents that describe computations of Minsky machines seem unlikely to appear in the real world; and therefore it sounds reasonable to place some restrictions on data trees, restrictions that are satisfied by normal XML documents, but violated by descriptions of Minsky machines.

This paper is devoted to the second approach.

Comparison with tree width

The archetype for our research is the connection between tree width and satisfiability of guarded second order logic, over graphs. Guarded second-order logic is a logic for expressing properties of undirected graphs. A formula of guarded second-order logic uses a predicate $E(x, y)$ for the edge relation, and can quantify over nodes of the graph, sets of nodes of the graph, and subsets of the edges in the graph. Satisfiability of guarded second-order logic over graphs is undecidable (already first-order logic has undecidable satisfiability). However, the picture changes when one bounds the tree width of graphs.

- Bounded tree width is a sufficient condition for decidability. More precisely, for every $k \in \mathbb{N}$, one can decide if a formula of guarded second-order logic is satisfied in a graph of tree width at most k [5].
- Bounded tree width is also a necessary condition for decidability: if a set of graphs X has unbounded tree width, then it is undecidable if a given formula of guarded second-order logic has a model in X [12].

Our contribution

Our goal in this paper is to find a parameter, which is to XPath over data trees, what tree width is to guarded second-order logic over graphs. As candidates, we define two parameters, called the *match width* and the *braid width* of data trees. Our results are:

- Bounded match width is a sufficient condition for decidability. For every $k \in \mathbb{N}$, one can decide if a formula of XPath is satisfied in a data tree of match width at most k.
- Bounded braid width is a necessary condition for decidability: if a set of data trees X has unbounded braid width, then it is undecidable if a given formula of XPath has a model in X.

Observe that the two statements talk about two different parameters. We conjecture that bounded match width is equivalent to bounded braid width, but we are unable to prove this.

Bounded depth data trees. Our results are restricted to data trees with bounded depth, which we believe is relevant since the depth of XML documents is very small in practice. However, we believe that our results can be extended to arbitrary data trees.

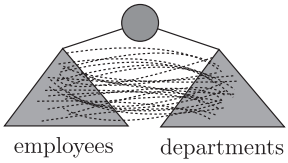
Why not tree width?

Instead of defining a new parameter for data trees, such as match width, why not simply work with tree width, which has proved so useful in other contexts? It is not difficult to

apply the notion of tree width to a data tree; for instance one can define the tree width of a data tree to be the tree width of the graph where the nodes are positions, and the edges are either tree edges (connecting a node to its parent in the tree), or data edges (connecting positions carrying the same data value). In fact, even guarded second-order logic (which is much more expressive than XPath) is decidable over data trees of bounded tree width.

So why not use tree width? A short answer is that because guarded second-order logic is much more powerful than XPath, one can define data trees which are difficult for guarded second-order logic (have high tree width), but easy for XPath (have low match width).

Another, related, point is that match width is a parameter that is more suited to analyzing data trees, with an emphasis on XML documents that store databases. A data tree contains two kinds of structure: the underlying tree structure, and the structure induced by comparing data values. When talking about the tree width of a data tree, these two kinds of structure are not distinguished, and merged into a single graph. Match width, on the other hand, is sensitive to the difference between the tree structure and the data structure. Consider the following example. Suppose that we have a data tree, which contains two subtrees: one subtree which contains employees, and one subtree which contains departments. Suppose that in each employee record, we have a pointer to a department record, which maps an employee to his/her department. Such a document is illustrated below, with the dashed lines representing links from employees to their departments.



As long as the sources of all the links (in the "employees" subtree) are located in a different subtree than the the targets of the links (in the "department" subtree), then the match width of the document will be small (as we shall prove in the paper), regardless of the distribution of the links. On the other hand, by appropriately choosing the distribution of the links (while still keeping all sources in the "employees" subtree and all targets in the "departments" subtree), the tree width of a document can be made arbitrarily high. In other words, the documents like the one in the picture are a class of data trees, where the match width is bounded, and the tree width is unbounded. (We also believe that this class of documents is rather typical for XML documents occurring in practice.) In particular, on this class of documents, guarded second-order logic is undecidable, while XPath is decidable.

