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.

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

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.

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

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

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.

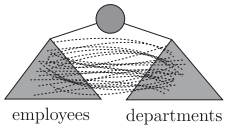
Why not tree width?

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.*

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

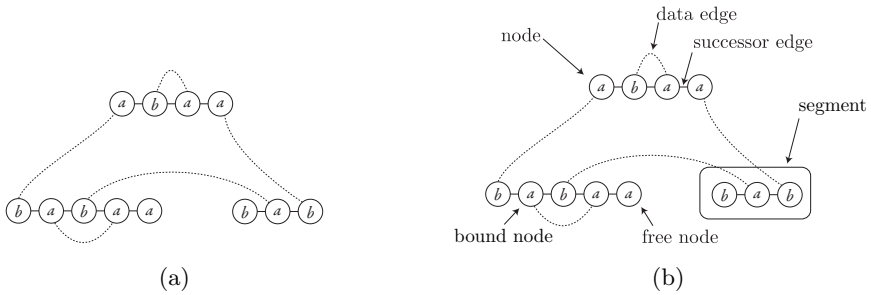


Figure 1 A split match word (a) and its parts (b).

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.

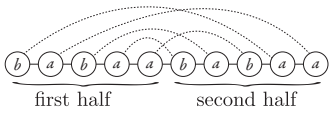
Width. The *rank* of a derivation is the maximal rank that appears in a node of the derivation. The *segment size* of a derivation is the maximal number of segments that appear in a split match word in one of the nodes of the derivation. Finally, the *width* of a derivation is the maximum of its rank 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.

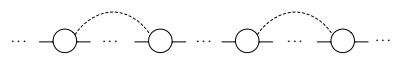
▶ **Example 2.** Every split match word with one segment (which is the same thing as a match word, with possibly some free nodes that are not matched) can be generated by a derivation of segment size 2, but possibly unbounded rank. The proof is by induction on the number of nodes. Consider then a split 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 $r, s \in M$ $\quad X_{r,s} \neq 0$ 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 segments(τ) 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* $\{\text{type}_\tau : \tau$ *is a* k*-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

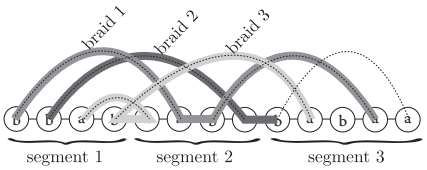
So far, we have defined a measure of match words, namely match width. Bounded match width is a sufficient condition for decidable emptiness of class automata. In this section, we define a complementary measure, called *braid width*, such that emptiness of class automata is undecidable on any class of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

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.

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

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.*

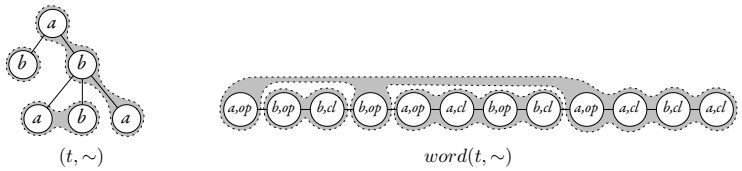
▶ **Lemma 12.** *Given a class automaton $\mathcal{A}$ and a number n, the emptiness problem for $\mathcal{A}$ restricted to data words with match width at most n is decidable.*

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.

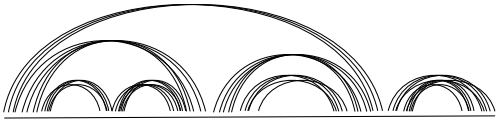
11 Rohit J. Parikh. On context-free languages. *J. ACM*, 13:570–581, October 1966.

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

A On match width

Here is a further example of a large class of match words with bounded match width.

▶ **Example 15.** Consider the class $\mathcal{P}$ of *parenthesized segregated words*. This is the smallest class of match words containing all segregated words and closed under concatenation and nesting. A 'typical' such word is illustrated below.



All parenthesized segregated match words have match width at most 3. To see that, consider split segregated words (u, v) each obtained from a segregated match word uv by splitting it in the middle ($|u| = |v|$). Just as segregated words, these too can be generated by a derivation of match width 2. To nest a parenthesized segregated word w within a segregated word uv we merely to derive w and the split segregated word (u, v) and apply *union* and *join* to obtain the match word uwv. This requires 3 components at an intermediate stage but no further uses of *match* and generates a match word $uwv \in \mathcal{P}$. Concatenations can be generated using only *union* and *join* and 2 components.

It can be verified that parenthesized segregated words are characterized as those match words not containing any of the following three configurations of 6 elements as subgraphs.

Proof of Lemma 6

Consider a match word where the sequence of labels is

$$(ab)^n w_1 \cdots w_k \qquad \text{where every } w_i \text{ is either } cd \text{ or } cdd$$

and every data edge connects a node from $(ab)^n$ to a node from $w_1 \cdots w_k$. In particular, this data word is segregated. Based on this data word w, define a graph G_w as follows (the graph will have nodes of degree at most 3). The nodes of the graph are the words $w_1, \ldots, w_k$. In the graph, there is an edge from w_i to w_j if there is some $l \in \{1, \ldots, n\}$ such that there is a data edge which connects the l-th a to some position in w_i, and there is another data edge which connects the l-th b (which is the successor of the l-th a) to some position in w_j.

It is not difficult to see that for every first-order formula on graphs φ, there is a first-order formula $\hat{\varphi}$ on match words (which has predicates for successor edges and data edges), such that

$$w \models \hat{\varphi} \qquad \text{iff} \qquad G_w \text{ is defined and } G_w \models \varphi.$$

Every graph with degree at most 3 is of the form G_w for some w. Therefore, if one wants to know of a formula φ is satisfiable over graphs of degree at most 3 (which is an undecidable problem); it suffices to test if $\hat{\varphi}$ is satisfiable in some segregated match word. Therefore, first-order logic is undecidable over segregated match words.

Proof of Lemma 7

In fact, every match word of tree width k can be generated by a derivation involving split match words of at most $3k+3$ segments and no nested applications of the *match* operation.

