Solutions in XML Data Exchange

Mikołaj Bojańczyk, Leszek A. Kołodziejczyk, Filip Murlak

University of Warsaw

Abstract

The task of XML data exchange is to restructure a document conforming to a source schema under a target schema according to certain mapping rules. The rules are typically expressed as source-to-target dependencies using various kinds of patterns, involving horizontal and vertical navigation, as well as data comparisons. The target schema imposes complex conditions on the structure of solutions, possibly inconsistent with the mapping rules. In consequence, for some source documents there may be no solutions.

We investigate three problems: deciding if all documents of the source schema can be mapped to a document of the target schema (absolute consistency), deciding if a given document of the source schema can be mapped (solution existence), and constructing a solution for a given source document (solution building).

We show that the complexity of absolute consistency is rather high in general, but within the polynomial hierarchy for bounded depth schemas. The combined complexity of solution existence and solution building behaves similarly, but the data complexity turns out to be very low.

In addition to this we show that even for much more expressive mapping rules, based on MSO definable queries, absolute consistency is decidable and data complexity of solution existence is polynomial.

Keywords: XML data exchange, regular queries, patterns, absolute consistency, solution building, solution existence

1. Introduction

One of the main challenges of modern data management is dealing with heterogeneous data. A typical scenario is that of data exchange, where one needs to restructure data stored in a source database under a target database schema, following a specification. The specification is given by a so-called *schema mapping*, a collection of logical formulas describing dependencies between the source schema and the target schema. The produced instance of data is called a *solution* for the source data with respect to the schema mapping.

Studies on relational data exchange and schema mappings were initiated several years ago [12, 13], and since then they have been a major topic (see recent surveys [5, 6, 16]).

In the XML context, the target schema, a DTD or XML Schema, imposes complex conditions on the structure of the solution. One of the consequences is that in practice schema mappings are

Email addresses: bojan@mimuw.edu.pl (Mikołaj Bojańczyk), lak@mimuw.edu.pl (Leszek A. Kołodziejczyk), fmurlak@mimuw.edu.pl (Filip Murlak)

often *over-specified*, in the sense that for some source instances there is no solution satisfying the specification and conforming to the target schema. One needs to be able to check if a mapping admits a solution or not.

Nowadays, schema mapping design is usually assisted by specialized software [17]. The ability to issue warnings that a mapping does not admit solutions for some source documents would be a most welcome feature, especially if the warning could also point out the source of the problem.

When a mapping is already there, one has to materialize the appropriate solution for the given source data. This requires checking if a solution exists, and constructing it.

We concentrate on three problems related to solutions:

- *Absolute consistency.* Is there a solution for every possible source document?
- *Solution existence.* Is there a solution for a given source document?
- *Solution building.* Find a solution for a given source document.

So far, absolute consistency was only investigated for very simple mappings, based on so-called tree patterns using only the child and descendant relations [1]. Solution building was investigated for even simpler mappings allowing only a restricted class of DTDs [3]. More practical mapping languages involve sibling order and data comparisons. For such mappings only the *consistency* problem was considered, i.e., whether *some* source document has a solution [1, 3]. Consistency of mappings, though interesting theoretically, seems to have less practical importance as it gives no information on how the mapping will perform on a given source document.

We work with mappings using general XML schemas, formalized as tree automata, DAG-shaped patterns using horizontal and vertical navigation, as well as data comparisons. We also consider a restricted case: *bounded-depth* mappings, where schemas only admit trees of bounded height. As most real-life schemas have small depth, this case is particularly important in practice.

Our main findings for pattern-based mappings can be summarized as follows:

- Absolute consistency is Π_2EXP-complete[1] in general, and Π_4P-complete in the bounded depth case.
- Data complexity of solution existence is in LOGSPACE, while combined complexity is NEXP-complete in general, and Σ_3P-complete for the bounded depth case.
- Solution building can be done in EXPSPACE in general, in EXP in the bounded depth case, and in P if the mapping is fixed.

The high combined complexity of the general case is disturbing, but the practically relevant bounded-depth case brings it down to low levels of the polynomial hierarchy, and the polynomial data complexity of solution existence and solution building gives real hope for applications.

We then go on to generalize these results even further. We show that for mappings based on queries definable in MSO, absolute consistency remains decidable, and the data complexity of solution existence is still polynomial.

The paper is organized as follows. After recalling the basic notions (Sect. 2), we study the solution building problem (Sect. 3) and the solution existence problem (Sect. 4) for pattern-based mappings. Then we apply the developed tools again to the absolute consistency problem (Sect. 5).

[1] Π_2EXP is the second level of the exponential hierarchy.

We conclude the first part of the paper with a complexity analysis of the three problems for bounded-depth mappings (Sect. 6). The second part extends the setting to mappings based on regular (MSO definable) queries. Section 7 contains the definitions and the proof strategy, while the proof itself is given in Sections 8 and 9. We finish with some suggestions for future work (Sect. 10).

2. Preliminaries

Data trees and multicontexts. The abstraction of XML documents we use is *data trees*: unranked labelled trees storing in each node a *data value*, i.e., an element of a countable infinite data domain $\mathbb{D}$. For concreteness, we will assume that $\mathbb{D}$ contains the set of natural numbers $\mathbb{N}$. Formally, a *data tree* over a finite labelling alphabet Γ is a structure $\langle T, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \ell_T, \rho_T \rangle$, where

- the set T is an unranked tree domain, i.e., a prefix-closed subset of $\mathbb{N}^*$ such that $n \cdot i \in T$ implies $n \cdot j \in T$ for all $j < i$;
- the binary relations $\downarrow$ and $\rightarrow$ are the child relation ($n \downarrow n \cdot i$) and the next-sibling relation ($n \cdot i \rightarrow n \cdot (i+1)$);
- $\downarrow^*$ and $\rightarrow^*$ are the transitive closures of $\downarrow$ and $\rightarrow$;
- the function ℓ_T is a labelling from T to Γ;
- ρ_T is a function from T to $\mathbb{D}$. We say that a node $s \in T$ stores the value d when $\rho_T(s) = d$.

Most often, when the interpretations of $\downarrow, \rightarrow, \ell_T$, and ρ_T are understood, we write just T to refer to a data tree. We use the terms "tree" and "data tree" interchangeably.[2] We write $|T|$ to denote the number of nodes of T. We write $T.v$ for the subtree of T rooted at v.

A *forest* is a sequence of trees. We write $F + G$ for the concatenation of forests F and G.

A *multicontext* C over an alphabet Γ is a tree over $\Gamma \cup \{\circ\}$ such that $\circ$-labelled nodes have at most one child. The nodes labeled with $\circ$ are called *ports*. A *context* is a multicontext with a single port, which is additionally required to be a leaf. A leaf port u can be *substituted* with a forest F, which means that in the sequence of the children of u's parent, u is replaced by the roots of F. An internal port u can be substituted with a context C' with one port u': first the subtree rooted at u's only child is substituted at u', then the obtained tree is substituted at u. Formally, the ports of a multicontext store data values just like ordinary nodes, but these data values play no role and we will leave them unspecified.

For a context C and a forest F we write $C \cdot F$ to denote the tree obtained by substituting the unique port of C with F. If we use a context D instead of the forest F, the result of the substitution is a context as well.

DTDs and automata. The principal schema language we use are tree automata, abstracting Relax NG [18, 19].

A *nondeterministic word automaton* can be presented as a tuple $\mathcal{B} = \langle \Gamma, Q, q_I, \delta, F \rangle$, where Γ is a finite input alphabet, Q is a finite set of states, $q_I \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\delta \subseteq Q \times \Gamma \times Q$ is the transition relation. A run on a word $\sigma_1 \sigma_2 \cdots \sigma_n \in \Gamma^*$ is a

[2] A different abstraction allows several *attributes* in each node, each attribute storing a data value [1, 3]. Attributes can be modelled easily with additional children, without influencing the complexity of the problems we consider.

sequence $q_1 q_2 \cdots q_{n+1} \in Q^*$ where q_1 is the initial state q_I, and $(q_i, \sigma_i, q_{i+1}) \in \delta$ for $i = 1, 2, \ldots, n$. A run is accepting if it ends in a final state, $q_{n+1} \in F$. A word is accepted if there is an accepting run for it. The set of accepted words is denoted by $L(\mathcal{B})$.

A *nondeterministic tree automaton* can be presented as a tuple $\mathcal{A} = \langle \Gamma, Q, \delta, F \rangle$ where Γ is the input alphabet, Q is a finite set of states, $F \subseteq Q$ is the set of final states, and δ is a function $Q \times \Gamma \to 2^{Q^*}$ such that $\delta(p, \sigma)$ is a regular language over Q for every $p \in Q$, $\sigma \in \Gamma$. A run of $\mathcal{A}$ on a data tree T is a labelling $\lambda : T \to Q$ such that for every $v \in T$ with n children,

$$\lambda(v0)\lambda(v1)\cdots\lambda(v(n-1)) \in \delta(\lambda(v), \ell_T(v))\,.$$

If v is a leaf, the condition reduces to $\varepsilon \in \delta(\lambda(v), \ell_T(v))$, which explains the lack of initial states. A run λ is *accepting* if the root is labelled with a final state, $\lambda(\varepsilon) \in F$. A data tree T is accepted by $\mathcal{A}$ if there is an accepting run for it. The set of trees accepted by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

Throughout the paper we assume that all states are reachable, i.e., for each state $p \in Q$ there is a tree which evaluates to p. This can be guaranteed by polynomial-time preprocessing.

In decision problems involving tree automata we assume that the regular languages $\delta(p, \sigma)$ are given by nondeterministic word automata $\mathcal{B}_{p,\sigma}$, and write $\|\mathcal{A}\|$ for the size of $\mathcal{A}$ including the automata $\mathcal{B}_{p,\sigma}$. In this case, an *extended run* of $\mathcal{A}$ on T is a run λ together with a labelling κ of T such that whenever $\ell_T(v) = \sigma$ and $\lambda(v) = p$, and v's children are $v_1, v_2, \ldots, v_k$, it holds that $q_I^{p,\sigma}, \kappa(v_1), \kappa(v_2), \ldots, \kappa(v_k)$ is an accepting run of $\mathcal{B}_{p,\sigma}$ on $\lambda(v_1), \lambda(v_2), \ldots, \lambda(v_k)$. The κ-label of the root is irrelevant, it can be any state of any $\mathcal{B}_{p,\sigma}$.

A simpler schema language is provided by DTDs. A *document type definition* (DTD) over a labelling alphabet Γ is a pair $D = \langle r, P_D \rangle$, where

- $r \in \Gamma$ is a distinguished root symbol;
- P_D is a function assigning regular expressions over $\Gamma - \{r\}$ to the elements of Γ, usually written as $\sigma \to e$, if $P_D(\sigma) = e$.

A data tree T *conforms to* a DTD D, denoted $T \models D$, if its root is labelled with r and for each node $s \in T$ the sequence of labels of children of s is in the language of $P_D(\ell_T(s))$. The set of data trees conforming to D is denoted $L(D)$.

Both DTDs and tree automata define languages of data trees. A DTD $D = \langle r, P_D \rangle$ over Γ can be viewed as a tree automaton $\mathcal{A}_D = \langle \Gamma, \Gamma, \delta, \{r\} \rangle$, with $\delta(\sigma, \sigma) = P_D(\sigma)$ and $\delta(\sigma, \tau) = \emptyset$ for $\sigma \neq \tau$. We can think of DTDs as restricted tree automata.

Patterns. Patterns were originally invented as convenient syntax for conjunctive queries on trees [7, 8, 14]. While most schema mapping research has concentrated on tree-shaped patterns, definable with an XPath-like syntax [1, 3], the full expressive power of conjunctive queries is only guaranteed by DAG-shaped patterns [7, 8]. Indeed, a tree shaped pattern cannot express "there is a path from a to b via c and d", but a DAG-shaped pattern or a conjunctive query can (see the middle pattern in Fig. 1). We base our mappings on DAG-shaped patterns, extending the setting used previously.

A *pattern* π over Γ can be presented as

$$\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi, \xi_\pi \rangle$$

where $\langle V, E_c \cup E_d \cup E_n \cup E_f \rangle$ is a finite DAG whose edges are split into child edges E_c, descendant edges E_d, next sibling edges E_n, and following sibling edges E_f, ℓ_π is a partial function from V to

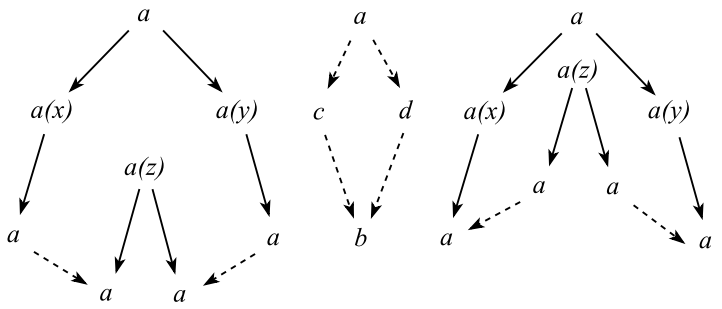


Figure 1: Typical patterns

Γ, and ξ_π is a partial function from V to the set of variables. The range of ξ_π, denoted $\operatorname{Rg}\xi_\pi$, is the set of variables used by π. By $\|\pi\|$ we mean the size of the underlying DAG.

A data tree $\langle T, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \ell_T, \rho_T\rangle$ *satisfies* a pattern $\pi = \langle V, E_c, E_d, E_n, E_f, \ell_\pi, \xi_\pi\rangle$ under a valuation $\theta : \operatorname{Rg}\xi_\pi \to \mathbb{D}$, denoted $T \models \pi\theta$, if there exists a homomorphism

$$h : \langle V, E_c, E_d, E_n, E_f, \ell_\pi, \xi_\pi \circ \theta\rangle \to \langle T, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*, \ell_T, \rho_T\rangle\,,$$

i.e., a function $h : V \to T$ such that

- $h : \langle V, E_c, E_d, E_n, E_f\rangle \to \langle T, \downarrow, \downarrow^*, \rightarrow, \rightarrow^*\rangle$ is a homomorphism of relational structures;
- $\ell_T(h(v)) = \ell_\pi(v)$ for all $v \in \operatorname{Dom}\ell_\pi$; and
- $\rho_T(h(u)) = \theta(\xi_\pi(u))$ for all $u \in \operatorname{Dom}\xi_\pi$.

We write $\pi(\bar{x})$ to express that $\operatorname{Rg}\xi_\pi \subseteq \bar{x}$. For $\pi(\bar{x})$, instead of $\pi\theta$ we usually write $\pi(\bar{a})$, where $\bar{a} = \theta(\bar{x})$. We say that T satisfies π, denoted $T \models \pi$, if $T \models \pi\theta$ for some θ.

Note that we use the usual non-injective semantics, where different vertices of the pattern can be witnessed by the same tree node, as opposed to injective semantics, where each vertex is mapped to a different tree node [11]. Under the adopted semantics patterns are closed under conjunction: $\pi_1 \wedge \pi_2$ can be expressed by the disjoint union of π_1 and π_2.

Without loss of generality we assume throughout the paper that *siblings have a common parent*, i.e., whenever two vertices of a pattern are connected by a next-sibling or a following sibling edge, there is a vertex connected to both of them by a child edge.

Examples of patterns are given in Fig. 1. Solid and dashed arrows represent E_c and E_d respectively. Figure 2 shows examples of homomorphisms witnessing that the left pattern is satisfied for $x = y = 1$, $z = 2$ and the right one is satisfied for $x = 3$, $y = 4$, $z = 0$. Note that in the left pattern, x and y always take the same value (not only in this tree).