Plan of the paper

In the next section we state the main definitions: data trees, data words, and class automata. Our main contributions are stated in terms of data words with at most two elements in each equivalence class, called *match words*, since it simplifies proofs and definitions. However, in later sections we show how to generalize our results to bounded depth data trees with unbounded number of elements per data equivalence class. We introduce *match width* in Section 3, and in Section 4 we show our main result, that class automata (or XPath) over classes of match words with bounded match width is decidable. In Section 5 we introduce *braid width* and show that XPath and class automata are undecidable over any class with unbounded braid width. In Sections 6 and 7 we extend the definitions and results to data words and to bounded depth data trees.

2 Preliminaries

Data trees and data words

We work with unranked trees, where the siblings are ordered. A tree over alphabet A is a tree where the nodes are labelled by letters from A. A *data tree* can be defined in two ways. The first way is that it is an unranked tree, where every node carries a label from the input alphabet A, and a data value from a infinite set (say, the natural numbers). The logics that we use can only compare the data values for equality, and therefore the only thing that needs to be known about data values in a data tree is which ones are equal. That is why we choose to model a data tree as a pair $(t, \sim)$, where t is a tree over the alphabet A, and $\sim$ is an equivalence relation on the nodes of t, which says which nodes have the same data value. Similarly, a *data word* is a pair $(w, \sim)$, where w is a word over the alphabet A and $\sim$ is an equivalence relation on the positions of w.

XPath

By XPath we mean the fragment capturing the navigational aspects of XPath 1.0. (called FOXPath in [2]). Expressions of this logic can navigate the tree by composing binary relations from a set of basic relations (a.k.a. *axes*): the parent relation (here noted $\uparrow$), child ($\downarrow$), ancestor ($\uparrow^*$), descendant ($\downarrow_*$), next sibling to the right ($\rightarrow$) or to the left ($\leftarrow$), and their transitive closures ($\rightarrow^*$, ${}^*\leftarrow$). For example, $\alpha = \uparrow[a]\uparrow\downarrow[b]$, defines the relation α between two nodes x, y such that y is an uncle of x labeled b and the parent of x is labeled a. Boolean tests are built by using these relations. An expression like $\langle\alpha\rangle$ (for a relation α) tests that there exists a node accessible with the relation α from the current node. Most importantly, a data test like $\langle\alpha = \beta\rangle$ (resp. $\langle\alpha \neq \beta\rangle$) tests that there are two nodes reachable from the current node with the relations α and β that have the same (resp. different) data value. A formal definition of the considered fragment of XPath can be found in [2]. XPath can be also interpreted over data words, using the sibling axes.

Class automata

In the technical parts of the paper, we will use an automaton model for XPath, which is called *class automata*, introduced in [4]. Suppose that t is a tree over a finite alphabet A, and X is a set of nodes in t. We write $t \otimes X$ for the tree over alphabet $A \times 2$ (this is the same as $A \times \{0, 1\}$) obtained from t by extending the label of each node by a bit, which indicates whether the node belongs to X. Similarly, for any two trees t_1, t_2 with the same nodes and labelled by alphabets A_1 and A_2 respectively, we write $t_1 \otimes t_2$ for the tree over alphabet $A_1 \times A_2$, which has the same nodes as t_1 and t_2, and where each node is labelled by the pair of labels from t_1 and t_2.

A *class automaton* consists of

- an input alphabet A;
- a work alphabet B;
- a class condition L, which is a regular language of trees over the alphabet $A \times B \times 2$.

The automaton accepts a data tree $(t, \sim)$ if there is some tree s over the work alphabet B with the same nodes as t, such that $t \otimes s \otimes X \in L$ for every equivalence class X of $\sim$.

It was shown in [4] that class automata capture XPath.