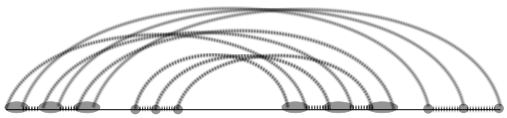
One proceeds by first generating $\frac{|w|}{2}$ many 2-segment match words, $w_{x,y}$, one for every data equality edge $\{x,y\}$ in w, by an application of *match* on the *union* of two singletons. Subsequently w is derived from all these pairs $w_{x,y}$ by fusing them together using only *union* and *join* as directed by the structure of a tree decomposition of w ensuring that all intermediate split match words comprise at most $3k+3$ segments.

Consider a tree decomposition $(T,<,\beta)$ of width k of a match word w. Here $(T,<)$ is the transitive closure of a directed tree with the $<$-minimal node as its root; and β maps every node $u \in T$ to a set of positions of w of size at most $k+1$. We may assume wlog. that every internal node u of T has precisely two successors, denoted by u^0 and u^1. By definition of tree decomposition, for each unordered pair of positions $\{x,y\}$ that share the same data value in w there is a node $v \in T$ such that $\{x,y\} \subseteq \beta(v)$. For future use we fix a $<$-maximal such $v = v_{x,y}$ for every such pair in an arbitrary fashion.

For each node u of T let $\omega(u)$ be the split match word obtained from the union of all $w_{x,y}$ such that $u \leq v_{x,y}$ by joining all successor pairs among them as in w. Naturally, $w = \omega(r)$ for r the root of T. Let us first notice that $\omega(u)$ for each u has at most $k+1$ segments. Indeed, some position x of each segment (the first or the last position) has to be later connected by a successor edge with some other position y (unless there is only segment which contains all positions of the split match word, in which case the claim holds trivially). Assume that $x \notin \beta(u)$. Then $x \in \beta(v)$ only for successors v of u, while $y \in \beta(v)$ only for nodes v not being a successor of u. This means that $\{x,y\} \not\subseteq \beta(v)$ for each node v, which contradicts the definition of a tree decomposition. Next, we claim that each $\omega(u)$ is derivable using intermediate split match words having no more than $3k+3$ segments. For u a leaf (i.e. $<$-maximal) node of T this is trivially true. Consider an internal node $u \in T$ and assume that $\omega(u^0)$ and $\omega(u^1)$ have been generated by a derivation as claimed. Observe that the sets of positions of $\omega(u^0)$ and $\omega(u^1)$ are disjoint. Then $\omega(u)$ can be derived from the *union* of $\omega(u^0)$, $\omega(u^1)$ and all those $w_{x,y}$ with $v_{x,y} = u$ (at most $3k+3$ many segments in total: at most $k+1$ from $\omega(u^0)$, from $\omega(u^1)$, and from all $w_{x,y}$) by adding all successor edges present in w using *join*s. The claim follows by bottom-up induction on T.

Proof of Lemma 8

Every $n \times m$-grid can be found as a topological minor of some segregated word. The following graphic illustrates this for 3×4. The claim follows, given that the tree-width of the $n \times n$-grid is n.



B Proof of Theorem 9: reduction to semilinear functions

Our strategy for the proof of Theorem 9 is that we abstract away the class automata and data words, and we work purely in the world of semilinear spaces and semilinear sets; in this world Theorem 16 is an analogue of Theorem 9. In this section we give all the necessary definitions, we present Theorem 16, and we show why Theorem 9 is its consequence. In the next section we prove Theorem 16.

Semilinear functions

A function which assigns a subset of a semilinear space to each element of a semilinear space

$$f\colon \coprod_{i\in I} X_i \to P(\coprod_{j\in J} Y_j)$$

can also be seen as a subset of $\coprod_{i\in I} X_i \times \coprod_{j\in J} Y_j$. If f is a semilinear subset of this semilinear space, then we say that f is a *semilinear function*. It is known that the image $\bigcup_{s\in S} f(s)$ of a semilinear subset $S \subseteq \coprod_{i\in I} X_i$ by a semilinear function f is a semilinear subset of $\coprod_{j\in J} Y_j$. Additionally, given f and S, this semilinear subset can be computed.

Move functions

A function $f : \mathbb{N}^n \to \mathbb{N}^k$ is called a *move function* if it is defined by

$$f(x_1, \ldots, x_n) = \big(\sum_{i\in A_1} x_i, \sum_{i\in A_2} x_i, \ldots, \sum_{i\in A_k} x_i \big),$$

for some choice of disjoint sets $A_1, \ldots, A_k \subseteq \{1, \ldots, n\}$. In other words, a move function is a linear function whose matrix is such that every column contains at most one entry with 1, and 0 everywhere else.

A semilinear function

$$f : \coprod_{i\in I} X_i \to P(\coprod_{j\in J} Y_j)$$

is called a move function if for each $i \in I$ either

- f restricted to arguments in X_i always returns the empty set, or
- there is some $j \in J$ such that f restricted to arguments in X_i returns always singletons, and the underlying function between semilinear spaces (returning the only element of the singleton) is a move function from X_i to Y_j.

It is a standard property of semilinear sets that when f is given by a semilinear function, one can compute its representation as a move function.

Derivation

Instead of operations on split match words we now consider semilinear functions. Namely, let S be a semilinear space, and let F be a finite set of semilinear functions of the form $f\colon S^{ar(f)} \to P(S)$ (where $ar(f)$ is the *arity* of f).

A *derivation* is a finite tree, where each node is labelled by a pair from $F \times S$, satisfying the following property. A node labelled by function f has exactly $ar(f)$ children. Suppose that x is a node in a derivation, labeled by (f, v), and $v_1, \ldots, v_{ar(f)}$ are the elements of S in the labels of the children of x. Then $v \in f(v_1, \ldots, v_{ar(f)})$. A derivation is said to *generate* the element of S that labels its root.

Level

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 every leaf is 0. Consider now an internal node x labelled by (f, v); let n be the maximum of the ranks of its children. If f is a move function, the rank of x is n. If f is not a move function, the rank of x is 0 if v has zeroes on all coordinates (recall that v is an element of some component of S, which is a linear space); otherwise it is $n + 1$. The *rank* of a derivation is the maximal rank that appears in a node of the derivation.

Main result

We now announce a theorem which is a generalization of Theorem 9.

▶ **Theorem 16.** *Let $n \in \mathbb{N}$, let S be a semilinear space, and let F be a finite set of semilinear functions of the form $f\colon S^{ar(f)} \to P(S)$. The set*

$$\{v \in S : v \text{ is generated by a derivation of rank} \leq n\}$$

is semilinear subset of S and can be computed.