For a class of patterns whose underlying graphs are tree-like we use the following syntax [1]:

$$\begin{array}{lcll} \pi & ::= & \sigma(x)[\lambda] \mid \sigma[\lambda] & \text{patterns} \\ \lambda & ::= & \varepsilon \mid \pi \mid /\!\!/\pi \mid \mu \mid \lambda, \lambda & \text{lists} \\ \mu & ::= & \pi \mid \pi \rightarrow \mu \mid \pi \rightarrow^* \mu & \text{sequences} \end{array}$$

where $\sigma \in \Gamma \cup \{\_\}$. That is, a pattern is given by its root node with label σ and variable x (or no variable), and a list of subtrees λ. A subtree can be rooted at a child of the root (corresponding

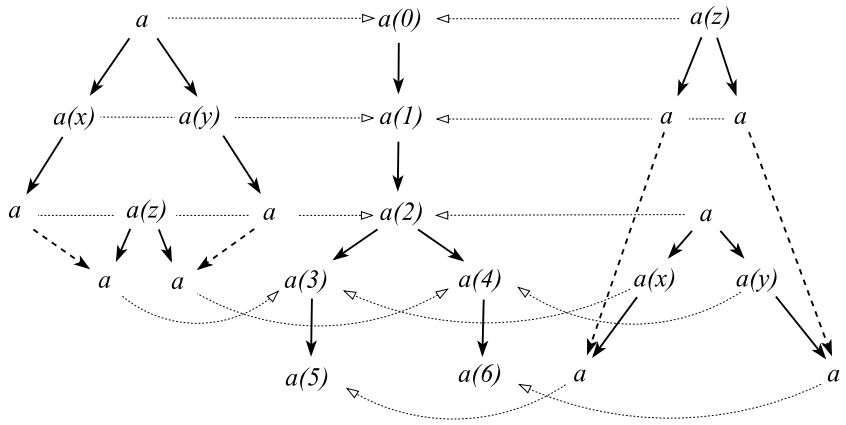


Figure 2: Homomorphisms witness satisfaction

to π in the definition of λ), or its descendant (corresponding to $/\!\!/\pi$). The subtrees rooted in the children of the root can be additionally connected into sequences μ by means of the next sibling relation (represented by $\rightarrow$) or the following sibling relation (represented by $\rightarrow^*$). The wildcard symbol $\_$ is used to denote a vertex without a label. We also write σ for $\sigma[\,]$, σ/π for $\sigma[\pi]$, $\sigma/\!\!/\pi$ for $\sigma[/\!\!/\pi]$, $\sigma[\lambda, \bigwedge_{i=1}^{k} \pi_i, \lambda']$ for $\sigma[\lambda, \pi_1, \pi_2, \ldots, \pi_k, \lambda']$, and similarly for $\sigma(x)$.

We also use this syntax to describe data trees. In this case we skip the horizontal ordering and write $r\big[a(1), b(1), a(2)\big]$ for a tree whose root is labelled with r and has three children, the leftmost labelled with a and storing the data value 1, etc. Note that the data value in the root is left unspecified. This will usually be the case, as the root is rarely used to store meaningful data.

Pattern-based mappings. A *(pattern-based) schema mapping* $\mathcal{M} = \langle \mathcal{S}_s, \mathcal{S}_t, \Sigma \rangle$ consists of a source schema $\mathcal{S}_s$, a target schema $\mathcal{S}_t$, and a set Σ of *source-to-target dependencies* (shortly, dependencies) that relate source and target instances. The source and target schemas are given as tree automata or as DTDs. Dependencies are expressions of the form:

$$\pi(\bar{x}), \eta(\bar{x}) \;\longrightarrow\; \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y}),$$

where π, π' are patterns, and η, η' are conjunctions of equalities and inequalities among $\bar{x}$ and $\bar{x}, \bar{y}$ respectively. We assume the usual safety condition: each variable used in η is also used in π.

A pair of trees (T, T') satisfies the dependency above if whenever $T \models \pi(\bar{a})$ and $\eta(\bar{a})$ holds, there is $\bar{b}$ such that $T' \models \pi'(\bar{a}, \bar{b})$ and $\eta'(\bar{a}, \bar{b})$ holds. Given a source $T \in L(\mathcal{S}_s)$, a target $T' \in L(\mathcal{S}_t)$ is called a *solution* for T under $\mathcal{M}$ if (T, T') satisfies all the dependencies in Σ. We let $\mathcal{M}(T)$ stand for the set of all solutions for T.

Suppose that the source schema is given as a DTD $r \to a^* b^*$, and the target schema is $r \to a\, b^*$. A mapping might be given, e.g, by a single dependency

$$r\big[a(x_1) \to b \to^* \_(x_2)\big], x_1 \neq x_2 \longrightarrow r\big[a(x_1) \to^* b(x_2)\big]\,.$$

Under this mapping every source tree has a solution. On the other hand, if we replace the source DTD with $r \to (a + b)^*$, some source trees will have no solutions.

In the complexity analysis of computational problems we use various restrictions on mappings. These include imposing tree-structure on patterns (instead of DAG structure), forbidding some axes in patterns, forbidding equality or inequality in dependencies, and restricting schemas to DTDs.

Evaluation and satisfiability. The evaluation problem is: given a pattern $\pi(\bar{x})$, a tuple $\bar{a}$, and a tree T, decide if $T \models \pi(\bar{a})$. The data complexity of this problem is folklore, the combined complexity was established in [14].

Proposition 2.1. *The data complexity of evaluating patterns is in* LOGSPACE *and the combined complexity is* NP*-complete.*

Proof. A pattern is an FO query using relations $\rightarrow^*, \downarrow^*$, which can be computed in LOGSPACE (for $\downarrow^*$ move against the arrows). In consequence, a fixed pattern can be evaluated in LOGSPACE. The upper bound for combined complexity is obtained by guessing a witnessing homomorphism and verifying it. The lower bounds for various sets of axes can be found in [14]. □

A pattern π is satisfiable with respect to an automaton $\mathcal{A}$ if there is a tree $T \in L(\mathcal{A})$ such that $T \models \pi$. The *satisfiability problem* is: given a pattern π and an automaton $\mathcal{A}$, decide if π is satisfiable with respect to $\mathcal{A}$. This problem was shown to be NP-complete for many variants of patterns [2, 4, 8, 15]. As we need the details of the NP-algorithm in some arguments, we sketch it here briefly.

Let h be a homomorphism from π to T. The *support* of h, denoted $\operatorname{supp} h$, is the subtree of T obtained by removing all nodes that cannot be reached from $\operatorname{Rg} h$ by going up, left, and right.

Lemma 2.2. *For each pattern π satisfiable wrt an automaton $\mathcal{A}$, there exists $T \in L(\mathcal{A})$ and a homomorphism $h : \pi \to T$ with $|\operatorname{supp} h| \leq 12\|\pi\| \cdot \|\mathcal{A}\|^2$.*

Proof. Take a tree $T \in L(\mathcal{A})$ satisfying π and let h be a homomorphism from π to T. Divide the nodes of $\operatorname{supp} h$ into four categories: the nodes from the image of h are *red*, the nodes that are not red and have more than one child that is an ancestor of a red node (or is red itself) are *green*, the others are *yellow* if they are ancestors of red nodes, and *blue* otherwise. Let N_{red}, N_{green}, N_{yellow}, and N_{blue} be the numbers of red, green, yellow, and blue nodes.

By definition, $N_{\text{red}} \leq \|\pi\|$. Also $N_{\text{green}} \leq \|\pi\|$: when going bottom-up, each green node decreases the number of subtrees containing a red node by at least one, and since in the root we arrive with one subtree containing a red node, $N_{\text{green}} \leq N_{\text{red}}$. By a pumping argument we may assume that all yellow $\downarrow$-paths and all blue $\rightarrow$-paths in $\operatorname{supp} h$ are not longer than $\|\mathcal{A}\|$. The number of (maximal) yellow $\downarrow$-paths is at most $N_{\text{red}} + N_{\text{green}}$. Hence there are at most $2\|\pi\| \cdot \|\mathcal{A}\|$ yellow nodes. Since all blue nodes are siblings of nodes of other colours, the number of (maximal) blue $\rightarrow$-paths is at most $2(N_{\text{red}} + N_{\text{green}} + N_{\text{yellow}}) \leq 4\|\pi\| \cdot (\|\mathcal{A}\| + 1)$ and so $N_{\text{blue}} \leq 4\|\pi\| \cdot (\|\mathcal{A}\| + 1)\|\mathcal{A}\|$. Altogether we have at most $2\|\pi\|(\|\mathcal{A}\| + 1)(2\|\mathcal{A}\| + 1) \leq 12\|\pi\| \cdot \|\mathcal{A}\|^2$ nodes. □

Proposition 2.3. *The satisfiability of patterns is in* NP*.*

Proof. By Lemma 2.2, we can guess a homomorphism into a polynomial tree T together with an "almost" run on T: in every leaf we additionally guess a sequence of states to which the missing subtrees should evaluate (by the pumping lemma, the sequence can be linear in the size of the automaton). Since all states are reachable, T can be extended to a tree accepted by the automaton. Checking correctness of the "almost" run, and of the homomorphism is polynomial in the size of the pattern and the tree T, hence it is polynomial. □

3. Building solutions

Outline. Building a solution for a given source tree T with respect to a mapping $\mathcal{M}$ can be seen as a constructive version of the satisfiability problem. Consider a mapping $\mathcal{M}$ with a single dependency $\pi(\bar{x}) \to \pi'(\bar{x}, \bar{y})$. For a given source tree T, we need to build a target tree T' such that *for every* $\bar{a}$ satisfying $T \models \pi(\bar{a})$ *there exists* $\bar{b}$ satisfying $T' \models \pi'(\bar{a}, \bar{b})$. In other words, T' should satisfy the conjunction of $\exists \bar{y}\ \pi'(\bar{a}, \bar{y})$ over all $\bar{a}$, which can be seen as a single pattern π'', polynomial in the size of T. The solution we are looking for is any tree satisfying π'' and conforming to the target schema. Unfortunately, building such a tree for an arbitrary given pattern π'' is intractable, unless NP = P. Indeed, constructing such a tree is clearly at least as hard as deciding if it exists, and it is easy to check the latter problem is NP-hard even for a fixed schema.

In what follows, we exploit the fact that the pattern π'' is not an arbitrary pattern, but a conjunction of polynomially many fixed size patterns. For a single pattern $\pi'(\bar{a}, \bar{y})$ of fixed size, we can easily construct a suitable target tree—a *partial solution* for $\bar{a}$—in P using exhaustive search. The difficult part is to combine all partial solutions into one tree.

What we need is the ability to build a π'*-union* of target trees $T_1, T_2, \ldots, T_n$, i.e., a tree T, such that whenever $T_i \models \pi'(\bar{a}, \bar{b})$ for some i, then $T \models \pi'(\bar{a}, \bar{b})$. Without a target schema a π'-union can be obtained by combining $T_1, T_2, \ldots, T_n$ under a fresh root node. In the presence of a target schema, however, a π'-union need not exist. For instance, consider the schema $r \to a\, b^*$, two trees $r[a(1), b(2)]$ and $r[a(3), b(4)]$, and $\pi'(x_1, x_2) = r\,[a(x_1) \to^* \_\,(x_2)]$, saying "there is an a-node storing x_1 with a following sibling storing x_2". The first tree satisfies $\pi'(1, 2)$, and the second satisfies $\pi'(3, 4)$, but clearly no tree conforming to the schema can satisfy both $\pi'(1, 2)$ and $\pi'(3, 4)$.

We resolve this difficulty by partitioning the set of target trees into subsets closed under π'-union. For instance, the set of all trees conforming to $r \to a\, b^*$ can be partitioned into sets L_d, $d \in \mathbb{D}$, consisting of all trees storing d in the a-node. It is easy to see that each L_d is closed under π'-union: for $T_1, T_2, \ldots, T_n$, the union tree simply needs to include all the b-nodes of $T_1, T_2, \ldots, T_n$.

The elements of the partition will be represented by finite (exponential-size) descriptions called *kinds*. Note that the a-node in the example above is the critical area of each tree: one can easily collect b-nodes from two trees in one tree, but there is always only one a-node. This leads to the main idea behind kinds: distinguishing the critical and the non-critical areas of trees. A kind should specify the critical areas entirely, including data values. The non-critical areas will be just "holes" in the tree, associated with the parts of the schema that can be repeated.

The solution building algorithm will first choose an appropriate target kind, then provide partial solutions of this kind for each tuple $\bar{a}$, and finally merge the partial solutions into one tree.

Kinds. An $\mathcal{A}$*-decorated multicontext* C is a multicontext whose every port is one of the following:

- p*-port*, a leaf port decorated with a state p of $\mathcal{A}$,
- p, p'*-port*, an internal port with a single child, decorated with states p, p' of $\mathcal{A}$,
- q, q'*-port*, a leaf port decorated with states q, q' of one of $\mathcal{A}$'s horizontal automata.

We refer to the three kinds of ports above as tree, context and forest ports.

By $L(C)$ we denote the set of trees that *agree* with C, i.e, can be obtained from C by compatible substitutions:

- a forest is compatible with a p-port if it is a tree, and admits a run of $\mathcal{A}$ evaluating to p;

- a context D is compatible with a p, p'-port if it admits a run of $\mathcal{A}$ in which the port of D is assumed to evaluate to p and the whole context evaluates to p' (formally, we add a transition $\delta(p, \circ) = \varepsilon$ to $\mathcal{A}$);
- a forest $F = S_1, S_2, \ldots, S_m$ is compatible with a q, q'-port if S_i admits a run of $\mathcal{A}$ evaluating to p_i such that there is a run of the corresponding horizontal automaton over $p_1 p_2 \ldots p_m$ starting in q and ending in q'.

For each $T \in L(C)$ there exists a *witnessing substitution*: a sequence of forests or contexts T_u, with u ranging over the ports of C, such that substituting each u in C with T_u we obtain T.

An *accepting run* of $\mathcal{A}$ over an $\mathcal{A}$-decorated multicontext must satisfy the usual conditions for ordinary nodes, and the following rules for the ports:

- each p-port is evaluated to p,
- the only child of each p, p'-port u is evaluated to p, and u evaluates to p',
- if u is a q, q'-port, the horizontal automaton arrives at u in state q and leaves in state q'.

More precisely, we assume that each port u is labelled with a unique label $\circ_u$ and we consider an accepting run of the automaton $\mathcal{A}$ extended with the following transition rules for each u:

$$\begin{array}{ll} \delta(p, \circ_u) = \varepsilon & \text{if } u \text{ is a } p\text{-port}, \\ \delta(p', \circ_u) = p & \text{if } u \text{ is a } p, p'\text{-port}, \\ \delta(p_u, \circ_u) = \varepsilon \text{ and } q, p_u \to q' & \text{if } u \text{ is a } q, q'\text{-port}, \end{array}$$

where p_u is a fresh state. Thus, if a tree T is obtained from C by substituting compatible forests or contexts, an accepting run of $\mathcal{A}$ over T can be obtained by combining an accepting run over C with runs witnessing compatibility. In consequence, if C admits an accepting run of $\mathcal{A}$, then $L(C) \subseteq L(\mathcal{A})$.

In order to implement the algorithm outlined in the beginning of this section we need small objects witnessing that a pattern is satisfied in a tree agreeing with a decorated multicontext. An argument similar to the proof of Lemma 2.2 leads to the following bound.

Lemma 3.1. *For an $\mathcal{A}$-decorated multicontext C, a pattern π, and a tuple $\bar{a}$, satisfiability of $\pi(\bar{a})$ in a tree agreeing with C can be witnessed by an object polynomial in the size of π, the height and branching of C, and the size of $\mathcal{A}$.*

Proof. We need to check if the ports of C can be filled in such a way that $\pi(\bar{a})$ is satisfied in the resulting tree. The contexts/forests needed might be large, but just like in the proof of Lemma 2.2 all we need is a support for the homomorphism from $\pi(\bar{a})$ to the tree. It is easy to see that the total size of the support in the substituted parts can be bounded polynomially. In particular, the number of ports involved in the realisation of $\pi(\bar{a})$ is polynomial, and for the remaining ones the support is empty. A homomorphism from $\pi(\bar{a})$ into C extended with the support can also be stored in polynomial memory. Verifying the witness can be done in P. □

We are now ready to introduce the notion of kind. The only restriction that needs to be imposed on decorated multicontexts is that the ports should be associated to parts of the schema

that can be repeated. This is guaranteed by taking the decorating states from appropriate strongly connected components of the automaton, as defined below.

With a tree automaton $\mathcal{A} = (\Gamma, Q, \delta, F)$ we associate a graph $G_{\mathcal{A}} = (Q, E)$, where $(p, p') \in E$ iff $Q^* p\, Q^* \cap \delta(p', \sigma)$ is nonempty for some $\sigma \in \Gamma$. We speak of strongly connected components (SCCs) of $\mathcal{A}$ meaning SCCs of $G_{\mathcal{A}}$. We say that an SCC is *non-trivial* if it contains an edge (it might have only one vertex though). A non-trivial SCC X is *branching* if there exist $p, p_1, p_2 \in X$ such that $Q^* p_1 Q^* p_2 Q^* \cap \delta(p, \sigma)$ is nonempty for some σ. If this is not the case, X is *non-branching*. We also work with SCCs of word automata, defined in the natural way.

An *$\mathcal{A}$-kind* K is an $\mathcal{A}$-decorated multicontext admitting an accepting run of $\mathcal{A}$, and such that each port u satisfies the following conditions:

- if u is a p-port, then p is contained in a branching SCC of $\mathcal{A}$;
- if u is a p, p'-port, then p, p' are in the same non-branching SCC of $\mathcal{A}$;
- if u is a q, q'-port, then q, q' are in the same SCC of one of $\mathcal{A}$'s horizontal automata.

Let us come back to the example discussed in the beginning of this section. The language L_d can be represented by the kind $r\big[a(d), \circ\big]$, where the port is a forest port decorated with states q, q, and q is the non-initial state of the two-state automaton recognizing $a\, b^*$ (transitions are $q_I, a \to q$ and $q, b \to q$, the only final state is q). Note that technically speaking, the horizontal automaton reads states of the vertical automaton rather than node labels, but since the schema is originally given as a DTD, the state in each node coincides with the label. Note also that the empty forest is compatible with this port.