▶ **Theorem 1** ([4]). *For every XPath boolean query, one can compute a class automaton, which accepts the same data trees.*

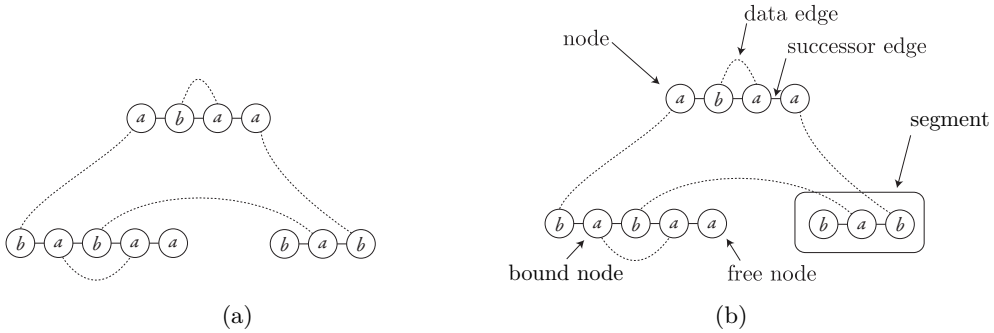


Figure 1 A split match word (a) and its parts (b).

3 Match width

In this section we define the match width of a data word. In fact, we will work with data words whose every class has size at most 2, that we call *match words*. Later on, we will comment on how to extend this definition to data words and data trees.

A *split match word* is a finite set of words, together with an additional set of edges, which connect positions of these words with each other. Figure 1 (a) contains an example of a split match word. We write τ for split match words. The words are called the *segments* of the split word.

We can think of a split word as a graph where the nodes are positions of the words, with two types of edges: *successor edges*, which go from one position to the next in each segment, and *data edges*, which are the additional edges. We require the data edges to be a matching. The matching need not be perfect: some nodes might not have an outgoing data edge, such nodes are called *free nodes*. The nodes that are not free are called *bound nodes*. These definitions are illustrated in Figure 1 (b). Note that a match word can be seen as a special case of a split match word, which has one segment.

3.1 Operations on split match words

We will construct split match words out of simpler split match words, using the following four operations.

- **Base** (no arguments). We can begin with a split match word which has one segment, with one position, and no data edges.
- **Union** (two arguments). We can take a disjoint union of two split match words. In the resulting split word, the number of segments is the sum of the numbers of segments of the arguments; and there are no data edges between segments from different arguments.
- **Join** (one argument). We can add a successor edge, joining two segments into one segment.
- **Match** (one argument). We can choose two distinct segments, and add any set of data edges with sources in one segment and targets in the other segment.

The operations (except union) are not deterministic (in the sense that the result is not uniquely determined by the operation name and the argument). The result of base depends on the choice of a label on the only position; the result of applying join to a split match word also depends on the choice of segments to be joined; and the result of applying match depends on the choice of new data edges.

Derivation. A *derivation* is a finite tree, where each node is of one of four types (base, union, join or match) and is labelled by a split match word, satisfying the following natural

property. The leaves are of base type, nodes of union type have two children, and nodes of join and match type have one child. Suppose that x is a node in a derivation. Then the split match word in the label of node x is constructed, using the operation in the type of x, from the split match words in the children of x. A derivation is said to *generate* the split match word that labels its root.

Rank. Suppose that t is a derivation. To each node of t, we assign a number, which is called the *rank* of the node. The rank of a base node is 0. The rank of a union or join node is the maximum of the ranks of the children. The rank of a match node is 0 if the split match word in the label of x has no free nodes; otherwise it is $n+1$, where n is the rank of the unique child of x.

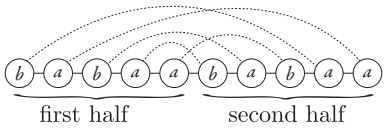
Width. The *rank* of a derivation is the maximal rank that appears in a node of the derivation. The *segment size* of a derivation is the maximal number of segments that appear in a split match word in one of the nodes of the derivation. Finally, the *width* of a derivation is the maximum of its rank and segment size. The *match width* of a split match word is the minimal width of a derivation that generates it. We are most interested in the special case of the match width of a match word, seen as a special case of a split match word.

In order to have small width, a derivation needs to have both small rank and small segment size. In the following examples we show that requiring only the rank to be small, or only the segment size to be small, is not restrictive enough, since all match words can be generated that way.

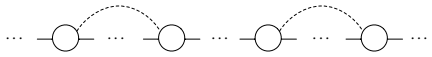
▶ **Example 2.** Every split match word with one segment (which is the same thing as a match word, with possibly some free nodes that are not matched) can be generated by a derivation of segment size 2, but possibly unbounded rank. The proof is by induction on the number of nodes. Consider then a split match word with $n+1$ nodes. Apply the induction assumption to the first n nodes. Now add the $(n+1)$-st node as a separate segment using a union rule, connect it if necessary with one of the first n nodes using a match rule, and then join it using the join rule.