Reduction from Theorem 9

As in the main paper, let X_k be the set of all possible types of words with k segments:

$$X_k = M^k \times \mathbb{N}^{k \times M} \times P(M^k).$$

Let us now fix the number $n \in \mathbb{N}$ which appears in Theorem 9, i.e. the maximal match width. As soon as this number is fixed, we can consider a semilinear space

$$S = \bigcup_{k=0}^{n} X_k.$$

The key idea of the proof is that for each of our four operations on split match words (base, union, join, match) it is enough to know types of the arguments, in order to determine the type of the result. Moreover this functions are semilinear, and for union and join they are even move functions. More precisely, we have the following three lemmas.

▶ **Lemma 17.** *There exists a move function* $\mathsf{union}\colon S \times S \to P(S)$ *such that for each two split match words σ, τ,*

- *if the total number of segments in σ and τ is $\leq n$, the type of the union of σ and τ is the only element of* $\mathsf{union}(\mathrm{type}_\sigma, \mathrm{type}_\tau)$*, and*
- *if the total number of segments in σ and τ is $> n$, the set* $\mathsf{union}(\mathrm{type}_\sigma, \mathrm{type}_\tau)$ *is empty.*

Proof. We assume here that the union operation returns the segments of the resulting split match word in a specific order, namely first the segments of the first argument, and then the segments of the second argument.

For types $(S_0, S_1, S_2) \in X_k$ and $(T_0, T_1, T_2) \in X_l$ we define that

$$(R_0, R_1, R_2) \in \mathsf{union}((S_0, S_1, S_2), (T_0, T_1, T_2))$$

when

- $k + l \leq n$, and
- the empty type R_0 is the concatenation of S_0 and T_0, and
- the free type R_1 maps $(i, m) \in \{1, \ldots, k+l\} \times M$ to $S_1(i, m)$ if $i \leq k$, and to $T_1(i-k, m)$ if $i > k$, and
- the bound type R_2 contains all tuples

$$\begin{aligned}
&(\sigma_1, \ldots, \sigma_k, \tau_1, \ldots, \tau_l), \quad \text{where } (\sigma_1, \ldots, \sigma_k) \in S_2, (\tau_1, \ldots, \tau_k) = T_0, \quad \text{and} \\
&(\sigma_1, \ldots, \sigma_k, \tau_1, \ldots, \tau_l), \quad \text{where } (\sigma_1, \ldots, \sigma_k) = S_0, (\tau_1, \ldots, \tau_k) \in T_2.
\end{aligned}$$

Let σ be a split match word with k segments, and τ a split match word with l segments, let the underlying words in σ and τ be, respectively, $w_1, \ldots, w_k$ and $v_1, \ldots, v_l$, and let the α-type of σ and τ be, respectively, (S_0, S_1, S_2) and (T_0, T_1, T_2). The second point of the lemma holds trivially, as for $k + l > n$ the result of the union function is the empty set. Assume now that $k + l \leq n$. We have to check that the α-type of the union of σ and τ is (R_0, R_1, R_2). The empty type of the union of σ and τ is the tuple

$$(\alpha(w_1), \ldots, \alpha(w_k), \alpha(v_1), \ldots, \alpha(v_l)),$$

hence it is the concatenation of S_0 and T_0. For $i \leq k$, the free type of the union maps (i, m) to the number of free positions x in the word w_i such that $\alpha(w_i \otimes \{x\}) = m$, which is exactly $S_1(i, m)$. Similarly for $i > k$, but now it describes word v_{i-k}, so it is $T_1(i - k, m)$. For the bound type notice that when x, y are matched positions in the union of σ and τ, then either both x, y are in the words coming from σ, or both in the words coming from τ.

To see that union is a move function, notice first that it returns either an empty set or a singleton set. Next notice that the R_0 and R_2 are determined by S_0, S_2, T_0, T_2. This means that the result's component in the semilinear space is determined by the argument's component, as required in the move function. By looking at the formula for R_1, we see that it is obtained from (S_1, T_1) by a move function. ◀

▶ **Lemma 18.** *For each two indices $i \neq j$ there exists a move function $\mathsf{join}_{i,j} \colon S \to P(S)$ such that for each split match word σ,*

- *if σ has $\geq \max(i, j)$ segments, and τ is obtained by joining the i-th and the j-th segment of σ, the type of τ is the only element of $\mathsf{join}_{i,j}(\mathrm{type}_\sigma)$, and*
- *if σ has $< \max(i, j)$ segments, the set $\mathsf{join}_{i,j}(\mathrm{type}_\sigma)$ is empty.*

Proof. We again have to assume some order of segments after the join operation: we say that the i-th and the j-th segments are removed from the tuple, and the new (joined) segment is appended at the end.

For $(S_0, S_1, S_2) \in X_k$ we define that $(R_0, R_1, R_2) \in \mathsf{join}_{i,j}(S_0, S_1, S_2)$ when

- $k \geq \max(i, j)$, and
- the empty type R_0 is obtained from S_0 by removing its i-th and j-th coordinates, and appending the product of the i-th and the j-th coordinate at the end of the tuple, and
- the free type R_1 maps $(a, m) \in \{1, \ldots, k-2\} \times M$ to $S_1(a', m)$ where $a' = a + |\{i, j\} \cap \{1, \ldots, a\}|$, so that $S_1(i, m)$ and $S_1(j, m)$ are not used; R_1 maps $(k-1, m) \in \{k-1\} \times M$ to

$$\sum_{m' \colon m' \cdot s_j = m} S_1(i, m') \; + \sum_{m' \colon s_i \cdot m' = m} S_1(j, m'),$$

  where $s_i, s_j \in M$ are the i-th and j-th coordinates of S_0, and
- the bound type R_2 contains tuples from S_2 with its i-th and j-th coordinates removed, and the product of the i-th and the j-th coordinates appended at the end of the tuple.