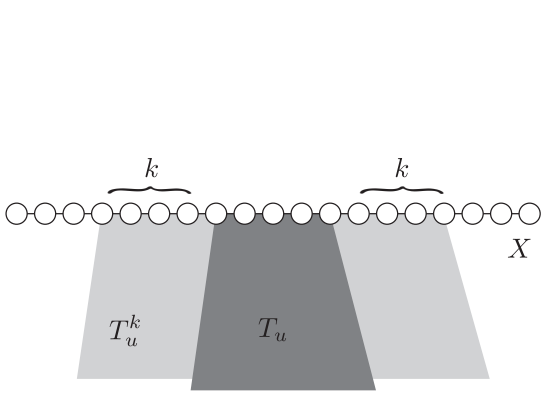
Recall that our aim is to show that each regular language can be covered by exponential-size kinds, each defining a language closed under π'-unions. Obviously, the choice of kinds has to take π' into account: the languages L_d are closed under union for $\pi'(x_1, x_2)$ saying "there is an a-node storing x_1 with a following sibling storing x_2", but not for $\pi''(x_1, x_2)$ saying "there is an a-node storing x_1 whose next sibling stores x_2". For π'' one should use kinds $r\big[a(d), b(d'), \circ\big]$ and $r\big[a(d)\big]$, where d, d' range over $\mathbb{D}$. Intuitively, to ensure closure under π'-union, we need to specify large enough areas of the tree around the ports. This is formalized by the notion of margin.

Margins. For $m \in \mathbb{N}$, we say that K *has margins of size* m if there exists an accepting extended run λ, κ over K such that for each port u

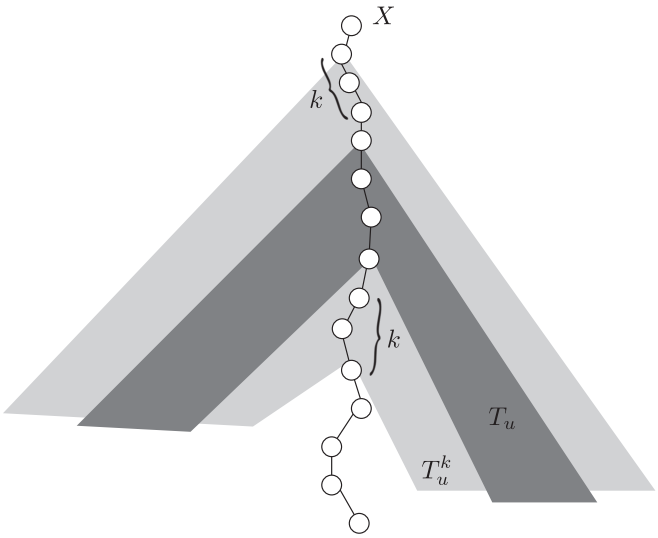
- if u is a p-port with p in an SCC X, there is a $\downarrow$-path $v_{-m}, v_{-m+1}, \ldots, v_0$ such that the only port on the path is $v_0 = u$ and $\lambda(v_j) \in X$ for all $-m \leq j < 0$,
- if u is a p, p'-port with p, p' in an SCC X, there is a $\downarrow$-path $v_{-m}, v_{-m+1}, \ldots, v_m$ such that the only port on the path is $v_0 = u$ and $\lambda(v_j) \in X$ for all $-m \leq j < 0$ and all $0 < j \leq m$,
- if u is a q, q'-port with q, q' in an SCC X of some horizontal automaton, there is a sequence of consecutive siblings $v_{-m}, v_{-m+1}, \ldots, v_m$ such that the only port in the sequence is $v_0 = u$ and the segment of the (horizontal) run corresponding to the sequence stays in X.

For $k \leq m$, $T \in L(K)$, and a witnessing substitution T_u, u ranging over ports in K, we define T_u^k as the extension of T_u by k nodes along the margins (see Fig. 3). More precisely, using the notation above:

- if u is a p-port, let $T_u^k = T.\tilde{v}_{-k}$,



(a) X is an SCC of a horizontal automaton



(b) X is a non-branching SCC of $\mathcal{A}$

Figure 3: Extension of T_u by k nodes along the margins around a port u associated with an SCC X.

- if u is a p, p'-port, T_u^k is the context obtained from $T.\tilde{v}_{-k}$ by replacing $T.\tilde{v}_k$ with a port,
- if u is a q, q'-port, let $T_u^k = \sum_{i=-k}^{-1} T.\tilde{v}_i + T_u + \sum_{i=1}^{k} T.\tilde{v}_i$,

where $\tilde{v}$ denotes the node in T corresponding to v. In particular, $T_u^0 = T_u$.

We show that margins of size $\|\pi\|$ guarantee closure under π-union, and each tree is covered by an exponential-size kind with such margins. Proving the first property requires more tools and we postpone it for a while. The second property is given by the lemma below.

Lemma 3.2. *For all $\mathcal{A}$, $T \in L(\mathcal{A})$ and $m \in \mathbb{N}$ there is an $\mathcal{A}$-kind K with margins of size m, height and branching at most $(2m+1)\|\mathcal{A}\|$, such that $T \in L(K)$.*

Proof. Fix an extended accepting run λ, κ of $\mathcal{A}$ on T. Prune T introducing ports as stubs according to the following rules.

First process the SCCs of the horizontal automata. Each SCC X in which the run stays for at least $2m+1$ steps becomes a q, q'-port with q, q' determined by κ, and the first and last m steps of the run spent in X become the margins around this node. The subtrees rooted at the margin nodes are kept, but the subtrees rooted at the remaining nodes in the middle are lost together with their roots.

Next, deal with the SCCs of $\mathcal{A}$. Each maximal path P on which the automaton stays in a branching SCC X is cut off at depth $m+1$; under the freshly obtained leaf put a p-port, where p is the state in the root of the removed subtree.

For non-branching X only consider paths of length at least $2m+1$. Replace the subtree rooted at the $(m+1)$st node of the path with a p, p'-port for suitable p, p', and under it put the subtree originally rooted at the mth node of the path, counting from the bottom.

Note that if a run stays in some component for at most $2m$ steps (m for branching SCCs), then no port is introduced. In particular, for some small trees T, the resulting kind may be T itself (no ports).

Let K be the resulting kind. By construction $T \in L(K)$. Each sequence of children in T can be split into segments according to the SCCs of the horizontal automata. Since no run enters the same SCC twice, we have at most as many segments as SCCs (not more then $\|\mathcal{A}\|$). After T has been pruned, each segment is shortened to at most $2m+1$ nodes. Hence, the branching of K can be bounded by $(2m+1)\|\mathcal{A}\|$. A similar argument shows that each branch of K has length at most $(2m+1)\|\mathcal{A}\|$. □

Safe extensions. The purpose of margins is to isolate the substituted forests/contexts from each other, and from the fixed part beyond the margins. In this way realizations of π can be rearranged after the substituted forests/contexts are extended to accommodate additional instances of π, provided that the extensions follow certain rules.

Let K be a kind with margins of size m and let $T \in L(K)$ with a witnessing substitution T_u. An *m-safe extension* of the substitution T_u is any substitution T'_u obtained by replacing each T_u with some compatible

- forest of the form $I + T_u^m + I'$, if u is a forest port;
- tree of the form $I \cdot T_u^m$, if u is a tree port;
- context of the form $I \cdot T_u^m \cdot I'$, if u is a context port.

A tree $T' \in L(K)$ is an *m-safe extension* of $T \in L(K)$ if it results from a substitution T'_u that is an m-safe extension of some witnessing substitution T_u.

Lemma 3.3. *Fix a pattern π, a natural number $m \geq \|\pi\|$, a kind K with margins of size m, and a tree $T \in L(K)$. For each m-safe extension T' of T*

$$T' \models \pi(\bar{a}) \text{ whenever } T \models \pi(\bar{a})\,.$$

Proof. Fix a witnessing substitution T_u, a tuple $\bar{a}$ such that $T \models \pi(\bar{a})$ and a witnessing homomorphism $h : \pi \to T$. We define a partition $\mathcal{V}$ of V_π, the set of π's vertices, such that a homomorphism $h' \colon \pi \to T'$ can be defined independently on each part as long as some simple rules are satisfied.

The areas potentially affected by the extension are the T_u's. With each port u such that T_u is not disjoint from the image of h, we associate a set Z_u satisfying $T_u \subseteq Z_u \subseteq T_u^m$, in which the homomorphism can be modified independently of the rest of the tree. There are three cases, depending on the type of u.

- If u is a forest port, then $T_u^m = M + T_u + N$ and since the image of h has cardinality at most $\|\pi\|$ and is not disjoint from T_u, by the pigeon-hole principle there exist roots v in M and v' in N outside of the image of h. (If there is choice, choose the ones closest to T_u). Let Z_u be the forest of trees rooted at the sequence of siblings between v and v' (excluding v and v').
- If u is a tree port, then $T_u^m = M \cdot T_u$ and there exists a node v in M on the path from the root to the port, closest to T_u, that is not in the image of h. Let $Z_u = T.v$.
- If u is a context port, then $T_u^m = M \cdot T_u \cdot N$, and one can find v on the path from the root to the port in M and v' on the path from the root to the port in N, closest to T_u, that are not in the image of h. Let $Z_u = T.v - T.v'$.

The sets Z_u need not be disjoint, but we can pick a pairwise disjoint subfamily $\mathcal{Z}$ covering all the sets T_u that intersect the image of h. To prove this, we first show by case analysis that for each u', u'' either $T_{u'} \subseteq Z_{u''}$, or $T_{u''} \subseteq Z_{u'}$, or $Z_{u'}$ and $Z_{u''}$ are disjoint. In fact, in all cases except the last one, $Z_{u'}$ and $Z_{u''}$ either are disjoint or one is contained in the other.

- For two forest ports, we are dealing with two forests whose roots are children of some nodes w and w'. If such forests are not disjoint, and neither is contained in a subtree of the other, then they share one of the roots. But this is impossible: two sibling ports are always separated by a margin, which must contain a node that is not in the image of h; the associated forests are disjoint because they were chosen minimal.

- For a tree port and a forest port (or a tree port), the corresponding sets are: a subtree, and a subforest whose roots are siblings. Such sets are always either disjoint, or one is contained in the other.

- For a context port and a tree port, whose corresponding sets are not disjoint and neither is contained in the other, the root of the tree must be contained in the path between the root and the port of the context. But this would mean that the margin paths of the two ports are not disjoint, which implies that the SCCs associated with the ports are the same. This leads to a contradiction, as one was assumed branching and the other non-branching.

- For two context ports, if the contexts are not disjoint and neither is contained in the other, the root of one of them lies on the path between the root and the port of the other. Like above, in this case both ports are associated with the same SCC. Moreover, since the SCC is non-branching, the two ports are on the same branch of K (otherwise, the two margin paths would branch). It follows that the ports are separated by a node that is not in the image of h, which guarantees disjointness of the associated contexts.

- Finally, for a context port u' and a forest port u'', we are dealing with a context $T.v - T.v'$, and a forest whose roots are children of a single node w. If such sets are not disjoint and neither is contained in the other, the node w must lie on the path between v and v' (excluding v'). The margin path around u' in K cannot contain other ports, which means that the set $T_{u''}$ is disjoint from this path. As v' lies on this path, $T_{u''}$ does not contain v'. It follows that $T_{u''} \subseteq Z_{u'}$.

To construct $\mathcal{Z}$ start with $\mathcal{Z} = \emptyset$ and for each port u such that T_u is not disjoint from the image of h, make sure that T_u is covered: let $\mathcal{Z}(u) = \{Z \in \mathcal{Z} \colon Z \cap Z_u \neq \emptyset\}$; if $T_u \subseteq Z$ for some $Z \in \mathcal{Z}(u)$, then T_u is already covered and we can proceed to the next port; otherwise, by the observation above, $T_{u'} \subseteq Z_u$ for all u' such that $Z_{u'} \in \mathcal{Z}(u)$, and we replace $\mathcal{Z}$ with $\big(\mathcal{Z} - \mathcal{Z}(u)\big) \cup \{Z_u\}$. Let $\mathcal{V} = \{h^{-1}(Z) \colon Z \in \mathcal{Z}\} \cup \{V_\pi - h^{-1}(\bigcup \mathcal{Z})\}$ and denote $V_u = h^{-1}(Z_u)$.

Since $h^{-1}(\bigcup \mathcal{Z})$ covers all vertices of π mapped to $\bigcup_u T_u$, for $x \in V_\pi - h^{-1}(\bigcup \mathcal{Z})$, $h(x) = w$ is a vertex from the fixed area of T, and hence has its counterpart w' in T'. Let $h'(x) = w'$. Now let us see how to define h' on $V_u \in \mathcal{V}$. Since T' is a safe extension of T, the forest/context substituted at u in T' contains a copy of Z_u, which in turn contains $h(V_u)$. We define h' on V_u by transfering h from Z_u to its copy in T'. To prove that this gives a homomorphism we need to check that no relation between vertices of π is violated. We are going to show that each relation between $x \in V_u$ and $y \notin V_u$ must be witnessed by a path visiting a *critical node* in the fixed part of T induced by K, and that in T' the witnessing paths to the critical nodes exist as well. Since the critical nodes are in the fixed part, the relations between them are the same in T and in T' and the correctness of h' follows.

Suppose first that u is a forest port. If $(y, x) \in E_c$, then $h(y) = w$, the parent of v and v'. The case $(x, y) \in E_c$ is not possible at all. Similarly, $(x, y) \notin E_n$, $(y, x) \notin E_n$, $(x, y) \notin E_d$, and edges $E_d(y, x), E_f(x, y), E_f(y, x)$ can only be witnessed by paths visiting w, v', and v respectively. Note

that w, v, v' are vertices of K and have their counterparts in T'. After transfering the image of V_u to the copy of Z_u contained in T'_u as a subforest, the child and descendant relation with w and the following-sibling relation with v, v' are preserved.

If u is a tree port, the only possible edge between $x \in V_u$ and $y \notin V_u$ is $E_d(y, x)$, and the witnessing path must visit v. Hence, the image of V_u can be transfered to the copy of Z_u contained in T'_u, without violating the existence of witnessing paths.

Finally, if u is a context port, let V_{below}, V_u, V_{above} be a partition of π's vertices into those mapped to $T.v'$, $Z_u = T.v - T.v'$, and elsewhere. The only possible edges between $x \in V_u$, $y \in V_{\text{above}}$, and $y' \in V_{\text{below}}$, are $E_d(y, x)$, $E_d(x, y')$, or $E_d(y, y')$ with the witnessing paths visiting v, v', or both of these, respectively. Consequently, we can transfer the image of V_u to the copy of Z_u contained in T'_u, preserving the existence of the witnessing paths. □

Algorithms. Using Lemma 3.3 we obtain closure under π-unions in a relatively easy way and to prove the main theorem we implement the recipe outlined in the beginning of this section.

Lemma 3.4. *Given an automaton $\mathcal{A}$, a natural number m, an $\mathcal{A}$-kind K with margins of size m, and trees $T_1, T_2, \ldots, T_n \in L(K)$ with witnessing substitutions $(T_1)_u, (T_2)_u, \ldots, (T_n)_u$, one can construct a tree $T \in L(K)$ that is a π-union of $T_1, T_2, \ldots, T_n$ for every pattern π with $\|\pi\| \leq m$. The construction can be carried out in time polynomial in $|K| \sum_{i=1}^{n} |T_i| + |K|(n+1)2^{p(\|\mathcal{A}\|)}$, for some polynomial p.*

Proof. The idea is to build a tree T that is an m-safe extension of *each* T_i. At each port u in K we substitute the concatenation of $(T_1)^m_u, (T_2)^m_u, \ldots, (T_n)^m_u$, together with some extra padding ensuring compatibility. We describe the algorithm in detail for $n = 2$. Let λ, κ be a run over K witnessing the margins of size m. For $i = 1, 2$ fix an extended run λ_i, κ_i compatible with λ, κ, witnessing that $T_i \in L(K)$.

Suppose that u is a q, q'-port with q, q' in an SCC X. Since κ_i in the roots of $(T_i)^m_u$ stays in the component X, we can construct forests I, I', I'' such that $I + (T_1)^m_u + I' + (T_2)^m_u + I''$ is compatible with u, giving an m-safe extension of T_1 and of T_2.

Let us now assume that u is a p-port with p in a branching SCC X. Then the state of λ_i in the root of $(T_i)_u$ is p and the state of λ_i in the root of $(T_i)^m_u$ is some $p_i \in X$. Since X is branching, there exists a multicontext I with two ports, u_1 and u_2, admitting a run of $\mathcal{A}$ in which the root evaluates to p if the ports are evaluated to p_1, p_2 respectively. Let $I((T_1)^m_u, (T_2)^m_u)$ be the tree obtained by substituting u_i with $(T_i)^m_u$. By construction, $I((T_1)^m_u, (T_2)^m_u)$ admits a run evaluating it to p, and can be substituted at u giving a safe extension of T_1 and T_2.