▶ **Example 3.** Every match word can be generated by a derivation of rank 1, but possibly unbounded segment size. We prove the following claim: every split match word τ without free nodes can be generated by a derivation of rank 1, but possibly unbounded segment size. The proof is by induction on the number of nodes in τ. For the induction step, consider a split match word τ. Choose some data edge in τ, which connects nodes x and y. Create a new split match word, call it σ, by removing the nodes x and y together with their incident successor edges and the data edge connecting x and y. (Observe that σ can have more segments than τ; for example if x and y are in separate segments, and they are not the first or last positions in those segments, then σ will have two more segments than τ, since the segments containing x and y will break into two segments each.) The new split match word σ has no free nodes; and therefore by induction assumption it has a derivation of rank 1. Since there are no free nodes, the root of the derivation has rank 0. Therefore, we can add x and y as separate segments, connect them with an edge, and then join them to σ creating τ. Using this claim, we can add free nodes and join all segments, creating any match word.

▶ **Example 4.** A segregated match word is a data word of even length $2n$, such that every data edge connects a node from the first half with a node in the second half.



(a) A segregated word.



(b) The forbidden pattern.

Clearly, segregated words have match width at most 2. They can be alternatively described as those data words not containing four elements in the configuration shown under (b).

In [4] it was shown that emptiness for class automata is decidable over segregated data words. It is instructive to recall the argument in anticipation of the broader technique underlying our main result. For the rest of this argument we say that a match word is *free* if no data value occurs twice in it, i.e. if it is entirely devoid of data equality edges and thus, as long as it is considered in isolation, it is hardly more than a word over a finite alphabet. It is easy to see that it is sufficient to just consider a fixed, one-letter work alphabet $B = \{b\}$. Let $\mathcal{A}$ be a class automaton with a class condition $L \subseteq (A \times B \times 2)^*$ recognized by a monoid morphism $\alpha : (A \times B \times 2)^* \to M$ so that $L = \alpha^{-1}(F)$ for some $F \subseteq M$. We define the *α-profile*, $\pi_\alpha(w)$, of a free match word w as the vector $X \in \mathbb{N}^M$ whose every entry X_m equals the number of positions x in w such that $\alpha(w \otimes b^{|w|} \otimes \{x\}) = m$. Observe that the set $\Lambda_\alpha = \{\pi_\alpha(w) \mid w \text{ free match word}\}$ of all profiles of free match words is the Parikh image of a suitable regular language, whence, by Parikh's theorem [11], it is semi-linear (Presburger definable [10]). Indeed, for each free match word w let $\sigma(w)$ be the word of the same length satisfying $\sigma(w)[i] = \alpha(w \otimes b^{|w|} \otimes \{i\})$ for each $1 \leq i \leq |w|$. Then $\pi_\alpha(w)$ is precisely the image of $\sigma(w)$ under the Parikh mapping. To see that the set $\{\sigma(w) \mid w \text{ a free match word}\}$ is regular it suffices to say that $\{w \times \sigma(w) \mid w \text{ a free match word}\}$ is recognizable by an automaton that aggregates in a straightforward manner the value of $\alpha(v \otimes b^{|v|} \otimes \{0\})$ for each prefix and each suffix v of w.

Whenever free match words u and v have the same length and share the same data values, then the segregated word $w = uv$ is accepted by the class automaton $\mathcal{A}$ if, and only if, there is a matrix $X \in \mathbb{N}^{M \times M}$ such that for all $r, s \in M$ $X_{r,s} \neq 0$ only if $r \cdot s \in F$ and $\sum_s X_{r,s} = Y_r$ and $\sum_r X_{r,s} = Z_s$, where $Y = \pi_\alpha(u)$ and $Z = \pi_\alpha(v)$. Here X represents a family of matching by prescribing how many positions of each type within u should be matched to how many positions of whichever type within v.

In conclusion, the class automaton α accepts some segregated word iff the above system of linear equations has a solution for $\{Y_r, Z_s, X_{r,s}\}_{r,s \in M}$. The set of solutions is definable in Presburger arithmetic (viz. semilinear [10]) and can be effectively verified for emptiness.