Let σ be a split word with k segments, let the underlying words in σ be $w_1, \ldots, w_k$, and let the α-type of σ be (S_0, S_1, S_2). The second point of the lemma is trivial; assume that $k \geq \max(i, j)$. We have to check that the α-type of the result of joining the i-th and j-th segment of σ is (R_0, R_1, R_2). For the empty type and the bound type this is obvious, as we do with the tuples of α-type the same what is done with the split match word (we use the fact that $\alpha((w_i w_j) \otimes P) = \alpha(w_i \otimes P) \cdot \alpha(w_j \otimes P)$ for any set of positions P). Similarly for the free type. The only nontrivial question is whether the number of free positions x in word $w_i w_j$ such that $\alpha((w_i w_j) \otimes \{x\}) = m$ is $R_1(k - 1, m)$, for each $m \in M$. But this is

equal to the number of free positions x in word w_i such that $\alpha(w_i \otimes \{x\})\alpha(w_j \otimes \emptyset) = m$, plus the number of free positions x in word w_j such that $\alpha(w_i \otimes \emptyset)\alpha(w_j \otimes \{x\}) = m$, which is exactly $R_1(k-1, m)$.

Like in the case of union, also the $\mathsf{join}_{i,j}$ function returns either a singleton set or an empty set, and R_0 and R_2 are determined by S_0 and S_2. By looking at the formula for R_1, we see that it is obtained from S_1 by a move function. This move function is different for different components of the semilinear space (for different S_0 and S_2), but that is immaterial. This shows that $\mathsf{join}_{i,j}$ is a move function. ◀

▶ **Lemma 19.** *There exists a semilinear function* $\mathsf{match}\colon S \to P(S)$ *such that for each split match word* σ*, and each element* $v \in S$*, the following two conditions are equivalent:*

- *there exists a split match word obtained from* σ *by applying the match operation, having type* v*, and*
- $v \in \mathsf{match}(\mathrm{type}_\sigma)$.

Proof. Before we define the match function, we first define (for each $1 \le k \le n$) a subset match'_k of the semilinear space

$$M^k \times \mathbb{N}^{\{1,\dots,k\}\times M} \times P(M^k) \times M^k \times \mathbb{N}^{\{1,\dots,k\}\times M} \times P(M^k) \times \{1,\dots,k\}^2 \times \mathbb{N}^{M\times M}.$$

We say that $(S_0, S_1, S_2, R_0, R_1, R_2, i, j, C) \in \mathsf{match}'_k$ when

1. $R_0 = S_0$, and
2. $i \neq j$, and
3. for each $m \in M$,

$$S_1(i,m) = R_1(i,m) + \sum_{n\in M} C(m,n), \quad \text{and}$$
$$S_1(j,m) = R_1(j,m) + \sum_{n\in M} C(n,m), \quad \text{and}$$
$$S_1(a,m) = R_1(a,m) \quad \text{for } a \notin \{i,j\}, \quad \text{and}$$

4. R_2 consists of all the tuples of S_2 and all the tuples $(m_1, \dots, m_k)$ such that $C(m_i, m_j) > 0$, and m_a is the i-th coordinate of S_0 for $a \notin \{i,j\}$.

Then we define match to be the projection of $\bigcup_{k=1}^{n} \mathsf{match}'_k$ to the first 6 coordinates.

Let σ be a split match word with k segments, let the underlying words in σ be $w_1, \dots, w_k$, and let the α-type of σ be (S_0, S_1, S_2). Let i, j be the two segments of σ between which we add the new edges of the matching (as in the definition of the match operation), and let τ be the resulting split match word. For $m_1, m_2 \in M$, let $C(m_1, m_2)$ be the number of positions x in w_i which were just matched with a position y in w_j such that $\alpha(w_i \otimes \{x\}) = m_1$ and $\alpha(w_j \otimes \{y\}) = m_2$. We will check that $(S_0, S_1, S_2, R_0, R_1, R_2, i, j, C) \in \mathsf{match}'_k$; for this we have to verify conditions 1–4. Condition 1 holds because the underlying words in σ and τ are the same. Condition 2 holds by assumptions of the match operation. Condition 3 holds by our definition of C. To see condition 4 notice that R_2 still contains all pairs from S_2, as positions x, y matched in σ are still matched in τ. But in the bound type R_2 we also have tuples

$$(\alpha(w_1 \otimes \{x,y\}), \dots, \alpha(w_k \otimes \{x,y\}))$$

for all newly matched positions x, y, where x is in w_i and y in w_j. The i-th coordinate of such tuple is equal to $\alpha(w_i \otimes \{x\})$, the j-th coordinate to $\alpha(w_j \otimes \{y\})$, and each other a-th coordinate ($a \notin \{i,j\}$) to $\alpha(w_a \otimes \emptyset)$, which is the a-th coordinate of the empty type S_0. This gives us the implication from the first point to the second point.

Again, let σ be a split match word with k segments, let the underlying words in σ be $w_1, \ldots, w_k$, and let the α-type of σ be (S_0, S_1, S_2). Let also $(R_0, R_1, R_2) \in \mathsf{match}(S_0, S_1, S_2)$. We have to show that $\mathrm{type}_\tau = (R_0, R_1, R_2)$ for some τ obtained by application of match to σ. By definition of match we know that for some i, j, C we have

$$(S_0, S_1, S_2, R_0, R_1, R_2, i, j, C) \in \mathsf{match}'_k.$$

We choose the matching between the positions of the i-th and the j-th segment of σ, so that for each $m_1, m_2 \in M$ there are exactly $C(m_1, m_2)$ positions x of w_i such that $\alpha(w_i \otimes \{x\}) = m_1$ matched with positions y of w_j such that $\alpha(w_j \otimes \{y\}) = m_2$. Condition 3 ensures that we have enough such positions, and that in w_a there remain exactly $R_1(a, m)$ free positions x such that $\alpha(w_a \otimes \{x\}) = m$, for each a, m. Similarly to the above paragraph, we see that the α-type of τ is (R_0, R_1, R_2).

Notice that match'_k is semilinear, as each one of the conditions defining it gives a semilinear subset of the space. Furthermore, match, being its projection, is also semilinear. ◀

Additionally we have a 0-ary function $\mathsf{base} \in P(S)$ which contains exactly the types of the split match words generated by the base operation. There are only finitely many such words, so this set is semilinear and can be computed.

Finally, we see that an element $v \in S$ is the type of some split match word of match width $\leq n$ if and only if v is generated by some derivation of rank $\leq n$. This follows from the above lemmas. Indeed, a derivation of rank $\leq n$ generating some split match word can be converted to a derivation of rank $\leq n$ generating the type of this word (and using functions base, union, $\mathsf{join}_{i,j}$, match), by simply replacing the split match word in each node by its type. Oppositely, we can also convert a derivation of rank $\leq n$ generating some $v \in S$ to a derivation of rank $\leq n$ generating some split match word having type v; the split match words in this derivation can be assigned in a bottom-up manner. Notice that the rank is defined in the same way for operations on split match words, and for the corresponding functions. From Theorem 16 it follows that the set of those $v \in S$ which are generated by some derivation of rank $\leq n$ (thus the set of those $v \in S$ which are a type of some split match word of match width $\leq n$) is semilinear and can be computed. At the end we restrict this set to X_k, and we obtain the set required in the statement of Theorem 9.