The last case is a p, p'-port u with p, p' in a non-branching SCC X. Like before, we can construct contexts I, I', I'' such that $I \cdot (T_1)^m_u \cdot I' \cdot (T_2)^m_u \cdot I''$ is compatible with u and gives a safe extension of T_1 and T_2.

Let T be the tree resulting from the substitutions above. By construction T is a safe extension of T_i for each i, and by Lemma 3.3 it is a π-union for each π of size at most m.

The sizes of forests or contexts I, I', I'' used in the construction can easily be bounded by an exponential of $\|\mathcal{A}\|$ (independent of T_1, T_2). Hence, the size of T is at most $|K|(|T_1|+|T_2|+3c)$, where $c = 2^{p(\|\mathcal{A}\|)}$ for some polynomial p. It is essential that the extensions are performed simultaneously. To see this, note that the margin contexts of some ports can contain other ports and their margins (e.g., usually, multiple p-ports have their margin contexts rooted at the same node). Performing the extensions one by one for these ports could lead to an exponential blow-up, as certain parts of the tree might be duplicated in every extension.

Similarly, merging n trees one by one using the above procedure would result in an exponential-size tree. Instead we merge them all in one go by concatenating the forests or contexts replacing u in all n trees. In this way we obtain a tree of size at most $|K|\big(|T_1|+|T_2|+\ldots+|T_n|+(n+1)c\big)$. □

Theorem 3.5. *For a mapping $\mathcal{M}$ and a source tree T one can build a solution (or determine that it does not exist) in* EXPSPACE *in general, and in* P *if the mapping is fixed.*

Proof. Let m be the maximum over the sizes of the target side patterns in $\mathcal{M}$, and let $\mathcal{A}$ be the automaton underlying the target schema. By Lemma 3.2, if there is a solution to T, it agrees with an $\mathcal{A}$-kind K with margins of size m, and height and branching at most $(2m+1)\|\mathcal{A}\|$. Note that this gives an upper bound on the size of K at most single exponential in $\|\mathcal{M}\|$ and independent of T. Checking if K is an $\mathcal{A}$-kind with margins of size m amounts to finding an accepting run witnessing the margins. The size of the run is linear in $|K|$ and its correctness can be checked in time polynomial in $|K|$ and $\|\mathcal{A}\|$. The kind only needs to use data values from T and nulls from a set $\{\bot_1, \bot_2, \ldots, \bot_{|K|}\}$ independent of T (single exponential in $\|\mathcal{M}\|$). Each null can appear more than once in K. It is intended to represent "data distinct from whatever appears in the source tree". (Formally, we partition $\mathbb{D}$ into two infinite sets, $\mathbb{D}_s$ containing data values that are allowed in the source and $\mathbb{D}_t$ containing the data values allowed only on the target side.) By checking the kinds one by one, we reduce the problem to finding a solution agreeing with a given kind K.

For each dependency $\pi(\bar{x}), \eta(\bar{x}) \longrightarrow \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ in $\mathcal{M}$ and each tuple $\bar{a}$ such that $\eta(\bar{a})$ holds and $T \models \pi(\bar{a})$, we need to find a tree $S \in L(K)$ and a tuple $\bar{b}$ satisfying $\eta'(\bar{a}, \bar{b})$ such that $S \models \pi'(\bar{a}, \bar{b})$. The size of S does not depend on T. Indeed, a pumping argument similar to the one in Lemma 2.2 shows that each part of S substituting a port of K can have size single exponential in $\|\mathcal{M}\|$ and independent of T. In consequence, we can find S by exhaustive search.

If for some $\bar{a}$ the search fails, there is no solution to T agreeing with K. If all partial solutions have been found, we can merge them into a single full solution in EXP, by Lemma 3.4.

For a fixed mapping, the number of possible kinds is polynomial in $|T|$ and their size is bounded by a constant. For each kind finding and merging partial solutions takes polynomial time. Hence, the whole procedure is in P. □

4. Solution existence

The solution built by the algorithm described in Sect. 3 is polynomial-size if the mapping is fixed and exponential-size in general. These bounds are tight: the smallest tree conforming to the target schema might need to be exponential, and a simple copying rule $r/\!/\_(x) \longrightarrow r/\!/\_(x)$ makes the solution at least as large as the source tree. This however does not give matching lower bounds for the combined complexity of solution building.

To understand the complexity of the solution building problem better, we consider *solution existence*, a decision version of the problem.

PROBLEM:	SOLEX
INPUT:	mapping $\mathcal{M}$, source tree T
QUESTION:	Is $\mathcal{M}(T)$ nonempty?

Clearly, finding a solution is at least as hard as deciding if it exists. The following theorem implies that an algorithm for solution building with exponential combined complexity is unlikely to exist.

Theorem 4.1. SolEx *is* NEXP*-complete. The problem is* NEXP*-hard even for mappings using only the child axis, wildcard, equality, and non-recursive DTDs without disjunction.*

Proof. To get the upper bound proceed just like in the proof of Theorem 3.5, only instead of examining every possible kind K, choose it nondeterministically together with a witnessing run. Then, for each dependency $\pi(\bar{x}), \eta(\bar{x}) \longrightarrow \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ in $\mathcal{M}$ and each tuple $\bar{a}$ satisfying $\eta(\bar{a})$ such that $T \models \pi(\bar{a})$, we need to check if there exists $T' \in L(K)$ and a tuple $\bar{b}$ satisfying $\eta'(\bar{a}, \bar{b})$ such that $T' \models \pi'(\bar{a}, \bar{b})$. By Lemma 3.1 this can be done nondeterministically in polynomial time.

To get the lower bound we give a reduction from the following NEXP-complete problem: given a nondeterministic Turing machine M and $n \in \mathbb{N}$, does M accept the empty word in at most 2^n steps? The idea of the reduction is to encode the run of the machine in the target tree. The run will be encoded as a sequence of 2^n configurations of length 2^n. The machine M is stored in the source tree, except the (specially preprocessed) transition relation, which is encoded in the target tree. The source tree is also used to address the configurations and their cells.

Let M have the tape alphabet A with the blank symbol $\flat \in A$ and the states $q_0, q_1, \ldots, q_f$. W.l.o.g. we assume that q_f is the only final accepting state. The *extended transition relation* of M, denoted $\hat{\delta}$, describes possible transitions in a window of three consecutive tape cells. Formally, $\hat{\delta} \subseteq (\{q_0, q_1, \ldots, q_f, \bot\} \times \hat{A})^6$, where $\hat{A}$ is the set of *decorated tape symbols*, defined as $\hat{A} = \{s, s^{\triangleright}, s^{\triangleleft} \colon s \in A\}$. The symbol $\bot$ means "the head is elsewhere", $\triangleright$ marks the beginning of the tape, $\triangleleft$ marks the end of the tape (at position 2^n), and $(p_1, \sigma_1, p_2, \sigma_2, \ldots, p_6, \sigma_6) \in \hat{\delta}$ iff at most one of p_1, p_2, p_3 is not equal to $\bot$, and $p_4\sigma_4 p_5\sigma_5 p_6\sigma_6$ is obtained from $p_1\sigma_1 p_2\sigma_2 p_3\sigma_3$ by performing a transition of M. Note that $p_1 = p_2 = p_3 = \bot$ and $p_4 \neq \bot$ is possible as long as σ_4 is not marked with $\triangleright$. Note that $\hat{\delta}$ can be computed in P. The constant d used in the target DTD is equal to $|\hat{\delta}|$.

The source DTD is given as

$$\begin{aligned} r &\to \mathit{zero}\ \mathit{one}\ q_0\, q_1 \cdots q_f \perp \tau_1\, \tau_2 \cdots \tau_m \\ \mathit{zero}, \mathit{one} &\to \mathit{bit} \end{aligned}$$

where $\hat{A} = \{\tau_1, \tau_2, \ldots, \tau_m\}$. The target DTD is given as

$$\begin{aligned} r &\to a_1\, b_1\, \mathit{tr}_1\, \mathit{tr}_2 \cdots \mathit{tr}_d \\ a_j, b_j &\to a_{j+1}\, b_{j+1} \\ a_{2n}, b_{2n} &\to \mathit{cell} \\ \mathit{tr}_i &\to \mathit{st}_1\, \mathit{sym}_1\, \mathit{st}_2\, \mathit{sym}_2 \cdots \mathit{st}_6\, \mathit{sym}_6 \\ \mathit{cell} &\to c_1\, c_2 \cdots c_{2n}\, \mathit{st}\, \mathit{sym} \end{aligned}$$

for $i = 1, 2, \ldots, d$, $j = 1, 2, \ldots, 2n-1$, and $d = |\hat{\delta}|$. The subtrees rooted at *cell* nodes store in binary a configuration number and a cell number, as well as a state and a decorated tape symbol. The subtrees rooted at tr_i nodes store $\hat{\delta}$.

The source tree T is any tree conforming to the source schema storing a different data value in each node. The children of the root store data values encoding the states and tape symbols used in $\hat{\delta}$. To ensure that $\hat{\delta}$ is stored properly on the target side, for each $(q_{i_1}, \tau_{j_1}, q_{i_2}, \tau_{j_2}, \ldots, q_{i_6}, \tau_{j_6}) \in \hat{\delta}$ add a dependency

$$r[q_{i_1}(u_1), \tau_{j_1}(v_1), q_{i_2}(u_2), \tau_{j_2}(v_2), \ldots, q_{i_6}(u_6), \tau_{j_6}(v_6)] \longrightarrow r/\mathit{Tr}(\bar{u}, \bar{v})\,,$$

where $Tr(\bar{u},\bar{v})$ stands for $\_[\bigwedge_{i=1}^{6} st_i(u_i) \wedge sym_i(v_i)]$. Note that d different dependencies are introduced, so each *tr* node in the target tree contains a tuple from $\hat{\delta}$.

The data values stored in T in the two *bit* nodes are used to address the configurations and their cells. This is done by means of three auxiliary patterns, $First(\bar{x})$, $Last(\bar{x})$, and $Succ(\bar{x},\bar{y})$ with $\bar{x} = x_1, x_2, \ldots, x_n$, $\bar{y} = y_1, y_2, \ldots, y_n$, which roughly speaking implement a binary counter over n bits. In the auxiliary patterns we use disjunction, but since we are only going to apply them on the source side of dependencies, disjunction can be easily eliminated at the cost of multiplying dependencies. Let

$$
\begin{aligned}
First(\bar{x}) &= zero\big[bit(x_1), bit(x_2), \ldots, bit(x_n)\big]\,,\\
Last(\bar{x}) &= one\big[bit(x_1), bit(x_2), \ldots, bit(x_n)\big]\,,\\
Succ(\bar{x},\bar{y}) &= \bigvee_{i=1}^{n}\left(\bigwedge_{j=1}^{i-1} \_\big[bit(x_j), bit(y_j)\big],\ zero/bit(x_i) \wedge one/bit(y_i),\ \bigwedge_{j=i+1}^{n} one/bit(x_j) \wedge zero/bit(y_j)\right).
\end{aligned}
$$

Using the auxiliary patterns we ensure that the target tree encodes an accepting run of M. In the first configuration the tape is empty and the head state q_0 is over the first cell,

$$
\begin{aligned}
r[First(\bar{x}), First(\bar{y}), q_0(u), \flat^{\triangleright}(v)] &\longrightarrow r /\!\!/ Cell(\bar{x},\bar{y},u,v)\,,\\
r[First(\bar{x}), Succ(\bar{z}_1,\bar{y}), Succ(\bar{y},\bar{z}_2), \bot(u), \flat(v)] &\longrightarrow r /\!\!/ Cell(\bar{x},\bar{y},u,v)\,,\\
r[First(\bar{x}), Last(\bar{y}), \bot(u), \flat^{\triangleleft}(v)] &\longrightarrow r /\!\!/ Cell(\bar{x},\bar{y},u,v)\,,
\end{aligned}
$$

where $Cell(\bar{x},\bar{y},u,v)$ stands for $cell[\bigwedge_{i=1}^{n} c_i(x_i) \wedge c_{n+i}(y_i), st(u), sym(v)]$. Note that the valuations of $\bar{y}$ correspond to all numbers from 0 to $2^n - 1$, and thus the content of each cell of the tape is specified.

The correctness of transitions is ensured by

$$
r\big[Succ(\bar{x}_0,\bar{x}_1),\ Succ(\bar{y}_1,\bar{y}_2), Succ(\bar{y}_2,\bar{y}_3)\big] \longrightarrow r\Big[Tr(\bar{u},\bar{v}), \bigwedge_{i,j} /\!\!/ Cell(\bar{x}_i,\bar{y}_j,u_{3i+j},v_{3i+j})\Big].
$$

Again, $\bar{x}_0, \bar{x}_1$ range over $k, k+1$ with $0 \leq k < 2^n - 1$, and y_1, y_2, y_3 range over $\ell, \ell+1, \ell+2$ with $0 \leq \ell < 2^n - 2$, so the evolution of each three consecutive cells is checked for every step.

Finally, the machine needs to reach the accepting state (w.l.o.g. we assume that the accepting state is looping),

$$
r/q_f(u) \longrightarrow r /\!\!/ Cell(\bar{x},\bar{y},u,v)\,.
$$

Note that each occurrence of $/\!\!/$ above can be replaced with a sequence of $\_$ and $/$ symbols of suitable length.

It is straightforward to verify that M has an accepting computation of length at most 2^n if and only if T has a solution with respect to the mapping defined above. □

In the reduction above, equalities can be eliminated from the target side by introducing a pattern $Succ_3(\bar{x},\bar{y},\bar{z})$, which expresses that $\bar{x},\bar{y},\bar{z}$ represent three subsequent values of the counter. We use this idea later in the proof of Theorem 5.3.

Finally, let us examine the data complexity of solution existence, i.e., the complexity of the following problem:

PROBLEM:	SOLEX($\mathcal{M}$)
INPUT:	Source tree T
QUESTION:	Is $\mathcal{M}(T)$ nonempty?

where the mapping $\mathcal{M}$ is fixed.

Theorem 4.2. SOLEX($\mathcal{M}$) *is in* LOGSPACE.

Proof. The target kind K that was nondeterministically chosen in the proof of Theorem 4.1 is now constant size and takes data values from a set A of linear size. In consequence, it can be stored in logarithmic space. Instead of guessing it, we can iterate over all possibilities. It remains to check in LOGSPACE if there is a solution of a given kind K.

By Lemma 3.4 it is enough to check if for each dependency $\pi(\bar{x}), \eta(\bar{x}) \to \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ in $\mathcal{M}$ and each $\bar{a}$ satisfying $\eta(\bar{a})$ such that $T \models \pi(\bar{a})$ there is a tuple $\bar{b}$ satisfying $\eta'(\bar{a}, \bar{b})$ and a tree T' of the kind K such that $T' \models \pi'(\bar{a}, \bar{b})$. Again, we can iterate over all dependencies and tuples $\bar{a}$, $\bar{b}$ with entries from A. For a fixed dependency and fixed $\bar{a}, \bar{b}$, by Proposition 2.1, $T \models \pi(\bar{a})$ can be checked in LOGSPACE and the remaining tests can be carried out in constant time, modulo suitable encoding of data values in K, $\bar{a}$, and $\bar{b}$, which can be prepared in LOGSPACE. □

5. Absolute consistency

As we mentioned in the introduction, a schema mapping $\mathcal{M} = \langle \mathcal{S}_s, \mathcal{S}_t, \Sigma \rangle$ is called *absolutely consistent* if every source tree has a solution, i.e., $\mathcal{M}(T) \neq \emptyset$ for every $T \in L(\mathcal{S}_s)$. We are interested in the following decision problem:

PROBLEM:	ABCONS
INPUT:	Mapping $\mathcal{M} = \langle \mathcal{S}_s, \mathcal{S}_t, \Sigma \rangle$
QUESTION:	Is $\mathcal{M}$ absolutely consistent?

Assume for a while that the mapping $\mathcal{M}$ contains a single dependency $\pi(\bar{x}) \to \pi'(\bar{x}, \bar{y})$. The logical structure of the condition we need to check is: *for every* source tree T *there exists* a target tree T' such that *for every* $\bar{a}$ satisfying $T \models \pi(\bar{a})$ *there exists* $\bar{b}$ satisfying $T' \models \pi'(\bar{a}, \bar{b})$. To turn this into an algorithm we would need to show a bound on the size of trees that need to be considered.

Instead, we will try to change the order of quantifiers to the following: "whenever $\pi(\bar{a})$ is satisfiable in a source tree, there exists $\bar{b}$ such that $\pi'(\bar{a}, \bar{b})$ is satisfiable in a target tree". The modified condition can be checked easily in Π_2P. Indeed, what really matters is the equality type of $\bar{a}$ and $\bar{b}$, so it is enough to chose their entries from a fixed set of linear size. Furthermore, one does not need to guess the source and target trees explicitly, it is enough to witness their existence. By Lemma 2.2, there exists a polynomial witness.