Main result

We now announce the main results of this paper: emptiness for class automata is decidable over match words of bounded match width.

▶ **Theorem 5.** *The following problem is decidable:*
Input *A number n and a class automaton.*
Question *Does the automaton accept some match word of match width $\leq n$?*

In Section 4, we sketch the proof of the theorem. Before proving the theorem, we further discuss the notion of match width, and show how it is related to tree width.

Match width and first-order logic. Match width is a measure that is adapted specifically for class automata. If we consider monadic second-order logic, or even first-order logic, over match words of bounded match width, then satisfiability is undecidable.

▶ **Lemma 6.** *Satisfiability of first-order logic is undecidable over segregated match words, as defined in Example 4, which have match width at most 2.*

Match width and tree width. On the other hand, bounded tree width implies bounded match width, but the converse does not hold.

▶ **Lemma 7.** *If a match word has tree width k, then it has match width at most $3k + 3$.*

▶ **Lemma 8.** *Match words of match width 2 can have unbounded tree width.*

4 Bounded match width sufficient for decidability

In this section, we present a proof sketch of Theorem 5. In fact, we show a stronger result, Theorem 9, which implies Theorem 5.

Numbered segments. Consider a split match word τ with n segments. In this section, it will be convenient to number the segments, so we assume that each split match word comes with an implicit ordering of the segments. The segments will be written as $\tau[1], \ldots, \tau[n]$. We write $\text{segments}(\tau)$ for the set $\{1, \ldots, n\}$.

Type of a split match word.

Consider a monoid morphism $\alpha : (A \times 2)^* \to M$. We define the *$\alpha$-type of a split match word* τ to be the following information.

- The *empty type*, which is the vector $\text{emptytype}_\tau \in M^{\text{segments}(\tau)}$ that maps $i \in \text{segments}(\tau)$ to the type $\alpha(\tau[i] \otimes \emptyset) \in M$.
- The *free type*, which is the function $\text{freetype}_\tau : \text{segments}(\tau) \times M \to \mathbb{N}$ that maps (i, m) to the number of free positions x in the word $\tau[i]$ that satisfy $\alpha(\tau[i] \otimes \{x\}) = m$.
- The *bound type*, which is the set $\text{boundtype}_\tau \subseteq M^{\text{segments}(\tau)}$ which contains a vector $v \in M^{\text{segments}(\tau)}$ if there is some matched pair of positions, call them x, y, such that on coordinate $i \in \text{segments}(\tau)$, the vector has the value $\alpha(\tau[i] \otimes \{x, y\})$. Note that it may be that positions x, y are not in component i, or that only one is in the component.

When the morphism α is clear from the context, we say type instead of α-type. The type is therefore some finite information (the empty type and the bound type), together with a vector of natural numbers (the free type). Because of the free type, the set of possible α-types is potentially infinite. However, as we shall see below, semilinear sets can be used to represent the vectors in the free type, at least as long as the match width is bounded. We briefly recall semilinear sets below.

Semilinear sets

A *linear space* is $\mathbb{N}^k$ for some dimension $k \in \mathbb{N}$. A *semilinear space* is a finite disjoint union of linear spaces. The components of the disjoint union are called the *components* of the semilinear space. We write semilinear spaces as $\coprod_{i \in I} X_i$, where the index set I is finite, each component X_i is a linear space, and $\coprod$ stands for disjoint union. Semilinear spaces are closed under Cartesian products and disjoint unions.

A *semilinear subset* of a linear space is defined in the usual way. That is, as a finite union of linear subsets, where the linear subsets of $\mathbb{N}^k$ are those of the following form: $\mathbf{v_0} + \sum_{j=1}^{t} \mathbb{N}\mathbf{v_j} = \{\mathbf{v_0} + \sum_{j=1}^{t} n_j \mathbf{v_j} : n_1, \ldots, n_t \in \mathbb{N}\}$, for some t and fixed $\mathbf{v_0}, \mathbf{v_1}, \ldots, \mathbf{v_t} \in \mathbb{N}^k$. A *semilinear subset* of a semilinear space associates a semilinear subset to each of the components.

Representing α-types.