C Proof of Theorem 9: semilinear functions

In this appendix we prove Theorem 16. We do this in three steps: first we prove Lemma 20, then Lemma 21, and finally Theorem 9.

▶ **Lemma 20.** *Consider a semilinear space S, and a finite set F of semilinear functions of the form $f\colon S^{ar(f)} \to P(S)$. Assume that every function $f \in F$ is either a move function, or has arity 0. Then the set*

$$\{v \in S : v \text{ is generated by some derivation}\}$$

is a semilinear subset of S, and can be computed.

Notice that in this lemma we do not say anything about the rank of the derivation, but this is redundant, as a set F satisfying the assumptions allows only derivations of rank 0.

Proof. Abusing the notation a bit we identify a function returning a singleton (i.e. a move function) with the function returning the only element of the singleton.

Denote the semilinear space S as $\coprod_{i\in I} X_i$. Fix some component $r \in I$. It is enough to check that the set

$$\{v \in X_r : v \text{ is generated by some derivation}\}$$

is semilinear (and to compute it). Then we repeat the argument for every r.

We define a set $\mathcal{T}$ of (finite) trees each node x of which satisfies the following conditions.

1. It is labeled by a triple (f, i, g), where $i \in I$ is a component number, $f \in F$, and $g\colon X_i \to X_r$ is a move function.
2. Let k be the arity of f; then x has exactly k children. Denote the label of the j-th child of x by (f_j, i_j, g_j).
3. f restricted to $X_{i_1} \times \cdots \times X_{i_k}$ is a move function to X_i.
4. The root is labeled by (f, r, id), where $\mathsf{id}\colon X_r \to X_r$ is the identity (move) function.
5. If x is not a leaf, for each $a_1 \in X_{i_1}, \ldots, a_k \in X_{i_k}$ we have (where by $+$ we denote the coordinatewise sum of vectors)

$$g_1(a_1) + \cdots + g_k(a_k) = g(f(a_1, \ldots, a_k)).$$

The labels of these trees come from a finite alphabet (as there are only finitely many move functions between given linear spaces). All the conditions relate the label of a node with labels of its children. It follows that there exists a context-free grammar $\mathcal{G}_\mathcal{T}$, whose derivation trees are exactly the trees from $\mathcal{T}$ (triples (f, i, g) with 0-ary f are terminals in this grammar, the other tuples are nonterminals). Notice also that given move functions g, f and $g_1, \ldots, g_k$ one can compute if they satisfy condition 5; thanks to this the context-free grammar can be computed. In fact, given move functions g and f, there is exactly one sequence of move function $g_1, \ldots, g_k$ such that condition 5 is satisfied (it follows directly from the definition of a move function).

Let also $\mathcal{D}$ be the set of all derivations which generate an element of X_r. We make however a small twist: we assume that nodes of a derivation are labelled by triples (f, i, v), where $v \in X_i$, instead of pairs (f, v). Trivially there is a one-to-one correspondence between these two kinds of derivations. Notice that the labels of these derivations come from an infinite alphabet.

Next, observe that we have a natural mapping $\mathcal{M}$ from $\mathcal{D}$ to $\mathcal{T}$: A tree $D \in \mathcal{D}$ is mapped to the tree $T \in \mathcal{T}$ which has the same shape and the same first two coordinates of the labels. The third coordinate of the labels is assigned in the only correct way: in the root we have to write the identity function, and then we fill in the rest in a top-down manner (as already observed, the g function in a node determines the g functions in the children). Because D is a derivation, we obtain conditions 1-5 for T. The mapping $\mathcal{M}$ has the following properties.

▶ Claim 1. Let $x_1, x_2, \ldots, x_n$ be a maximal antichain of nodes of a tree $D \in \mathcal{D}$ (i.e. a maximal set of nodes such that none of them is a descendant of the other); simultaneously they can be treated as nodes of the tree $\mathcal{M}(D) \in \mathcal{T}$. For $1 \le j \le n$, let v_j and g_j be the last coordinate of x_j in D and in $\mathcal{M}(D)$, respectively. Then the last coordinate of the root label in D is

$$g_1(v_1) + \cdots + g_n(v_n)\ .$$

Proof. We prove this property by induction on the sum of depths (distances from the root) of the nodes x_j. The base case is when $n = 1$ and x_1 is the root. In this case the condition holds trivially, as g_1 is the identity function and v_1 is read from the root. In any other case, the antichain must contain all children of some node x (they can be found among nodes on the maximal depth); without loss of generality we can assume that these are $x_1, \ldots, x_k$.

Let (f, i, v) and (f, i, g) be the labels of their parent x in D and $\mathcal{M}(D)$, respectively. By condition 5 of $\mathcal{T}$ and by the definition of a derivation we know that

$$g_1(v_1) + \cdots + g_k(v_k) = g(f(v_1, \ldots, v_k)) = g(v), \quad \text{thus}$$
$$g_1(v_1) + \cdots + g_n(v_n) = g(v) + g_{k+1}(v_{k+1}) + \cdots + g_n(v_n).$$

Notice also that $x, x_{k+1}, \ldots, x_n$ is also a maximal antichain, and the sum of depths of its nodes is smaller. The claim now follows from the induction hypothesis for this antichain and the above equality. ◀

▶ Claim 2. Let $v \in X_r$. Then v is generated by some derivation if and only if there exists a word $(f_1, i_1, g_1) \cdots (f_n, i_n, g_n)$ generated by $\mathcal{G}_\mathcal{T}$, and a sequence of values $v_1, \ldots, v_n$ such that $v_j \in X_{i_j} \cap f_j()$ for $1 \leq j \leq n$, and $v = g_1(v_1) + \cdots + g_n(v_n)$.