To justify the reordering of the quantifiers, we would need to know that the set of target trees is closed under π'-*unions*. As we have learned in Sect. 3, this need not be true in general, but is guaranteed within the set of trees agreeing with some kind. A single kind usually cannot provide solutions to all source trees. For instance, if $\mathcal{M} = (\{r \to a^*b^*\}, \{r \to a\, b^*\}, \{\pi(x) \to \pi'(x)\})$ with $\pi(x) = \pi'(x)$ saying "there is an a-node storing x whose next sibling is a b-node", and the target kind fixes the value in the a node to some d, a solution exists only for source trees that store the same d in the last a-node. Thus, source documents have to be partitioned into subsets admitting

solutions of a single kind. It turns out that elements of this partition can be defined with kinds as well, because each set of source documents closed under π-unions admits solutions of a single kind.

Based on this we reformulate the absolute consistency condition as "for every source kind K there exists a target kind K' such that whenever $\pi(\bar{a})$ is satisfiable in a tree of kind K, there exists $\bar{b}$ such that $\pi'(\bar{a}, \bar{b})$ is satisfiable in a tree of kind K'", which ultimately leads to a Π_2EXP-algorithm.

Theorem 5.1. AbCons *is in* Π_2EXP.

Proof. We present the algorithm as a two-round game between two players, $\forall$ and $\exists$. In each round, $\forall$ moves first. Moves are made by the choice of an object of size exponential in $\|\mathcal{M}\|$ during the first round, and polynomial in $\|\mathcal{M}\|$ during the second round. The winning condition, a polynomial time property of the chosen objects, is defined so that $\exists$ wins exactly if $\mathcal{M}$ is absolutely consistent. In the first round, $\forall$ states what kind of tree is a counterexample to absolute consistency, while $\exists$ chooses the kind of tree that gives solutions to the purported counterexamples. In the second round, $\forall$ picks a tree of the declared kind and a tuple of data values witnessing that the solutions fail, and $\exists$ tries to respond with a tree and a tuple that would prove $\forall$ wrong.

Let the source and target schemas be given by the automata $\mathcal{A}_s$, $\mathcal{A}_t$, and let m be the maximum over the sizes of all patterns used in $\mathcal{M}$.

In the first round $\forall$ plays an $\mathcal{A}_s$-kind $K_\forall$ with margins of size m, and height and branching bounded by $(2m+1)\|\mathcal{A}_s\|$, together with a witnessing run. The data values used in K are to represent an equality type, so it is enough to choose them from $\{1, 2, \ldots, |K_\forall|\}$. The response $K_\exists$ of $\exists$ is similar except that $\mathcal{A}_s$ is replaced by $\mathcal{A}_t$, and some of the nodes can store nulls taken from a fixed set $\{\bot_1, \bot_2, \ldots, \bot_{|K_\exists|}\}$.

In the second round, $\forall$ chooses a dependency $\pi(\bar{x}), \eta(\bar{x}) \longrightarrow \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y})$ and a tuple $\bar{a}$ (*without* nulls) satisfying $\eta(\bar{a})$ together with a polynomial object witnessing that $\pi(\bar{a})$ can be realized in a tree agreeing with $K_\forall$ (Lemma 3.1). $\exists$ then responds with a tuple $\bar{b}$ (possibly *including* nulls) satisfying $\eta'(\bar{a}, \bar{b})$ and a polynomial witness that $\pi'(\bar{a}, \bar{b})$ can be realized in a tree agreeing with $K_\exists$.

A player loses if he fails to make a move complying with the rules. If all moves are made, $\exists$ wins.

It remains to show that $\exists$ has a winning strategy if and only if $\mathcal{M}$ is absolutely consistent.

If $\mathcal{M}$ is not absolutely consistent, $\forall$'s strategy in the first round is to choose a data tree T for which no solution exists and play $K_\forall$ such that $T \in L(K_\forall)$ (see Lemma 3.2).

If $\mathcal{M}$ is absolutely consistent, $\exists$'s strategy in the first round is to choose $K_\exists$ so that for every tree agreeing with $K_\forall$ there exists a solution agreeing with $K_\exists$. If such a $K_\exists$ cannot be produced, then there exists a tree agreeing with $K_\forall$ for which there is no solution at all, contradicting absolute consistency. To see this, reason as follows. For each possible response K to $K_\forall$, let T_K be a tree agreeing with $K_\forall$ for which there is no solution agreeing with K. By Lemma 3.4 there is a tree $T \in L(K_\forall)$ such that $T \models \pi(\bar{a})$ whenever one of the T_K's satisfies $\pi(\bar{a})$ for every source side pattern π. Since every $T' \in L(\mathcal{A}_t)$ agrees with one of the K's (Lemma 3.2), there is no solution for T.

In the second round, if $\mathcal{M}$ is not absolutely consistent, there is some T agreeing with $K_\forall$ for which there is no solution. $\forall$'s strategy now is to choose a dependency δ, say $\pi_\delta(\bar{x}), \eta_\delta(\bar{x}) \longrightarrow \pi'_\delta(\bar{x}, \bar{y}), \eta'_\delta(\bar{x}, \bar{y})$, and a tuple $\bar{a}$ satisfying $\eta_\delta(\bar{a})$ such that $T \models \pi_\delta(\bar{a})$, but there is no $\bar{b}$ satisfying $\eta'_\delta(\bar{a}, \bar{b})$ such that $\pi'_\delta(\bar{a}, \bar{b})$ can be realized in a tree agreeing with $K_\exists$. Some suitable δ and $\bar{a}$ exist, as otherwise a solution for T could be obtained as follows. Assume that for each δ and $\bar{a}$ as above there is a tree $T_{\delta,\bar{a}}$ in $L(K_\exists)$ such that $T_{\delta,\bar{a}} \models \pi'_\delta(\bar{a}, \bar{b})$ for some $\bar{b}$ satisfying $\eta'_\delta(\bar{a}, \bar{b})$. Then by

Lemma 3.4 there is a tree $T' \in L(K_\exists)$ that is a π''-union of $T_{\delta,\bar{a}}$'s for every target side pattern π''. Clearly, T' is a solution for T.

If $\mathcal{M}$ is absolutely consistent, then whatever δ and $\bar{a}$ was played, $\pi'_\delta(\bar{a}, \bar{b})$ can be realized in a tree from $L(K_\exists)$ for some $\bar{b}$ satisfying $\eta'_\delta(\bar{a}, \bar{b})$. $\exists$'s strategy is to choose suitable $\bar{b}$ and a witness. □

To give a lower bound for the complexity of AbCons let us consider the following problem:

Problem:	2^n-universality
Input:	Nondeterministic Turing machine M, number n in unary
Question:	Does M accept every word of length 2^n in at most 2^n steps?

The problem is obviously in Π_2EXP. It can be also shown that it is hard for this class.

Lemma 5.2. 2^n-universality *is* Π_2EXP*-complete.*

Proof. Take a Π_2EXP language L. There is a nondeterministic machine M and $k \leq l$ such that $x \in L$ iff for all y such that $|y| \leq 2^{|x|^k}$, M accepts (x, y) (M always stops after at most $2^{|x|^l}$ steps). Let $M_{x,k}$ write x on the tape, mark the first $2^{|x|^k}$ positions of the input as relevant, and simulate M on x and the relevant part of the input. Since $k \leq l$, the machine stops after at most $2^{C|x|^l}$ steps for some C independent of x. Hence, $x \in L$ iff $(M_{x,k}, 1^{C|x|^l})$ is a positive instance of 2^n-universality. □

We give a reduction from 2^n-universality to AbCons for a very restricted class of mappings.

Theorem 5.3. AbCons *is* Π_2EXP*-hard even for mappings using only child axis, wildcard, equality, and non-recursive DTDs.*

Proof. This reduction is very similar to the one in the proof of Theorem 4.1. The possible source trees will encode the input words, and the solutions will encode accepting runs as sequences of configurations. The main difficulty is that we need to copy the exponential input word to the target tree by means of polynomially many dependencies. For this purpose we use a modified mechanism of addressing the configurations and their cells, based on a linear order of length 2^n explicitly stored in the source tree. Note that this could not be done in Theorem 4.1, as the source tree was part of the input and thus had to be polynomial. Instead, the order was simulated with n-bit sequences representing in binary the elements of the order.

Let an instance of 2^n-universality be $n \in \mathbb{N}$ and a Turing machine M with the tape alphabet A and the blank symbol $\flat \in A$, the state space $q_0, q_1, \ldots, q_f$ and the extended transition relation $\hat{\delta}$ (see the proof of Theorem 4.1 for details). W.l.o.g. we assume that q_f is the only final accepting state.

The source DTD is given as

$$\begin{aligned} r &\to \mathit{ord}\, q_0\, q_1\, \ldots\, q_f \perp \tau_1\, \tau_2 \cdots \tau_m \\ \mathit{ord} &\to a_1\, b_1 \\ a_i, b_i &\to a_{i+1}\, b_{i+1} \\ a_n, b_n &\to c \\ c &\to t_1 \,|\, t_2 \,|\, \cdots \,|\, t_k \end{aligned}$$

where $t_1, t_2, \ldots, t_k$ are the tape symbols except $\flat$, and $\tau_1, \tau_2, \ldots, \tau_m$ are the decorated tape symbols, i.e., elements of $\hat{A} = \{s, s^{\triangleright}, s^{\triangleleft} \mid s \in A\}$, $i = 1, 2, \ldots, n-1$ and the labels $q_0, q_1, \ldots, q_f, \bot$, $\tau_1, \tau_2, \ldots, \tau_m, c$ have a single attribute. The target DTD is given as

$$\begin{aligned} r &\to a_1\, b_1\, tr_1\, tr_2 \cdots tr_d \\ a_j, b_j &\to a_{j+1}\, b_{j+1} \\ a_{2n}, b_{2n} &\to \mathit{cell} \\ tr_i &\to st_1\, sym_1\, st_2\, sym_2 \cdots st_6\, sym_6 \\ \mathit{cell} &\to \mathit{confnum}\ \mathit{cellnum}\ \mathit{st}\ \mathit{sym} \end{aligned}$$

where $i = 1, 2, \ldots, d$, $j = 1, 2, \ldots, 2n-1$. Under the tr_i nodes we store $\hat{\delta}$, the extended transition relation of M, and $d = |\hat{\delta}|$. Under the *cell* nodes we store a configuration number, a cell number, a state, and a decorated tape symbol.

Assume for a while that we only need to handle source trees in which all data values are distinct. Correctness of the encoding of $\hat{\delta}$ is ensured with dependencies

$$r[q_{i_1}(u_1), \tau_{j_1}(v_1), q_{i_2}(u_2), \tau_{j_2}(v_2), \ldots, q_{i_6}(u_6), \tau_{j_6}(v_6)] \longrightarrow r / \mathit{Tr}(\bar{u}, \bar{v})\,,$$

for each $(q_{i_1}, \tau_{j_1}, q_{i_2}, \tau_{j_2}, \ldots, q_{i_6}, \tau_{j_6}) \in \hat{\delta}$, with $\mathit{Tr}(\bar{u}, \bar{v})$ standing for $\_[\bigwedge_{i=1}^{6} st_i(u_i) \wedge sym_i(v_i)]$.

The addressing mechanism is based on the data values stored in the c-nodes, encoding a linear order of length 2^n. With every c-node v we associate the sequence of a's and b's on the path leading from the root to v. This sequence is interpreted as a binary number (a read as 0, b read as 1), which is the position of v in the order. The auxiliary patterns take the following form:

$$\begin{aligned} \mathit{First}(x) &= \mathit{ord}/a_1/a_2/\cdots/a_n/c(x)\,, \\ \mathit{Last}(x) &= \mathit{ord}/b_1/b_2/\cdots/b_n/c(x)\,, \\ \mathit{Succ}(x,y) &= \bigvee_{i=1}^{n} /\!\!/\_\big[a_i/b_{i+1}/b_{i+2}/\cdots/b_n/c(x),\ b_i/a_{i+1}/a_{i+2}/\cdots/a_n/c(y)\big]\,, \\ \mathit{Succ}_3(x,y,z) &= \bigvee_{i=1}^{n-1} /\!\!/\_\Big[a_i/b_{i+1}/b_{i+2}/\cdots/b_{n-1}\big[a_n/c(x), b_n/c(y)\big],\ b_i/a_{i+1}/a_{i+2}/\cdots/a_n/c(z)\Big] \ \vee \\ &\quad \vee \bigvee_{i=1}^{n-1} /\!\!/\_\Big[a_i/b_{i+1}/b_{i+2}/\cdots/b_n/c(x),\ b_i/a_{i+1}/a_{i+2}/\cdots/a_{n-1}\big[a_n/c(y), b_n/c(z)\big]\Big]\,. \end{aligned}$$

We use the pattern $\mathit{Succ}_3(x,y,z)$ instead of $\mathit{Succ}(x,y), \mathit{Succ}(y,z)$ to eliminate equality from the source side. Each occurrence of $/\!\!/$ above can be replaced with a sequence $\_/\_/\ldots/\_/$ of corresponding length (at most $2n+1$). Disjunction can be eliminated since we only use the auxiliary patterns on the source side. All this increases the size of the output at most by a linear factor.

To build the first configuration we copy the input word encoded in the labels of the c-nodes' children with the head over the first cell, in the state q_0:

$$\begin{aligned} r\big[\mathit{First}(x), \mathit{First}(y)/s, q_0(u), s^{\triangleright}(v)\big] &\longrightarrow r /\!\!/ \mathit{Cell}(x,y,u,v)\,, \\ r\big[\mathit{First}(x), \mathit{Succ}_3(z_1,y,z_2)/s, \bot(u), s(v)\big] &\longrightarrow r /\!\!/ \mathit{Cell}(x,y,u,v)\,, \\ r\big[\mathit{First}(x), \mathit{Last}(y)/s, \bot(u), s^{\triangleleft}(v)\big] &\longrightarrow r /\!\!/ \mathit{Cell}(x,y,u,v)\,, \end{aligned}$$

where s ranges over $A - \{\flat\}$, $Cell(x, y, u, v)$ stands for $cell[confnum(x), cellnum(y), st(u), sym(v)]$, and by $Succ_3(z_1, y, z_2)/s$ we mean $Succ_3(z_1, y, z_2)$ with $c(y)$ replaced with $c(y)/s$.

To make sure we encode a correct accepting run, we add

$$r\big[Succ(x_0, x_1),\ Succ_3(y_1, y_2, y_3)\big] \longrightarrow r\Big[\_/\, Tr(\bar{u}, \bar{v}), \bigwedge_{i,j} /\!\!/ Cell(x_i, y_j, u_{3i+j}, v_{3i+j})\Big],$$

$$r/q_f(u) \longrightarrow r /\!\!/ Cell(x, y, u, v)\,.$$

We claim that the mapping we have just defined is absolutely consistent iff the answer to 2^n-UNIVERSALITY is "yes". Assume that the mapping is absolutely consistent. Every input word w can be encoded in a source tree using distinct data values. An inductive argument shows that a solution to such a tree encodes an accepting run of M on w. Conversely, if the answer is "yes", for each source tree S using distinct data values, a solution is obtained from the run of M on the word encoded in the sequence of t_i-leaves of S. What if S uses some data values more than once? For a function $h : \mathbb{D} \to \mathbb{D}$ and a tree U, let $h(U)$ be the tree obtained from U by replacing each data value a with $h(a)$. Now, let S' be a tree with the structure identical as S, but using distinct data values, and let h be a function on data values such that $h(S') = S$. By the previously considered case, there is a solution T' for S'. Since our mapping does not use inequality on the target side, nor equality on the source side, $h(T')$ is a solution for $h(S') = S$. □

In the reduction above we can remove disjunction from the DTDs at the cost of allowing inequality or next-sibling axis on the source side. Without disjunction, inequality and sibling order one can prove NEXP-hardness [1].

6. Bounded depth mappings

We have seen that absolute consistency is highly untractable even for patterns using only vertical axes and non-recursive DTDs with very simple productions. In this section we show that the complexity can be lowered substantially if the height of trees is bounded by a constant. We say that a mapping $\mathcal{M}$ *has depth at most* d if the source and target schema only admit trees of height at most d.

Proposition 6.1. ABCONS *for mappings of bounded depth is in* Π_4P.

Proof. We claim that the general algorithm presented in Theorem 5.1 has the desired complexity for mappings of bounded depth.

Assume that an automaton $\mathcal{A}$ only accepts trees of height at most d and let K be an $\mathcal{A}$-kind. Obviously K's height is at most d. Moreover, K contains no vertical ports, as otherwise arbitrarily high trees would agree with K.