We are now ready to state the main technical result used in the proof of Theorem 5. Fix a number k of segments. An α-type of a split match word with k segments is an element of

$$X_k \stackrel{\text{def}}{=} \underbrace{M^k}_{\text{empty type}} \times \underbrace{\mathbb{N}^{k \times M}}_{\text{free type}} \times \underbrace{P(M^k)}_{\text{bound type}} .$$

The above is a semilinear space, with one linear space of dimension $k \times M$ for every pair of empty and bound types. Therefore, it makes sense to ask if the set of possible α-types is semilinear or not. The answer is positive, when the match width is bounded, as the following theorem shows.

▶ **Theorem 9.** *The set* $\{\text{type}_\tau : \tau \text{ is a } k\text{-segment split match word of match width} \leq n\}$, *where* $n \in \mathbb{N}$, *is a semilinear subset of* X_k *and can be computed.*

The proof of the theorem uses Parikh's theorem, which says that the Parikh image of a context-free language is semilinear. Here we only show that Theorem 9 implies Theorem 5.

Proof of Theorem 5. Recall that the theorem says that one can decide if a class automaton accepts some match word of match width $\leq k$. Let A be the input alphabet, B be the work alphabet, and $L \subseteq (A \times B \times 2)^*$ the class condition of the class automaton. Let $\alpha : (A \times B \times 2)^* \to M$ be a monoid morphism which recognizes the language L. In other words there is some set $F \subseteq M$ such that L is the inverse image $\alpha^{-1}(F)$.

The match width of a word does not change after its positions have been additionally labelled by the work alphabet B. Therefore, we want to decide the following question: is there some match word $(w, \sim)$ over the alphabet $A \times B$, with match width $\leq k$, and so that

- $w \otimes \{x, y\} \in L$ for every matched positions $\{x, y\}$ in $\sim$, and
- $w \otimes \{x\} \in L$ for every free position $\{x\}$ in $\sim$.

The above condition says that there is a split match word with one segment such that the free type maps every pair (i, m) with $m \notin F$ to 0 (i.e. all free positions x are so that $w \otimes \{x\} \in L$), and the bound type is a subset of F. By Theorem 9, we can compute all possible α-types of match words with one segment, and test if there is some such α-type. ◀

5 A necessary condition for decidability

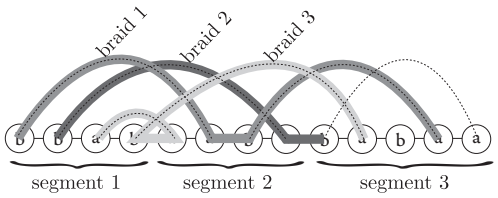
So far, we have defined a measure of match words, namely match width. Bounded match width is a sufficient condition for decidable emptiness of class automata. In this section, we define a complementary measure, called *braid width*, such that emptiness of class automata is undecidable on any class of match words with unbounded braid width. Therefore, having bounded braid width is a necessary condition for decidable emptiness of class automata.

Definition of braid width

Consider a match word, interpreted as a graph with successor edges and data edges. Split the match word into n consecutive subwords, henceforth called *segments*. Consider a path, which uses successor and data edges, and visits nodes $x_1, \ldots, x_k$. Such a path is a *braid* if:

- the path visits all n segments, and
- if a node of the path is in segment i, then subsequent nodes on the path cannot revisit any of the segments $1, \ldots, i-2$.

(The notion of a braid is relative to some decomposition into segments.) We say that a match word has *braid width* at least n if it can be split into n segments, such that one can find n node-disjoint braids. Below there is a picture of a match word with braid width at least 3, along with the witnessing segments and braids.



Notice that a braid can visit the previous segment. In fact, this is necessary, as there are classes of match word with unbounded braid width that, if we would restrict the braids to go only forward, would otherwise have bounded braid width. Also, for any fixed $k \in \mathbb{N}$ one can construct a class automaton $\mathcal{A}_k$ recognizing all match words of braid width at least k.