Proof. To prove this, assume first that v is generated by some derivation $D \in \mathcal{D}$ (its root has v on the last coordinate of the label). Let $(f_1, i_1, g_1) \cdots (f_n, i_n, g_n)$ be the word of labels in all leaves of $\mathcal{M}(D)$; it is generated by $\mathcal{G}_\mathcal{T}$. Let $v_1, \ldots, v_n$ be the values on the third coordinate on the leaves' labels in D. By the definition of a derivation we know that $v_j \in X_{i_j} \cap f_j()$ for $1 \leq j \leq n$. Thanks to Claim 1 applied for the antichain of all leaves, we know that $v = g_1(v_1) + \cdots + g_n(v_n)$; this proves one direction.

For the other direction, assume that there exists a word $(f_1, i_1, g_1) \cdots (f_n, i_n, g_n)$ generated by $\mathcal{G}_\mathcal{T}$, and a sequence of values $v_1, \ldots, v_n$ such that $v_j \in X_{i_j} \cap f_j()$ for $1 \leq j \leq n$, and $v = g_1(v_1) + \cdots + g_n(v_n)$. Then we have a tree $T \in \mathcal{T}$ with n leaves, whose j-th leaf is labeled by (f_j, i_j, g_j) (for $1 \leq j \leq n$). Notice that there exists a tree $D \in \mathcal{D}$ such that $\mathcal{M}(D) = T$ and the labels of the leafs of D are $(f_1, i_1, v_1), \ldots, (f_n, i_n, v_n)$. Indeed, in D we take the shape and the first two coordinates of the labels from T; the third coordinate of the leaves labels is determined by by the sequence $v_1, \ldots, v_n$; the third coordinate of labels of all other nodes is filled in a bottom-up manner (the v values in the children of a node determine the v value in the node). Conditions 1-5 for T guarantee that we indeed obtain a derivation. Thanks to Claim 1 applied for the antichain of all leaves, we know that the last coordinate of the root's label in D is $v = g_1(v_1) + \cdots + g_n(v_n)$. But this means that v is generated by D. ◀

Let Γ be the set of all tuples used as labels of trees in $\mathcal{T}$. For $\gamma = (f, i, g) \in \Gamma$ let S_γ be the set of possible values of $g(v)$ for $v \in X_i \cap f()$. We see that S_γ is semilinear and can be computed (for each γ). For a word $w = \gamma_1 \ldots \gamma_n$ over alphabet Γ, let

$$S_w = S_{\gamma_1} + \cdots + S_{\gamma_n} = \{x_1 + \cdots + x_n \colon x_1 \in S_{\gamma_1}, \ldots, x_n \in S_{\gamma_n}\} .$$

From Claim 2 it follows that the set of elements $v \in X_r$ which are generated by some derivation is equal to the union of S_w over all words w generated by $\mathcal{G}_\mathcal{T}$. Recall that the Parikh image of a word $w \in \Gamma^*$ is a vector from $\mathbb{N}^\Gamma$, mapping $\gamma \in \Gamma$ to the number of occurrences of the γ letter in w. Notice that S_w depends only on the Parikh image of w (does not depend on the order of letters in w). The Parikh theorem [11] says that that the set of Parikh images of words in a context-free language is semilinear, and can be computed. We use this theorem to the language generated by $\mathcal{G}_\mathcal{T}$. It is easy to deduce that the union of the S_w sets over all words w generated by $\mathcal{G}_\mathcal{T}$ is also a semilinear sets, and it can be computed (based on the set of Parikh images of all words w generated by $\mathcal{G}_\mathcal{T}$, and on the sets S_γ). This finishes the proof. ◀

We say that a derivation *has a reset* if it has an internal node of rank 0 which is not labelled by a move function.

▶ **Lemma 21.** *Let $n \in \mathbb{N}$. Consider a semilinear space S, and a finite set F of semilinear functions of the form $f\colon S^{ar(f)} \to P(S)$. Then the set*

$$Y_k = \{v \in S : v \text{ is generated by some derivation without resets of rank} \leq n\}$$

is a semilinear subset of S, and can be computed.

Proof. Let $G \subseteq F$ be the set containing all 0-ary functions and move functions from F. We make an induction on n. If $n = 0$, functions from $F - G$ cannot be used at all, as otherwise the rank of a derivation without resets would be at least 1. Thus Y_0 is equal to

$$\{v \in S : v \text{ is generated by some derivation using only functions from } G\}.$$

This set is semilinear and can be computed thanks to Lemma 20.

Otherwise we make an induction on n. Let us first consider another set

$$Z = \{v \in S : v \text{ is generated by some derivation without resets of rank} \leq n, \text{ having a function from } F - G \text{ in the root}\}.$$

We will show that this a semilinear set, and can be computed. Notice that if a derivation without resets of rank $\leq n$ has a function from $F - G$ in the root, then all nodes except the root have rank $\leq n - 1$. Thus $v \in S$ is an element of Z if and only if it can be obtained by applying a function from $F - G$ to elements of Y_{n-1}, and v does not have zeroes on all coordinates (i.e. there will be no reset at the root of a derivation). By induction assumption Y_{n-1} is semilinear and can be computed. Because functions from $F - G$ are semilinear, after applying any of them we still have a semilinear set which can be computed. Finally, we can easily remove from our set the elements having zeroes on all coordinates, which shows that Z is semilinear and can be computed.

Next, consider a set of functions $H = G \cup \{h\}$, where h is a 0-ary function such that $h() = Z$. Notice that Y_n is equal to

$$\{v \in S : v \text{ is generated by some derivation using only functions from } H\}.$$

Indeed, consider a derivation using only functions from H. We do the following for every node x using h. Notice that the value in x is an element of Z, so it is generated by some derivation without resets of rank $\leq n$; we substitute this derivation instead of the node x (which is a leaf). This way we obtain a derivation without resets of rank $\leq n$ using functions from F (because all used derivations were not using resets, and the original derivation were using only move functions and 0-ary functions). Oppositely, consider a derivation without resets of rank $\leq n$. We identify all nodes which use a function from $F - G$ such that none of its ancestors uses a function from $F - G$. Of course the subtree rooted in such a node is a derivation without resets of rank $\leq n$, having a function from $F - G$ in the root, so the value in this node is in Z. We replace the whole subtree by a node labeled by h and the same value. When this is applied to all such nodes, we end with a derivation using only functions from H. Finally we see by Lemma 20 that this set is semilinear and can be computed. ◀

Proof of Theorem 16. Let $G \subseteq F$ be the set containing all 0-ary functions and move functions from F. Let also $S_0 \subseteq S$ be the (finite) set containing elements having zeroes on all coordinates. Recall that a rank is reset to 0 in a derivation when we use a function from $F - G$ and the result is from S_0.