It follows that, for a bounded depth mapping, the kinds played in the first round have polynomial branching and bounded depth, which means they are polynomial. In the second round, the objects played are polynomial in the size of those from the first round. As the verification of the moves is polynomial, this gives a Π_4P-algorithm. □

A small modification of our techniques makes it possible to prove Proposition 6.1 for a more general definition of boundedness: $\mathcal{M}$ *has depth at most* d if every pattern it uses can only be realized within the initial d levels of every tree conforming to the schema. This includes mappings

based on patterns starting at the root that use neither descendant nor child-paths of length greater than d.

We next show that absolute consistency is Π_4P-hard even for schema mappings of depth 1.

Proposition 6.2. AbCons *is* Π_4P*-hard for bounded-depth mappings, even for depth 1 mappings using only child and next-sibling axes, and equality.*

Proof. We provide a reduction from Tautology for Π_4 quantified propositional formulas. Let

$$\varphi = \forall x_1, x_2, \ldots, x_n \, \exists y_1, y_2, \ldots, y_n \, \forall u_1, u_2, \ldots, u_n \, \exists v_1, v_2, \ldots, v_n \bigwedge_{i=1}^{m} X_i \vee Y_i \vee Z_i$$

with $X_i, Y_i, Z_i \in \{x_j, y_j, u_j, v_j, \bar{x}_j, \bar{y}_j, \bar{u}_j, \bar{v}_j \mid j = 1, \ldots, n\}$. The source schema $\mathcal{S}_s$ is given as

$$r \to (a_1|a_1')(a_2|a_2') \ldots (a_n|a_n')ee \,,$$

and the target schema $\mathcal{S}_t$ is given as

$$r \to eeea_1a_1a_2a_2 \ldots a_na_nb_1b_1b_2b_2 \ldots b_nb_n(g_1g_2g_3)^7 \,.$$

The source tree encodes a valuation of $x_1, x_2, \ldots, x_n$: a_i means that x_i is *true*, a_i' means it is *false*. In e-positions we store values representing *true* and *false*. On the target side, we want to keep a copy of the valuation of x_i's and a guessed valuation of y_i's, except this time we use a different coding. The first position labelled with a_i stores the value of x_i, *true* or *false*, and the following position stores the value of the negation of x_i. Similarly, b_i's store values of y_i. We also want two copies of *true* and a copy of *false* in the target tree, arranged so as to enable nondeterministic choice between a pair (*true*, *false*) or (*false*, *true*), as well as all triples with at least one entry *true*, which will help us to check that each clause of φ is satisfied.

Let us now describe the dependencies. First we make sure that values representing true and false are copied properly,

$$r\big[e(x) \to e(y)\big] \longrightarrow r\big[e(x) \to e(y) \to e(x)\big] \,,$$

for each i translate the a_i/a_i' coding of values of x_i into true/false coding,

$$r\big[a_i, e(t) \to e(f)\big] \longrightarrow r\big[a_i(t) \to a_i(f)\big] \,,$$
$$r\big[a_i', e(t) \to e(f)\big] \longrightarrow r\big[a_i(f) \to a_i(t)\big] \,,$$

and enforce in the target tree all triples with at least one entry *true*,

$$r\big[e(t) \to e(f)\big] \longrightarrow r\big[g_1(f) \to g_2(f) \to g_3(t),\ g_1(f) \to g_2(t) \to g_3(f),\ \ldots,\ g_1(t) \to g_2(t) \to g_3(t)\big] \,.$$

Next, we guess a value of y_i for each i,

$$r\big[e(t) \to e(f)\big] \longrightarrow r\big[b_i(t), b_i(f)\big] \,,$$

and ensure that it makes the internal Π_2 part of φ true:

$$r\Big[\bigwedge_{i=1}^{n} e(u_i)\Big] \longrightarrow r\Big[\bigwedge_{i=1}^{n} e(u_i) \to e(\bar{u}_i), \bigwedge_{i=1}^{n} e(v_i) \to e(\bar{v}_i), \bigwedge_{i=1}^{n} a_i(x_i) \to a_i(\bar{x}_i), \bigwedge_{i=1}^{n} b_i(y_i) \to b_i(\bar{y}_i), \bigwedge_{j=1}^{m} g_1(X_j) \to g_2(Y_j) \to g_3(Z_j)\Big]$$

(the literals X_j, Y_j, Z_j are taken from φ).

The obtained mapping is absolutely consistent iff φ is a tautology. Indeed, if the mapping is absolutely consistent, it has a solution for each source tree that uses two different data values in the e-positions. By construction, such trees have solutions iff φ is a tautology. If the data values in the e-positions are equal, the dependencies are satisfied trivially. □

In the reduction above, disjunction can be eliminated from the source schema at a cost of allowing data comparisons on the source side. To achieve this, replace $(a_i|a'_i)$ with $a_i a_i$, and encode the truth value as (in)equality of the two data values.

We finish this section with a proposition determining the complexity of the solutions existence problem and the solution building problem for bounded-depth mappings.

Proposition 6.3. *For bounded-depth mappings, solution building is in* EXP *and* SolEx *is* Σ_3P*-complete.*

Proof. The complexity analysis in Proposition 6.1 shows that the target kinds tested in the algorithm from Theorem 3.5 are polynomial-size. Hence, the algorithm runs in EXP.

A Σ_3P algorithm for SolEx is similar to the absolute consistency algorithm from Theorem 5.1, except that the first move of $\forall$ is skipped and the role of $K_\forall$ is played by the given source tree T. Correctness of the modified algorithm follows from the correctness of the original one, and the complexity analysis carries over from Proposition 6.1.

For the lower bound for SolEx we modify the reduction from Proposition 6.2 to obtain a reduction from Tautology for Σ_3 quantified propositional formulas. Take a formula

$$\varphi = \exists y_1, y_2, \ldots, y_n \, \forall u_1, u_2, \ldots, u_n \, \exists v_1, v_2, \ldots, v_n \bigwedge_{i=1}^{m} X_i \vee Y_i \vee Z_i$$

with $X_i, Y_i, Z_i \in \{y_j, u_j, v_j, \bar{y}_j, \bar{u}_j, \bar{v}_j \mid j = 1, \ldots, n\}$. The corresponding mapping is obtained by removing each reference to a_i and a'_i from the mapping in the proof of Proposition 6.2. Thus, the source schema is just $r \to ee$ and the target schema is $r \to eeeb_1b_1b_2b_2 \ldots b_nb_n(g_1g_2g_3)^7$. The second and third dependencies are removed, and in the last dependency we remove the subpatterns $a_i(x_i) \to a_i(\bar{x}_i)$. The resulting mapping has a solution for the tree $r\big[e(1), e(0)\big]$ if and only if the formula φ holds true. □

Note that the solution building algorithm can be implemented in polynomial working space (excluding the space taken by the output).

7. Regular schema mappings

In the previous sections, we used patterns to describe π and π' in the dependencies. In this section, we study a more general form of schema mappings, where instead of a pattern just talking about labels and the child and descendant relations, there a formula monadic second-order logic (MSO). We prove that for this richer language, all the problems are still decidable. We only establish decidability. The questions of complexity, which we do not address here, are most interesting when the queries are given not by MSO formulas, but by automata. If the input contains MSO formulas, then any algorithm will be necessarily be nonelementary, because satisfiability for MSO formulas is nonelementary.

The general setup is similar to the one in the previous sections. A single source-to-target dependency is also an implication of the form

$$\pi(\bar{x}), \eta(\bar{x}) \ \longrightarrow \ \pi'(\bar{x}, \bar{y}), \eta'(\bar{x}, \bar{y}).$$

The formulas η and η' are boolean combinations of equality constraints on their arguments (this is slightly more general than before, since previously we had conjunctions of equalities and inequalities). The main generalization concerns π and π'. Previously, the tuples of data values selected by π and π' came from patterns. In this section, we allow a richer syntax for π and π', where formulas of MSO are used. Below, in Section 7.1, we give a precise definition of the problem, and introduce some of the notation that we use. Then, in Section 7.2, we describe our proof strategy. The proof itself is in Sections 8 and 9.

7.1. Some notation and the problem statement

We are now interested in a more general form of dependencies, where formulas of MSO are used to select tuples of nodes, instead of tree patterns. For such formulas, we use the name *node-selecting queries*; a tree pattern can be thought of as a special case of a node-selecting query. We begin by presenting a different notation, where queries are modelled as functions from trees to sets of tuples.

A syntax for queries and tuples. Suppose that X is a set of *variables*. An *X-tuple* from a set A is an element of A^X. We are mainly interested in X-tuples of nodes, or in X-tuples of data values. We will use the letter u to denote X-tuples. If $Y \subseteq X$, and u is an X-tuple, then we write $u|Y$ for the Y-tuple obtained from u by only keeping coordinates from Y. We lift this notation to sets in the natural manner: for a set of tuples U, we define $U|Y$ to be the set $\{u|Y : u \in U\}$.

To avoid confusion, we distinguish between two kinds of queries:

- A *node-selecting query with variables X* maps a data tree to a set of X-tuples of nodes:

$$T \in \text{datatrees} \qquad \mapsto \qquad \alpha(T) \subseteq \text{nodes}(T)^X$$

- A *data-selecting query with variables X* maps a data tree to a set of X-tuples of data values.

$$T \in \text{datatrees} \qquad \mapsto \qquad \pi(T) \subseteq \mathbb{D}^X$$

For queries, be they node-selecting or data-selecting, we use operators of relational algebra:

$$(\alpha \vee \beta)(T) \stackrel{\text{def}}{=} \alpha(T) \cup \beta(T) \qquad (\alpha \wedge \beta)(T) \stackrel{\text{def}}{=} \alpha(T) \cap \beta(T) \qquad (\alpha \times \beta)(T) \stackrel{\text{def}}{=} \alpha(T) \times \beta(T)$$

We use letters α and β to denote node-selecting queries, and the letter π to denote data-selecting queries.

Node-selecting queries. A node-selecting query is called *regular* if it can be described using a regular tree language. The idea is to annotate a tree with a tuple of nodes, and then use a regular tree language to see if that tuple belongs to the output of the query on that tree. For a tree T and an X-tuple of nodes u, we define a tree $T \otimes u$ as follows. The nodes of $T \otimes u$ are the same as the nodes of T. The label of a node x consists of: the label of x in T, and then a bit-vector in 2^X that

says which coordinates of the tuple u are equal to x. A node-selecting query with variables X is called *regular* if there is some regular tree language L such that

$$\alpha(T) = \{u : u \text{ is an } X\text{-tuple of nodes in } T \text{ such that } T \otimes u \in L\} \qquad \text{for every tree } T.$$

An equivalent formalism for defining regular node-selecting queries would be to use formulas of monadic second-order logic with free first-order variables. The formalism is equivalent in the following sense: a node-selecting query α with variables X is regular if and only if there is an MSO formula φ with free first-order variables X, such that

$$u \in \alpha(T) \quad \text{iff} \quad T, u \models \varphi \qquad \text{holds for every tree } T \text{ and } X\text{-tuple } u \text{ of nodes in } T.$$

Observe that regular node-selecting queries ignore the data values in an input data tree, and only read the labeling with the finite alphabet. That is why we will sometimes feed a node-selecting query with a tree over a finite alphabet, without the data values.

Data-selecting queries. We study two kinds of data-selecting queries, as described below.

- A boolean combination η of equalities on variables X can be seen as a data-selecting query with variables X in the following way: the query ignores its input data tree, and it outputs the (possibly infinite) set of X-tuples of data values that satisfy the equalities and inequalities from η. For example, the formula $x = y \wedge y \neq z$ can be seen as a data-selecting query which maps every data tree to the set of $\{x, y, z\}$-tuples u of data values where coordinates x and y are equal, but unequal to coodinate z.

- There is a natural transformation from node-selecting queries to data-selecting queries, defined as follows. For an X-tuple of nodes u in a data tree T, define an X-tuple of data values $\hat{u}$ by

  $$\hat{u}(x) = \text{ the data value in node } u(x) \text{ of } T.$$

  The tuple $\hat{u}$ depends on both u and T. If α is a node-selecting query with variables X, then we define $\hat{\alpha}$ to be the following data-selecting query with variables X

  $$\hat{\alpha}(T) = \{\hat{u} : u \in \alpha(T)\}.$$

  We will be interested in the case when the node-selecting query α is regular.

Regular schema mappings. We are now ready to define regular XML schema mappings, and state their decision problems. A *regular schema mapping* $\mathcal{M}$ is a tuple $\langle \mathcal{S}_s, \mathcal{S}_t, \Sigma \rangle$, where $\mathcal{S}_s$ is a source schema, $\mathcal{S}_t$ is a target schema, and Σ is a set of dependencies of the form

$$\alpha, \eta \longrightarrow \alpha', \eta' \tag{1}$$

where

- α and α' are regular node-selecting queries with variables X and X', respectively; and

- η and η' are boolean combinations of equalities over variables X and X', respectively.

Since we will only care about decidability, and not complexity, it is not important how the node-selecting queries α and α' are represented. For the sake of concreteness, suppose that they are represented by MSO formulas. A pair of trees (T, T') satisfies the dependency (1) if

$$\text{for every } u \in \hat{\alpha}(T) \cap \eta(T) \qquad \text{there is some } u' \in \hat{\alpha}'(T') \cap \eta'(T') \qquad \text{such that } u = u'|X.$$

Given a data tree T in the source schema $\mathcal{S}_s$, we say that a data tree T' in the target schema $\mathcal{S}_t$ is a *solution* to T if the pair (T, T') satisfies every dependency in Σ.

We would like to emphasize that the MSO formulas α, α' do not depend on the data values in the trees T and T'. The data values are only inspected by η and η' in a very restrictive way: by boolean combinations of equalities. If we allowed predicates for data values in MSO logic, then the problem would become undecidable.

The main result of Sections 7, 8 and 9 is that the decision problems considered in the previous sections are also decidable in the more general setting of regular schema mappings.

Theorem 7.1. *For regular schema mappings* AbCons *is decidable,* SolEx *is decidable, and* SolEx($\mathcal{M}$) *is in* P.

In Section 7.2 we describe the proof strategy for Theorem 7.1. We begin, however, with an example that illustrates the absolute consistency problem for regular source-to-target dependencies. Let α be a regular node-selecting query with variables X. We define a regular schema mapping $\mathcal{M}_\alpha$. The source and target schemas are positively trivial, i.e. they select all documents. The schema contains only one dependency:

$$\alpha_\top, \eta_\top \longrightarrow \alpha, \eta_\top$$

where the queries $\alpha_\top$ and $\eta_\top$ are also positively trivial, i.e. they are defined by

$$\alpha_\top(T) = \text{nodes}(T)^X \qquad \eta_\top(T) = \mathbb{D}^X.$$

We will show that already for this very simple regular schema mapping, solving absolute consistency is equivalent to a nontrivial problem about regular node-selecting queries, which we call the *unbounded disjointness problem*. In the definition of the unbounded disjointness problem, we say that a set U of X-tuples is *completely disjoint* if every node appears in at most one tuple in at most one coordinate. In other words,

$$\forall u, u' \in U \ \forall x, x' \in X \qquad u \neq u' \vee x \neq x' \quad \Rightarrow \quad u(x) \neq u'(x').$$

A node-selecting query is called *unboundedly disjoint* if for every $n \in \mathbb{N}$ there is some input tree T_n such that $\alpha(T_n)$ contains at least n completely disjoint tuples. The unbounded disjointness problem is the following decision problem.

Problem:	Unbounded disjointness
Input:	A regular node-selecting query α, with variables X.
Question:	Is α unboundedly disjoint?

It is not difficult to see that a regular node-selecting query α is unboundedly disjoint if and only if the schema mapping $\mathcal{M}_\alpha$ is absolutely consistent. It follows that the absolute consistency problem is at least as difficult as the unbounded disjointedness problem for regular node-selecting queries. Solving the latter problem is actually one of the steps on our way to deciding the absolute consistency problem. This step is presented in Section 9.

7.2. Proof strategy

In this section we describe our strategy for proving Theorem 7.1.

Eliminating schemas and multiple dependencies. We begin by showing that without loss of generality, we can assume that the schema mapping given in Theorem 7.1 has no source or target schemas, and it has just one dependency. Consider a schema mapping $\langle \mathcal{S}_s, \mathcal{S}_t, \Sigma \rangle$, where Σ contains the n dependencies below

$$\alpha_1, \eta_1 \longrightarrow \alpha'_1, \eta'_1 \qquad \dots \qquad \alpha_n, \eta_n \longrightarrow \alpha'_n, \eta'_n.$$

For $i \in \{1, \dots, n\}$ let the source and target variables of the i-th dependency be X_i and X'_i. Assume also that $X'_1, \dots, X'_n$ are all disjoint. Without loss of generality, we assume that for each i, the constraints η_i and η'_i are satisfiable. Our goal is to combine schemas $\mathcal{S}_s$ and $\mathcal{S}_t$, and all the dependencies, into a single dependency.