▶ **Theorem 10.** *Let X be a class of match words of unbounded braid width. Then, emptiness of class automata is undecidable on X. If in addition X is closed under arbitrary relabellings, then even XPath is undecidable on X.*

Proof idea. Observe that the first claim follows from the second, since class automata capture XPath by Theorem 1. The second claim is proved by reduction from the halting problem for 2-counter Minsky machines. To every Minsky machine $\mathcal{M}$ we associate a boolean XPath query whose models are match words that code accepting runs of $\mathcal{M}$. It is not hard to see that, for any k, there must be a word in X with k braids and k segments such that:

1. every braid begins in the first segment, and finishes in the last segment, and
2. consecutive nodes on every braid are either in the same or in neighboring segments.

Such a match word with an appropriate labelling will represent a run of $\mathcal{M}$ of length k so that $n_1 + n_2 \leq k$, where n_i is the maximum value of counter i reached during the run.

We use labels to mark which are the nodes involved in the k braids, and for every such node in a braid, where to find the previous and next node in the traversal of the braid (it could be: the node to the left or to the right, or the node in the same equivalence class). Also, labels mark the beginning and end of each segment; and for every segment and braid, we add a special label to flag the first node of the segment in the braid traversal.

An XPath formula can verify that the labels correctly code braids with the properties 1 and 2. Now, making use of the fact that there are exactly k flagged nodes in every segment, and that there is a way of linking these k nodes in a segment with the k flagged nodes of the next segment, we can code configurations of $\mathcal{M}$. In order to code the counters, each braid is associated to one of the counters through a label, and each flagged element is labelled with being active or inactive. The value of counter $i \in \{1, 2\}$ is then the number of active flagged nodes of braids associated to counter i inside the segment. If we further label each segment with a state, each segment codes a configuration of $\mathcal{M}$. Depending on the instruction, an XPath formula can test whether a counter is 0: no active flagged elements in the segment. Further, a formula can make a counter to increase (or decrease) its value: we choose an inactive flagged node in the segment and we force that the next flagged node in the braid traversal is active, while preserving the active/inactive state of all remaining flagged nodes.

Each of the above properties is expressible in XPath and we can therefore ensure that an accepting run of $\mathcal{M}$ is encoded in the match word. ◀

Figure 2 An example of a data expansion $(w, \sim)$ of a data word $(w', \sim')$.

6 Data words instead of match words

We discuss how the definition of braid and match width can be generalized to classes of data words with arbitrarily many elements in each equivalence class. The same decidability and undecidability results can also be transfered to this general definition.

Suppose a data word $(w, \sim)$ with two data classes X, Y with at least two elements each, so that the rightmost position of X is x and the leftmost position of Y is $x+1$ (i.e., the next position of x). We say that $(w, \sim)$ is a *data expansion* of $(w', \sim')$, if $(w', \sim')$ is the result of removing $x + 1$ from $(w, \sim)$ and joining the data classes X and Y (cf. Figure 2). We say that a match word $(w, \sim)$ is a *match expansion* of $(w', \sim')$ if it is the result from applying repeatedly data expansions to $(w', \sim')$. The class of match words C is the match expansion of a class of data words C', if C consists of all match expansions of data words from C'.

We define that a class C of data words has match width at least/at most n if the match expansion of C has match width at least/at most n. Similarly, C has braid width at least/at most n if the match expansion of C has braid width at least/at most n.

These definitions allow us to transfer our decidability and undecidability results for class automata. Further, the notions of bounded match width and bounded braid width coincide on match words if and only if they coincide on data words.

▶ **Lemma 11.** *For any class of data words with unbounded braid width, the emptiness problem for class automata is undecidable.*

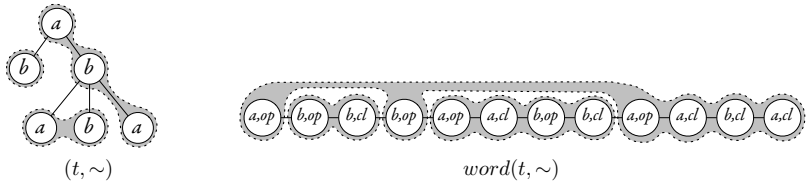
▶ **Lemma 12.** *Given a class automaton $\mathcal{A}$ and a number n, the emptiness problem for $\mathcal{A}$ restricted to data words with match width at most n is decidable.*

Note that, by Theorem 1, Lemma 12 is also true for XPath. However, we do not know if the undecidability result for XPath (or even *regular*-XPath) of Theorem 10 holds for classes of data words of unbounded braid width.