Our algorithm will be working in stages (at each moment keeping only semilinear sets). As the result of the i-th stage we will have sets $Y_i \subseteq S_0$ and $Z_i \subseteq S$. We begin with the empty sets $Y_0 = Z_0 = \emptyset$.

In the i-th stage of the algorithm we consider the set of functions $F_i = F \cup \{f_i\}$, where f_i is a 0-ary functions defined as $f_i() = Y_{i-1}$. By Lemma 21 we can calculate the set of elements v generated by some derivation without resets of rank $\leq n$, using functions from F_i; denote this set Z_i. To Y_i we take all elements of Y_{i-1}, and those elements of S_0 which are results of some function $f \in F - G$ on arguments from Z_i. Of course Y_i is semilinear (even finite) and we can calculate it. If we have $Y_i = Y_{i-1}$, we finish the algorithm; otherwise we continue doing another stage.

This algorithm stops at some moment, as the sets Y_i can only increase, and are subsets of a finite set S_0. It is enough to observe that, if the algorithm stops after an i-th stage, the sets Z_i contain exactly the elements generated by some derivation of rank $\leq n$.

We will first show, by induction on i, that if $v \in Y_i \cup Z_i$ then v is generated by some derivation of rank $\leq n$ using functions from F, and additionally that if $v \in Y_i$ then this derivation has rank 0 in the root. This is trivial for $i = 0$ as the sets are empty. Take now any $v \in Z_i$ for $i > 0$. By definition it is generated by some derivation without resets of rank $\leq n$, using functions from F_i. In this derivation each leaf using f_i has a value from Y_{i-1}. By the induction assumption, this value is generated by a derivation of rank $\leq n$ using functions from F which has rank 0 in the root. We replace the considered leaf by this derivation (this is applied to all such leaves). In this way we obtain a derivation of rank $\leq n$ using functions from F, which generates v. Next, take $v \in Y_i$. If $v \in Y_{i-1}$, the thesis is trivial by induction assumption. Otherwise v is a result of applying a function f from $F - G$ to elements of Z_i. These arguments all have derivations of rank $\leq n$; we create a new derivation by putting (f, v) in the root, and attaching these derivations in the children of the root. The obtained derivation has rank $\leq n$, and its root has rank 0, because in the root we have a reset.

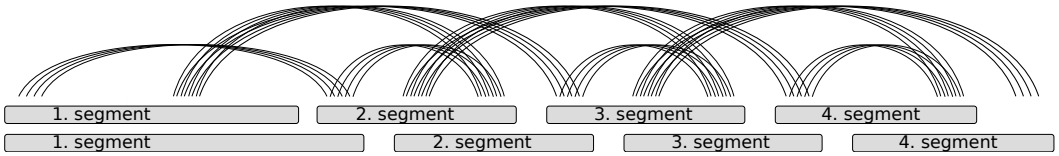
Oppositely, take $v \in S$ generated by some derivation of rank $\leq n$. We will show that then $v \in Z_i$, where i denotes the last stage of the algorithm; additionally if the derivation has a reset in the root, then $v \in Y_i$. The proof is by induction on the size of the smallest derivation (size can be defined in any way, say as a number of nodes). Assume first that we have a reset in the root; let $v_1, \ldots, v_n$ be the values in the children of the root of the derivation. By induction $v_1, \ldots, v_n$ are in Z_i, and because of the reset, we have $v \in S_0$ and the function in the root is from $F - G$, so $v \in Y_i = Y_{i-1} \subseteq Z_i$. Next assume that we do not have a reset in the root. We consider every highest node which has a reset (i.e. a node having a reset such that none of its ancestors has a reset). By the induction assumption, a value of such a node is in $Y_i = Y_{i-1}$. We replace this node, and the whole subtree rooted in it, by a single node using function f_i. This way we obtain a derivation derivation without resets of rank $\leq n$, using functions from F_i; thus $v \in Z_i$. ◀

D Braid width

Data braid and backward edges

The following example shows the importance of having backward edges in the definition of data braids, going to preceding segments.

▶ **Example 22.** Consider the four braids on four segments illustrated below.



This pattern can be arbitrarily prolonged to the right and easily generalized to any number of parallel braids. One can verify that there is no segmentation into three or more segments relative to which the braids would not make a backward move. More broadly, a stricter notion of braids disallowing backward progressions would result in the braid width of this family of match words to remain bounded even as the number of parallel paths increases. The idea is that most braids (except a handful that can travel from one segment to the next one by means of next position arcs) need to take the equal-length arcs, which go backwards. There is no other way to connect the braids from segments 2 and 3, but going backwards to segment 1.

Proof of Theorem 10

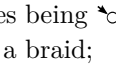
Observe that the first claim follows from the second, as class automata can existentially guess an arbitrary labelling and then evaluate any XPath expression on the resulting match word. The second claim is proved by reduction from the halting problem for Minsky machines. To every 2-counter Minsky machine $\mathcal{M}$ we associate a boolean XPath query Ψ whose models are match words inside of which an accepting run of $\mathcal{M}$ is encoded. To facilitate one such encoding we use match words of sufficiently high braid width. An appropriately labelled match word of braid width $3k$ can represent a run of $\mathcal{M}$ of length and width at most k, where *width* is the sum of the maximum values of the two counters reached during a run. As X is closed under relabellings we may freely specify what labelling is appropriate for encoding a run.

Consider a match word in X with a segmentation into $3k$ segments and a collection of k selected pairwise disjoint braids. We merge every three consecutive segments; we obtain k longer segments. The braids relative to the original segments are still braids, and moreover

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

We require the labelling to meet the following criteria.

The first position of each segment carries a symbol identifying it as a segment border. Every position carries a label specifying a transition of $\mathcal{M}$, such that positions within each segment specify the same transition, and transitions associated to consecutive segments are compatible in the sense that the target state of one is the source state of the next. Moreover, the transition associated to the first segment has the initial state of $\mathcal{M}$ as its source state and the transition associated to the last segment ends in an accepting state of $\mathcal{M}$. Furthermore, each position participating in a braid carries a symbol identifying it as such. In addition, every such position carries symbols

- specifying the incoming and the outgoing edges, or "directions", through which the braid enters and leaves the position, the six possibilities being  , plus six possibilities for the first and the last position of a braid;
- specifying whether it is the first position within a *new* segment along the braid;
- signifying whether it is a *front* or *rear* position, i.e. whether it lies in the right-most segment visited by the braid thus far or falls within the one prior;
- identifying it as contributing either to counter *#1* or to *#2*;
- identifying it as being either *on* or *off*;
- specifying whether it is to *keep* or *change* its value in the next segment.