A *boolean regular query* is defined to be a query with an empty set of variables. A boolean query corresponds to a regular language, which consists of those trees where the query selects the empty tuple. The source schema can be modeled as a boolean query α_s, and the target schema can be modeled as a boolean query α_t.

One can combine queries using the cartesian product. For instance, $\alpha_s \times \prod_{i \in \{1,\dots,n\}} \alpha_i$ is a query, with variables $\bigcup_{i \in \{1,\dots,n\}} X_i$, such that:

- in trees violating the source schema, the query selects no tuples
- in trees satisfying the source schema, the query selects tuples that are combined from all tuples selected by the queries $\{\alpha_i\}_{i \in I}$.

A first idea would be to replace the whole schema mapping by a single source-to-target dependency

$$\alpha_s \times \prod_{i \in \{1,\dots,n\}} \alpha_i, \prod_{i \in \{1,\dots,n\}} \eta_i \qquad \longrightarrow \qquad \alpha_t \times \prod_{i \in \{1,\dots,n\}} \alpha'_i, \prod_{i \in \{1,\dots,n\}} \eta'_i.$$

The problem is that this dependency is vacuously true whenever the source data tree is such that at least one of the queries $\{\alpha_i\}$ selects no tuples. That is why we need a slightly more refined approach.

For $I \subseteq \{1, \dots, n\}$, consider the following source-to-target dependency, call it d_I:

$$\alpha_s \times \prod_{i \in I} \alpha_i, \prod_{i \in I} \eta_i \qquad \longrightarrow \qquad \alpha_t \times \prod_{i \in I} \alpha'_i, \prod_{i \in I} \eta'_i.$$

The following lemma shows that testing a schema mapping for absolute consistency reduces, as a decision problem, to multiple instances of testing a single source-to-target dependency for absolute consistency. Likewise for the two variants of solution existence, except after the reduction we care for nontrivial solutions.

Lemma 7.2. *Let the schema mapping, and the dependencies d_I be as defined above.*

1. *Let T be a data tree which satisfies the source schema, and let I be the set of $i \in \{1, \dots, n\}$ such that α_i selects at least one tuple in T. Then T has a solution with respect to the schema mapping if and only if T has a solution with respect to d_I.*

2. *The schema mapping is absolutely consistent if and only for every $I \subseteq \{1, \ldots, n\}$, the dependency d_I is absolutely consistent.*

Proof. The first item is straightforward, and the second item follows from the first. □

From now on, we assume that the schema mapping consists of a single source-to-target dependency, and has no schemas.

Potential. The key notion in our proof of Theorem 7.1 is called a potential. For a data-selecting query π with variables X, we define its *potential* to be

$$[\![\pi]\!] \stackrel{\text{def}}{=} \{\pi(T) : T \text{ is a data tree}\},$$

which is a family of subsets of $\mathbb{D}^X$. We define a preorder $\leq_{\forall\exists}$ on families of subsets of $\mathbb{D}^X$ as follows

$$\mathcal{P} \leq_{\forall\exists} \mathcal{Q} \qquad \text{if for every } P \in \mathcal{P} \text{ there exists some } Q \in \mathcal{Q} \text{ with } P \subseteq Q.$$

Observe that the relation $\leq_{\forall\exists}$ is a preorder. A dependency as in (1) is absolutely consistent if and only if the following in equality holds for potentials

$$[\![\hat{\alpha} \wedge \eta]\!] \leq_{\forall\exists} [\![\hat{\alpha}' \wedge \eta']\!]|X. \tag{2}$$

(The above says that for every data tree T, there exists a data tree T' such that the output $(\hat{\alpha} \wedge \eta)(T)$ is included in the output of $(\hat{\alpha}' \wedge \eta')(T')$, after projecting the latter output to coordinates from X. This is precisely the definition of absolute consistency.) For a data tree T, there exists a solution if and only if

$$\{(\hat{\alpha} \wedge \eta)(T)\} \leq_{\forall\exists} [\![\hat{\alpha}' \wedge \eta']\!]|X. \tag{3}$$

Our approach to both the absolute consistency and the solution existence problem is as follows. First, we define potential expressions, which are a formalism for describing potentials. We show that the preorder $\leq_{\forall\exists}$ is decidable for potentials presented by potential expressions. Finally, we prove that the potentials which appear on both sides of the inequalities (2) and (3) can be expressed using potential expressions.

Potential expressions. Fix a finite set X of variables and a finite set C of constant names. A *potential expression* with variables X and constants C is a boolean combination of equalities over variables and constants. An example of a potential expression over variables $\{x, y, z\}$ and constants $\{c\}$ is

$$x = c \wedge y = z.$$

To evaluate a potential expression, we need a finite set $D \subseteq \mathbb{D}$ of data values, and a C-tuple u of data values from D. We then define

$$[\![e]\!]_{D,u} \subseteq \mathbb{D}^X$$

to be the set of X-tuples of data values D that satisfy the expression e, assuming that the constants are evaluated according to u. We define the *potential* of a potential expression to be the set

$$[\![e]\!] \stackrel{\text{def}}{=} \{[\![e]\!]_{D,u} : D \text{ is a finite subset of } \mathbb{D}, \text{ and } u \text{ is a } C\text{-tuple from } D\}$$

of possible values of the expression e, ranging over all choices of D and u.

Proof strategy. We now describe our proof strategy for Theorem 7.1. Let $\equiv_{\forall\exists}$ denote the equivalence relation induced by the preorder $\leq_{\forall\exists}$ on potentials. The main part of the proof is the following proposition, which says that the potential of a regular data-selecting query can be represented using potential expressions.

Proposition 7.3. *For every, even non-regular, node-selecting query α there is a finite set $\{e_i\}_{i\in I}$ of potential expressions such that*

$$[\![\hat{\alpha}]\!] \equiv_{\forall\exists} \bigcup_{i\in I} [\![e_i]\!].$$

If the query α is regular, then the set $\{e_i\}_{i\in I}$ can be computed.

The proof of Proposition 7.3 is long, and will be presented in Sections 8 and 9. We first show how the proposition implies Theorem 7.1.

Consider the following corollary to Proposition 7.3.

Corollary 7.4. *Let α be a regular node-selecting query, and η a boolean combination of equalities over the variables of α. One can compute a finite set $\{e_i\}_{i\in I}$ of potential expressions such that*

$$[\![\hat{\alpha} \wedge \eta]\!] \equiv_{\forall\exists} \bigcup_{i\in I} [\![e_i]\!].$$

Proof. Because η talks about equalities and inequalities of data values, and potential expressions can do this. □

We begin with deciding absolute consistency. By Lemma 7.2, we only need to solve absolute consistency for a single source-to-target dependency

$$\alpha, \eta \longrightarrow \alpha', \eta',$$

without considering source and target schemas. Let the source and target variables in the dependency be X and X'. As we have observed previously, absolute consistency of the dependency is equivalent to the following inequality

$$[\![\hat{\alpha} \wedge \eta]\!] \leq_{\forall\exists} [\![\hat{\alpha}' \wedge \eta']\!]|X.$$

Apply Corollary 7.4 to both sides of the inequality, yielding sets of potential expressions such that

$$[\![\hat{\alpha} \wedge \eta]\!] \equiv_{\forall\exists} \bigcup_{i\in I} [\![e_i]\!] \qquad \text{and} \qquad [\![\hat{\alpha}' \wedge \eta']\!] \equiv_{\forall\exists} \bigcup_{i\in I'} [\![e'_i]\!].$$

By restricting the variables of potential expressions e'_i to X, we get a finite set of potential expressions representing $[\![\hat{\alpha}' \wedge \eta']\!]|X$. Theorem 7.1 then follows from the following lemma, which says that inequality can be computed for finite sets of potential expressions.

Lemma 7.5. *For finite sets $\{e_i\}_{i\in I}$ and $\{f_j\}_{j\in J}$ of potential expressions, one can decide if*

$$\bigcup_{i\in I} [\![e_i]\!] \leq_{\forall\exists} \bigcup_{j\in J} [\![f_j]\!]$$

Proof. It is enough to solve the problem for $\{e_i\}_{i\in I} = \{e\}$, where e is a potential expression with variables X and constants C. Extending the sets of constants if necessary, we can assume that all the f_j's have the same set of constants C'. We can also assume that all potential expressions are positive boolean combinations of equalities and inequalitites. For a C-tuple u and an X-tuple v we will say that v, u satisfy (violate) e, if v satisfies (violates) e with constants interpreted according to u.

We claim that $[\![e]\!] \leq_{\forall\exists} \bigcup_{j\in J}[\![f_j]\!]$ can be equivalently expressed as follows: for each C-tuple u there exists j and a C'-tuple u' using only values from u and nulls, such that for each X-tuple v if v, u satisfies e, then v, u' satisfies f_j. It is clear that for u, u', and v only the equality/inequality pattern matters, so their entries can be taken from a set of bounded size, and the property can be checked algorithmically.

The reformulated condition is clearly sufficient. Let us see it is also necessary. By contraposition, assume that there is no appropriate j and u' for a C-tuple u. Let $D \subseteq \mathbb{D}$ be a finite set containing all the entries of u, plus $|X| + |C'|$ fresh values. We will show that $[\![e]\!]_{D,u} \not\subseteq [\![f_j]\!]_{D',w}$ for all j, w, D'. Let u' be obtained by replacing in w all values not in u with distinct nulls, and let v be such that v, u satisfy e but v, u' violate f_j. Let $E \subseteq D$ be the set of values used in w but not in u. Since D was chosen large enough, we can find an X-tuple v' from $D - E$ such that the entries of v, u and v', u have identical equality/inequality pattern. Clearly, $v' \in [\![e]\!]_{D,u}$. In order to prove $v' \notin [\![f_j]\!]_{D',w}$, note that the entries of v, u' and v', w have the same equality/inequality pattern (the values from E play the role of nulls), so v', w violate f_j. □

We now consider the solution existence problem. Let T be a data tree. Define I as in Lemma 7.2. By the Lemma 7.2, T has a solution with respect to the whole schema mapping if and only if T has a solution with respect to the single dependency d_I. By (3), this question boils down to testing inequality between two potentials. For the potential on the right hand side, we use Corollary 7.4, just as we did for the absolute consistency problem. In order to finish the proof of Theorem 7.1, it remains to show the following lemma, which is shown the same way as Lemma 7.5

Lemma 7.6. *For a finite set of tuples $U \subseteq \mathbb{D}^X$, and a finite set of potential expressions $\{e_i\}_{i\in I}$, one can decide if*

$$\{U\} \leq_{\forall\exists} \bigcup_{i\in I} [\![e_i]\!]|X.$$

If the expressions $\{e_i\}_{i\in I}$ are fixed, and only U is the input, the algorithm runs in polynomial time.

8. Potentials for regular data-selecting queries

The goal of this section is to prove Proposition 7.3, which says that for every node-selecting query α there is a finite set $\{e_i\}_{i\in I}$ of potential expressions such that

$$[\![\hat{\alpha}]\!] \equiv_{\forall\exists} \bigcup_{i\in I} [\![e_i]\!],$$

and that this set can be computed when α is regular. We do the proof for the case when α is regular, in which case we also need to compute the potential expressions $\{e_i\}_{i\in I}$. The case when α is not regular follows the same lines, except that all the decidability and regularity issues are ignored.

Our strategy is to show that every regular node-selecting query can be expressed in a normal form (Section 8.1), and then to write potential expressions for queries in normal form (Section 8.2).

8.1. A normal form for queries

In this section we prove a normal form for queries. The result is stated in Proposition 8.1.

Embellishments. We write $var(\alpha)$ for the variables of a query. A function $f : Y \to X$ can be used to convert an X-tuple u into a Y-tuple $u \circ f$:

$$(u \circ f)(y) = u(f(y)).$$

This conversion can be lifted to a node-selecting query with variables X, by

$$(\alpha \circ f)(T) = \{u \circ f : u \in \alpha(T)\}.$$

Let Φ be a set of queries. We allow queries in Φ to have different sets of variables, and possibly different arities (numbers of variables). An *embellishment with basis* Φ is a query of the form

$$\bigvee_{i \in I} (\beta_i \times \gamma) \circ f_i,$$

such that for every $i \in I$

- β_i is a query from Φ;
- γ is a singleton query, which means that it selects exactly one tuple in every tree[3];
- f_i is a function $f_i : var(\alpha) \to var(\beta_i) \cup var(\gamma)$.

We say a query α has basis Φ if it can be presented as some embellishment with basis Φ. The query has a regular basis Φ if the queries β_i and γ are regular. We say that a basis Φ is *unboundedly disjoint* if the product query $\prod_{\beta \in \Phi} \beta$ is unboundedly disjoint. Equivalently, this means that for every $k \in \mathbb{N}$, there is some tree where each query $\beta \in \Phi$ selects at least k completely disjoint tuples.

We adopt the convention that the product of an empty set of queries is a boolean query which says "yes" (outputs the empty tuple) in every tree. We also adopt the convention that a boolean query is unboundedly disjoint if and only if it is satisfiable. In particular, an empty basis is unboundedly disjoint.

We define an equivalence relation on finite sets of queries by saying that Φ is equivalent to Φ' if one can go from Φ to Φ' by a finite number of steps of the form: replace a query by another query with the same number of variables. We next define a partial order on the equivalence classes of this relation as follows. Let $[\Phi] \prec [\Psi]$ if one can reach Φ from a query equivalent to Ψ by a positive finite number of steps of the form: replace a query with n variables by a finite (possibly empty) set of queries with at most $n - 1$ variables. This partial order is well founded, but for every Φ containing a query with at least 2 variables, there are arbitrarily long decreasing chains that start in $[\Phi]$. We will abuse notation and write $\Phi \prec \Psi$ for $[\Phi] \prec [\Psi]$.

We are now ready to state the result on normal forms.

Proposition 8.1. *Let Φ be a set of regular queries. For every regular query α with regular basis Φ one can compute regular queries $\{\alpha_i\}_{i \in I}$ with disjoint domains such that*

$$\alpha = \bigvee_{i \in I} \alpha_i$$

and every α_i has an unboundedly disjoint regular basis.

[3]The idea is that γ selects some "important nodes" in the input tree, such as the root or the leftmost leaf.

The reason that Φ is in the statement of the proposition is that it (or rather its equivalence class) serves as an induction parameter, with respect to the order $\prec$. Observe that every regular query α has some regular basis, e.g., $\{\alpha\}$, so Proposition 8.1 implies that every regular query can be decomposed as a disjoint union of queries with unboundedly disjoint regular bases. The rest of Section 8.1 is devoted to proving Proposition 8.1. Before presenting the proof, we state three auxiliary results, Lemmas 8.2, 8.3 and 8.4.

Lemma 8.2. *If a regular query β is not unboundedly disjoint, one can compute a regular basis Φ for β such that $\Phi \prec \{\beta\}$.*

Proof. We say that a set C of nodes *covers* a set U of tuples of nodes if every tuple from U contains a node from C on at least one coordinate. It is not difficult to see that for every $k \in \mathbb{N}$, every tree T, and every set U of X-tuples of nodes, one of the two conditions below must hold:

- There are at least k completely disjoint tuples in U; or
- There is a set C of at most $k \cdot |X|$ nodes that covers U.

Since we assume that β is not unboundedly disjoint, it follows that there is some $k \in \mathbb{N}$ such that in every tree T, the output $\beta(T)$ has a cover of size at most k.