7 Data trees instead of data words

In the previous sections we discussed the case of data words. But what about data trees? Here we extend the notions of braid width and match width to data trees and generalize our decidability and undecidability results.

One solution is to reduce data trees to words. For a data tree $(t, \sim)$, we define its word representation $word(t, \sim)$, which is like the text representation of an XML tree. If the labels of t are A, then the labels of $word(t, \sim)$ are $\{open, close\} \times A$. Every node of $(t, \sim)$ corresponds to two nodes in $word(t, \sim)$, one with an opening tag and one with a closing tag, with the same data value, as depicted below.



If the depth of the tree is known in advance, and can be encoded in the states of an automaton, then this representation can be decoded by a class automaton.

▶ **Lemma 13.** *Fix $d \in \mathbb{N}$. For any class automaton on data trees $\mathcal{A}$, one can compute a class automaton on data words $\mathcal{A}_d$ such that $\mathcal{A}$ accepts a data tree $(t, \sim)$ if and only if $\mathcal{A}_d$ accepts the data word $word(t, \sim)$ provided that $(t, \sim)$ has depth at most d.*

We extend the definition of match and braid width to data trees following the $(t, \sim) \mapsto word(t, \sim)$ coding. A class of data trees has braid/match width of at least/at most n if its data word representation has braid/match width of at least/at most n. In view of the lemma above, we have the following.

▶ **Lemma 14.** *For any class automaton on data trees $\mathcal{A}$ and numbers d, n, the emptiness problem for $\mathcal{A}$ restricted to data trees of depth at most d and match width at most n is decidable.*

8 Discussion

We conjecture that match width and braid width are equivalent in the sense that a class of data words has bounded match width if, and only if, it has bounded braid width. One implication of the conjecture follows from Theorems 5 and 10. Namely, for every k, the class of documents of match width at most k has bounded braid width, since otherwise the class would have undecidable emptiness for class automata. In other words, unbounded braid width means unbounded match width. The content of the conjecture is therefore the other implication: is it the case that unbounded match width means unbounded braid width?

We also conjecture that XPath is undecidable on unbounded match width data words or data trees, but we are unable to prove it.

Finally, we believe that the results can also be extended to arbitrary data trees, by defining accordingly split data trees and using forest algebra [3] instead of monoids.

References

1 Michael Benedikt, Wenfei Fan, and Floris Geerts. XPath satisfiability in the presence of DTDs. *Journal of the ACM (JACM)*, 55(2):1–79, 2008.

2 Michael Benedikt and Christoph Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.

3 M. Bojańczyk and I. Walukiewicz. Forest algebras. In *Automata and Logic: History and Perspectives*, pages 107–132. Amsterdam University Press, 2007.

4 Mikolaj Bojanczyk and Slawomir Lasota. An extension of data automata that captures XPath. In *LICS*, pages 243–252. IEEE Computer Society, 2010.

5 Bruno Courcelle. Graph rewriting: A bibliographical guide. In *Term Rewriting*, volume 909 of *Lecture Notes in Computer Science*, page 74. Springer, 1993.

6 Diego Figueira. Alternating register automata on finite data words and trees. *Logical Methods in Computer Science (LMCS)*, 8(1:22), 2012.

7 Diego Figueira. Decidability of downward XPath. *ACM Transactions on Computational Logic (TOCL)*, 13(4), 2012. To appear.

8 Diego Figueira and Luc Segoufin. Bottom-up automata on data trees and vertical XPath. In *International Symposium on Theoretical Aspects of Computer Science (STACS'11)*. Springer, 2011.

9 Floris Geerts and Wenfei Fan. Satisfiability of XPath queries with sibling axes. In *International Symposium on Database Programming Languages (DBPL'05)*, volume 3774 of *Lecture Notes in Computer Science*, pages 122–137. Springer, 2005.

10 Seymour Ginsburg and Edwin H. Spanier. Semigroups, presburger formulas, and languages. *Pacific Journal of Mathematics*, 16:285–296, 1966.

11 Rohit J. Parikh. On context-free languages. *J. ACM*, 13:570–581, October 1966.

12 D. Seese. The structure of the models of decidable monadic theories of graphs. *Ann. Pure Appl. Logic*, 53(2), 1991.