The XPath formula Ψ is designed to validate the correctness of the intended encoding. In particular it needs to verify that

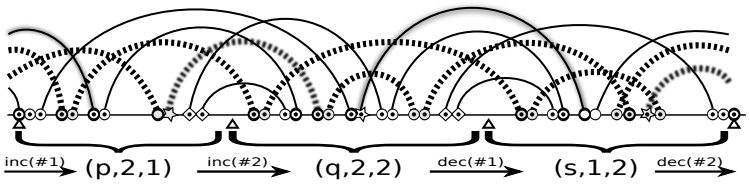


Figure 3 Detail of 4 braids on 3 segments encoding a partial run of $\mathcal{M}$. Legend: *#1* – solid arcs, white figures; *#2* – thick dashed arcs, shaded figures; segment border – triangle; *on* – black dot; *change* – star; *new* – fat circle; *rear* – diamond; edges where a *change* occurs are highlighted.

- every position marked with an outgoing successor (resp. predecessor) edge has a successor (resp. predecessor) marked with an incoming predecessor (successor) edge;
- every position marked with an outgoing data equality edge shares the same data value with a position marked with an incoming data equality edge;
- **(no big jumps)** between ends of each edge of a braid there is at most one segment border;
- **(*front* and *rear*)** if an edge of a braid has both its ends in the same segment, these ends are either both *rear* or both *front*; if it goes to the next segments, its right end is *front*; if it goes to the previous segment, its left end is *rear* and its right end is *front*;
- **(*new*)** every position marked *new* is either the first position of a braid, or the incoming braid edge comes from a *front* position of the previous segment;
- **(counter maintenance)** consecutive positions on a braid are either both *#1* or both *#2*; furthermore, they are either both *on* or both *off*, unless the first one is set to *change* and the next one is *new*, in which case they have complementary *on*/*off* status;
- **(change as needed)** for both $i \in \{1, 2\}$, in every segment marked with a transition that increments (decrements) counter i there is exactly one position marked $\#i$, *change*, and it is marked *off* (resp. *on*); if the transition does not change counter i, there is no such position;
- **(zero test)** if a transition testing counter i for zero marks a segment, then the previous segment does not contain positions marked $\#i$, *new*, *on*.

Each of the above constraints is easily expressible in XPath and together they ensure that an accepting run of $\mathcal{M}$ is encoded in the match word.

E Proofs of Lemmas 11 and 12

Lemma 11 follows from the following result.

▶ **Lemma 23.** *For any class automaton $\mathcal{A}$ there is a computable class automaton $\mathcal{A}'$, accepting all data words that have a match expansion in the language of $\mathcal{A}$.*

Proof. Let $\mathcal{A}$ be a class automaton with a class condition $L \subseteq (A \times B \times 2)^*$. The class automaton $\mathcal{A}'$ is built from $\mathcal{A}$ by guessing, for any given input data word $(w', \sim')$, a label from the alphabet $C = (A^2 \times B^2) \cup (A \times B)$. It verifies that, for every position x that has a previous and a future position with equal data value (like position x in Figure 2), we guess two labels (a_1, a_2) from the alphabet A and two labels (b_1, b_2) from B, such that a_1 is the actual label of x in w'. And for any other position it just guesses the current label from A and a label from B. The idea of this guessing is that a_1, a_2 will be the labels that correspond to the current and next positions of x in the data expansion of $(w', \sim')$, and b_1, b_2 are the

guessed labels needed to verify the property of $\mathcal{A}$. Let $s \in C^*$ be the word of such guessings, and let us assume that the aforementioned conditions of the guessing hold, since they can be easily verified in the class condition.

Let $w \in A^*$ be the result of replacing every position z in s with label (a_1, a_2, b_1, b_2) by the two-letter word a_1a_2, and every position z with guessed label (a_1, b_1) by a_1. Note that w has length $|s| + n$, where n is the number of positions of s with labels in $A^2 \times B^2$. The idea is that w is the guessed label of the expansion of $(w, \sim)$. Let $s' \in B^*$ be defined as w', but projecting onto the alphabet B instead of A (i.e., having b_1b_2/b_1 instead of a_1a_2/a_1). Finally, for any position x of w', let $\hat{x}$ be the $(x+n)$-th position of s, where n is the number of labels from $A^2 \times B^2$ to the left of x in s (excluding x). That is, $\hat{x}$ in w is where x in w' is mapped after the data expansion.

Now, the automaton $\mathcal{A}'$ checks,

- for every singleton class $\{x\}$ of $(w', \sim')$, that $w \otimes s' \otimes \{\hat{x}\} \in L$; and
- for every two positions x, y with $x < y$ in an equivalence class X of $\sim'$ which are consecutive in the word restricted to X, that $w \otimes s' \otimes \{\hat{x}+1, \hat{y}\} \in L$.

Note that it can do this for every two such x, y of a given equivalence class X. That is, there is a regular language L' such that for every class X with at least two elements, we have $w' \otimes s \otimes X \in L'$ if and only if for every two such x, y in X, $w \otimes s' \otimes \{\hat{x}+1, \hat{y}\} \in L$. This is because all such pairs x, y are ordered, in the sense that there are no two distinct (x, y) and (x', y') such that $x \leq x' \leq y \leq y'$. Thus the statement follows. ◀

In a similar way, Lemma 12 follows from the fact that class automata can guess the "contraction" (the opposite of the match expansion) of a data word.

▶ **Lemma 24.** *For any class automaton $\mathcal{A}$ there is a computable class automaton $\mathcal{A}'$, accepting all match expansions of data words in the language of $\mathcal{A}$.*

Proof. The automaton $\mathcal{A}'$ guesses a set of positions x such that x is the rightmost element from a equivalence class and $x+1$ is the leftmost element of a (different) equivalence class. It then treats all such positions x and $x+1$ as only one position with the label of x, and the class of x and $x+1$ as the same equivalence class, and verifies the class condition of $\mathcal{A}'$. ◀