For $k \in \mathbb{N}$, consider a query β_k with variables $\{x_1, \ldots, x_k\}$, which selects a k-tuple of nodes in a tree T if and only if the nodes from the tuple cover the set of tuples $\beta(T)$. Since β is not unboundedly disjoint, there is some $k \in \mathbb{N}$ such that β_k selects at least one tuple in every tree. By testing the queries $\beta_1, \beta_2, \ldots$, we can find this k. Let γ_k be the query which selects in T only the first tuple from $\beta_k(T)$ in some MSO-definable linear order on k-tuples of nodes, e.g., the lexicographic lifting of the preorder node traversal. By definition, the query γ_k is a singleton query. In every tree T, every tuple from $\beta(T)$ contains at least one coordinate from γ_k. It follows that every tuple selected by β can be constructed by replacing at least one of its coordinates, call it x, by one of the coordinates, call it y, of the unique tuple selected by γ_k. For a coordinate $x \in var(\beta)$ and $y \in var(\gamma_k)$, define $\beta_{x,y}(T)$ to be the set of tuples u with variables $var(\beta) - \{x\}$, such that $\beta(T)$ contains the tuple obtained from u by adding a new coordinate x, storing coordinate y of the unique tuple selected by γ_k. It is not difficult to see that $\beta_{x,y}$ is a regular query. By definition, each tuple in $\beta(T)$ can be obtained by adding nodes from $\gamma_k(T)$ to a tuple from $\beta_{x,y}$ for some x, y. It follows that the set of queries

$$\Phi = \{\beta_{x,y} : x \in var(\beta), y \in var(\gamma_k)\}$$

is a regular basis for β. Because each query in Φ has fewer variables than β, it follows that $\Phi \prec \{\beta\}$. □

Lemma 8.3. *If a basis Φ is not unboundedly disjoint, one can compute a regular partition $\{L_i\}_{i \in I}$ of all trees such that for each $i \in I$, there is $\beta \in \Phi$ such that $\beta \wedge L_i$ is not unboundedly disjoint.*

Proof. If Φ is not unboundedly disjoint, then this means that there is some $k \in \mathbb{N}$ such that in every tree, some query $\beta \in \Phi$ selects at most k completely disjoint tuples. Let $L_{\beta,k}$ be the set of trees where the query β selects at most k completely disjoint tuples. It is easy to see that this is a regular tree language, by assumption on the basis being regular. For $k = 1, 2, \ldots$ we test if the regular language

$$\bigcup_{\beta \in \Phi} L_{\beta,k}$$

covers all trees, by doing a universality test for its automaton. Eventually we reach some k where this happens. Finally, we can make the languages disjoint, e.g., by using an ordering $\Phi = \{\beta_1, \ldots, \beta_n\}$ and defining

$$L_i \quad \stackrel{\text{def}}{=} \quad L_{\beta_i,k} - \bigcup_{j \in \{1,\ldots,i-1\}} L_{\beta_j,k} \qquad \text{for } i \in \{1, \ldots, n\}.$$

□

Lemma 8.4. *If α has basis Φ, and $\beta \in \Phi$ has basis Ψ, then α also has basis $\Phi - \{\beta\} \cup \Psi$.*

Proof. By substitution. □

Proof of Proposition 8.1. The induction base is when Φ is empty. By our convention, this basis is unboundedly disjoint.

Consider now the induction step. Suppose that α is a query with basis Φ. We use the decision procedure for the unbounded disjointness problem (given in Sect. 9) to decide if Φ is unboundedly disjoint. If Φ is unboundedly disjoint, then we are done. Otherwise, apply Lemma 8.3 to Φ, yielding languages $\{L_i\}_{i \in I}$. Define α_i to be $\alpha \wedge L_i$. Clearly the queries $\{\alpha_i\}_{i \in I}$ have disjoint domains, because the languages $\{L_i\}_{i \in I}$ are disjoint. It is easy to see that the set

$$\Phi_i = \{\beta \wedge L_i : \beta \in \Phi\}$$

is a basis for α_i. By Lemma 8.3, we know that there is some $\beta_i \in \Phi$ such that $\beta_i \wedge L_i$ is not unboundedly disjoint. By Lemma 8.2, the query $\beta_i \wedge L_i$ has a basis, call it Ψ_i, such that $\Psi_i \prec \{\beta_i \wedge L_i\}$. By Lemma 8.4, the query α_i has basis

$$\Phi_i - \{\beta_i \wedge L_i\} \cup \Psi_i.$$

The basis above is smaller, in the order $\prec$, than the basis Φ_i, which is in turn smaller or equivalent to Φ. We can therefore apply the induction assumption to the query α_i. □

8.2. Proof of Proposition 7.3

We now complete the proof of Proposition 7.3, which says that for every regular node-selecting query α, one can compute a finite set $\{e_i\}_{i \in I}$ of potential expressions such that

$$[\![\hat{\alpha}]\!] \equiv_{\forall\exists} \bigcup_{i \in I} [\![e_i]\!].$$

Consider a regular node-selecting query α. Apply Proposition 8.1 to α, yielding a decomposition $\alpha = \bigvee_{i \in I} \alpha_i$. Since the queries $\{\alpha_i\}_{i \in I}$ have disjoint domains, it follows that

$$[\![\hat{\alpha}]\!] = \bigcup_{i \in I} [\![\hat{\alpha}_i]\!].$$

Suppose that for each query α_i we have a potential expression e_i such that $[\![\hat{\alpha}_i]\!] = [\![e_i]\!]$. The result follows, since $\equiv_{\forall\exists}$ is a congruence with respect to union:

$$\mathcal{P} \equiv_{\forall\exists} \mathcal{P}' \ \wedge \ \mathcal{Q} \equiv_{\forall\exists} \mathcal{Q}' \qquad \text{implies} \qquad \mathcal{P} \cup \mathcal{Q} \equiv_{\forall\exists} \mathcal{P}' \cup \mathcal{Q} \qquad \text{for all families of sets } \mathcal{P}, \mathcal{P}', \mathcal{Q}, \mathcal{Q}'.$$

Therefore, it remains to provide the potential expressions e_i for each α_i. This we do now. By construction, the query α_i has an unboundedly disjoint basis, call it Φ. Let the embellishment with basis Φ be

$$\alpha_i = \bigvee_{j \in J_i} (\beta_{ij} \times \gamma_i) \circ f_{ij}.$$

We define a potential expression e_i with variables $var(\alpha_i)$ and constants $var(\gamma_i)$. For each $j \in J_i$, define an expression

$$e_{ij} \quad \stackrel{\text{def}}{=} \quad \bigwedge_{z \in var(\beta_{ij})} \bigwedge_{x,y \in f_{ij}^{-1}(z)} x = y \quad \wedge \quad \bigwedge_{c \in var(\gamma_i)} \bigwedge_{x \in f_{ij}^{-1}(c)} x = c$$

The following lemma completes the proof.

Lemma 8.5. $[\![\hat{\alpha}_i]\!] \equiv_{\forall\exists} [\![\bigvee_{j \in J_i} e_{ij}]\!]$.

Proof. We only show the more interesting inequality

$$[\![\hat{\alpha}_i]\!] \geq_{\forall\exists} [\![\bigvee_{j \in J_i} e_{ij}]\!].$$

Choose some element of the set on the right side, which itself is a set of data tuples of the form $[\![e_i]\!]_{D,c}$ for some finite set D of data values, and some valuation of constants $c : var(\gamma_i) \to D$. We will construct a data tree T such that

$$\hat{\alpha}_i(T) \supseteq [\![e_i]\!]_{D,c} \tag{4}$$

holds.

By the assumption that the basis $\{\beta_{ij}\}_{j \in J_i}$ is disjointly unbounded, we know that there is some tree (without data) S, such that for each $j \in J_i$, the set $\beta_{ij}(S)$ contains many completely disjoint tuples (the exact number of tuples needed could be determined from the argument below).

We will add data to this tree, creating a data tree T such that (4) holds. Consider the unique tuple selected by the singleton query γ_j in S. Choose the data values of the nodes in this unique tuple according to the valuation $c : var(\gamma_i) \to D$. If the number of completely disjoint tuples selected by the queries β_{ij} in S is sufficiently large, then we know that for each j there is a set

$$U_j \subseteq \beta_{ij}(S)$$

of at least $|D^{var(\beta_{ij})}|$ completely disjoint tuples. Without loss of generality, we can assume that the union of the sets U_j is also completely disjoint, i.e., no node appears in a tuple from U_j and $U_{j'}$ for $j \neq j'$. Because of the size of the sets U_j, we can choose a labeling of the tree S with data values, so that for each $j \in J_i$,

$$D^{var(\beta_{ij})} \subseteq \{\hat{u} : u \in U_j\}.$$

(Recall that $\hat{u}$ is the tuple of data values labeling the tuple of nodes u.) Let T be the data tree obtained from S by adding the data values in the manner described above. We will show (4).

To prove (4), we will show that for every $j \in J_i$,

$$[\![e_{ij}]\!]_{D,c} \subseteq \hat{\alpha}_i(T).$$

Let then d be a tuple of data values in the set $[\![e_{ij}]\!]_{D,u}$. Coordinates of this tuple are indexed by $var(\alpha_i)$. Recall the function

$$f_{ij} : var(\alpha_i) \to var(\beta_{ij}) \cup var(\gamma_i).$$

We may assume without loss of generality that the image of the function f_{ij} contains all of $var(\beta_{ij})$; otherwise some coordinates in the output of β_{ij} would not contribute anything to the output of α_i, and therefore these coordinates could be projected away. By recalling the potential expression e_{ij}, we observe two things about the tuple d.

1. Suppose that $x \in var(\gamma_i)$. Then on all coordinates from $f_{ij}^{-1}(x)$, the tuple d stores the data value $c(x)$.
2. Suppose that $x \in var(\beta_{ij})$. Then on all coordinates from $f_{ij}^{-1}(x)$, the tuple d stores the same data value, call it d_x.

Define a tuple of data values $e : var(\beta_{ij}) \to D$ by $e(x) = d_x$. Like for any tuple indexed by $var(\beta_{ij})$, this tuple is equal to $\hat{u}$ for some node tuple $u \in U_j$. Therefore, item 2. above can be restated as "on all coordinates from $f_{ij}^{-1}(x)$, the tuple d stores the data value $e(x)$". By definition of the query α_i, we know that there is some tuple of nodes in $\alpha_i(T)$, which is obtained from u by the embellishment f_{ij}. It is not difficult to check that the data values stored in that tuple are exactly the tuple d. □

9. The unbounded disjointness problem

In this section we prove decidability for the unbounded disjointness problem for regular queries, as defined in Section 7.1.

Theorem 9.1. *It is decidable if a regular query is unboundedly disjoint.*

The rest of this section is devoted to showing the above theorem.

Forest algebra. To prove Theorem 9.1, we use methods of forest algebra. In particular, we use a different definition of a context than previously: a context is a forest (and not necessarily a tree, as before) where exactly one of the leaves is a port. The idea behind forest algebra is to study the relationships between forests and contexts, with respect to the seven operations defined below:

1. a constant[4] which represents the empty forest with no trees;
2. a constant which represents the identity context with a single node which is a port.
3. a binary operation which concatenates two forests $F_1 + F_2$ yielding a forest;
4. a binary operation which concatenates a forest to a context $F + C$ yielding a context;
5. a binary operation which concatenates a context to a forest $C + F$ yielding a context;
6. a binary operation which substitutes one context in another $C_1 \cdot C_2$, yielding a context;
7. a binary operation which substitutes a forest in a context $C \cdot F$, yielding a forest.

[4] A constant is an operation of arity 0.

Let Γ be a finite alphabet. A *forest algebra congruence over* Γ consists of:

- an equivalence relation $\sim_H$ for forests over Γ;
- an equivalence relation $\sim_V$ for contexts over Γ;

such that the two equivalences are a congruence with respect to all operations of forest algebra. A forest algebra congruence recognizes a forest language over Γ, which covers the special case of a tree language, if the language is a union of equivalence classes of the congruence. In [10] it is shown that a tree language over Γ is regular if and only if it is recognized by a forest algebra congruence with finitely many equivalence classes in both $\sim_H$ and $\sim_V$.

Suppose that α is a regular query with variables X, for which we want to decide unbounded disjointness. Fix α for the rest of this section. We use the term *partial X-tuple* for an X-tuple where some of the coordinates are not defined. The *domain* of a partial X-tuple is the set of coordinates where the tuple is defined. Recall the definition of $F \otimes u$, which was introduced together with regular node-selecting queries. This definition also makes sense when u is a partial X-tuple.

Let Γ be the input alphabet for the trees considered by α. By definition of regular node-selecting queries, the tree language

$$L = \{T \otimes u : u \in \alpha(T)\},$$

is a regular language of unranked trees over the alphabet $\Gamma \times 2^X$. Suppose that the language L is recognized by a forest algebra congruence $(\sim_H, \sim_V)$. Let H be the equivalence classes in $\sim_H$, likewise for V. We denote elements of H by g, h and elements of V by v, w. We write $[F]$ for the equivalence class of a forest F, and likewise $[C]$ for the equivalence class of a context C. Because the equivalence relations $\sim_H$ and $\sim_V$ are congruences, one can apply the operations of forest algebra to equivalence classes. For instance, for $v \in V$ and $h \in H$ there is a unique element $v \cdot h$ such that $v \cdot h = [C \cdot F]$ holds for every context C and forest F with $[C] = v$ and $[F] = h$.

We can treat $h \in H$ as a query, call it $\bar{h}$, which maps a forest F over alphabet Γ to the set

$$\bar{h}(F) = \{u : u \text{ is a partial } X\text{-tuple of nodes in } F \text{ with } [F \otimes u] = h\}.$$

Likewise, we can treat $v \in V$ as a query $\bar{v}$ which maps a context to a set of partial X-tuples inside the context (the tuples are not allowed to use the port node). We say that $h \in H$ is *accepting* if it is a subset of L, as an equivalence class. It is not difficult to see that

$$\alpha = \bigvee_{h \text{ is accepting}} \bar{h}.$$

It follows that α is unboundedly disjoint if and only if for some accepting $h \in H$, the query $\bar{h}$ is unboundedly disjoint. Therefore, in order to prove Theorem 9.1, it remains to determine for which $h \in H$ the query $\bar{h}$ is unboundedly disjoint. This is the subject of the rest of this section.

An element $h \in H$ is called productive if there is some $v \in V$ such that vh is accepting. Likewise, $v \in V$ is called productive if vh is productive for some $h \in H$. The idea is that productive elements are those that can appear in some tree from L. For every productive $h \in H$ there is some set of variables $var(h) \subseteq X$ such that for every forest F, all tuples in $\bar{h}(F)$ have domain $var(h)$. This is because otherwise, α could select a tuple where either not all variables from X are used, or some variable is used twice. Likewise for every productive $v \in V$ one can define $var(v)$.

Let $v \in V$. We define erase(v) to be the set of elements $w \in V$ such that for some context C and some $var(v)$-tuple u of nodes in C, we have

$$[C \otimes u] = v \qquad \text{and} \qquad [C \otimes \emptyset] = w. \tag{5}$$

$$\frac{v \in V \quad var(v) = \emptyset}{v \in V_\infty} \text{ (base)}$$

$$\frac{v \in V_\infty \quad h \in H_\infty \quad var(v) \cap var(h) = \emptyset}{vh \in H_\infty} \text{ (subst)}$$

$$\frac{v \in V_\infty \quad w \in V_\infty \quad var(v) \cap var(w) = \emptyset}{vw \in V_\infty} \text{ (compose)}$$

$$\frac{v \in V \quad vw = v = wv \text{ for some } w \in \text{erase}(v)}{v \in V_\infty} \text{ (idem)}$$

Table 1: The proof rules.

9.1. A proof system

In this section we define a sound and complete proof system, which finds the unboundedly disjoint elements of H and V. The proof system is illustrated in Table 1. It generates two sets

$$H_\infty \subseteq H \qquad \text{and} \qquad V_\infty \subseteq V.$$

Proposition 9.2. *The system in Table 1 is sound and complete in the following sense:*

- *Sound: if $h \in H_\infty$ then $\bar{h}$ is unboundedly disjoint, and likewise for V_∞.*
- *Complete: if $\bar{h}$ is unboundedly disjoint then $h \in H_\infty$, and likewise for V_∞.*

Proposition 9.2 yields Theorem 9.1. This is because, using a fixpoint algorithm, we can compute the sets H_∞ and V_∞ derived by the proof system. The algorithm runs in polynomial time, assuming that the query is represented as a forest algebra. Actually, we believe that the algorithm might also be polynomial even when the input is represented by a nondeterministic automaton on unranked trees; we intend to study this possibility in the future.

We begin with the simpler soundness proof.

Lemma 9.3. *The proof system from Table 1 is sound.*

Proof. The rule (base) is sound because of our convention that boolean queries are unboundedly disjoint. The rule (subst) is sound because for every context C and forest F,

$$\text{disjoint}(\bar{vh}(CT)) \geq \min(\text{disjoint}(\bar{v}(C)), \text{disjoint}(\bar{h}(T))).$$

A similar argument shows that the rule (compose) is sound.

The most interesting rule is (idem). Suppose that $v \in V$ and $w \in \text{erase}(v)$ satisfy the assumption of the rule (idem). We will show that $\text{disjoint}(\bar{v}) = \infty$. Consider a context C with a tuple u such

that condition (5) in the definition of erase(v) is satisfied. For $n \in \mathbb{N}$, consider the context C^n. For every $i \in \{1, \ldots, n\}$,

$$\big[(C \otimes \emptyset)^{i-1} \cdot (C \otimes u) \cdot (C \otimes \emptyset)^{n-i}\big] \qquad = \qquad w^{i-1} \cdot v \cdot w^{n-i},$$

which is equal to v by the assumptions of the rule (idem). Therefore, there are at least n disjoint tuples in $\bar{v}(C^n)$. It follows that $\bar{v}$ is unboundedly disjoint. □

The rest of Section 9 is devoted to showing that the proof system from Table 1 is complete. The completeness proof is in Section 9.3. We begin by stating some lemmas in Section 9.2.

9.2. Two applications of the Ramsey theorem

In this section, we present two results that are used in the completeness proof, namely Lemmas 9.5 and 9.7. Both of these results are similar in that they find some regular behavior in a set of completely disjoint tuples; in both cases the regular behavior is established using the Ramsey theorem, as stated below.

Theorem 9.4 (Ramsey Theorem)**.** *Let A be a finite set of colors. For every $k \in \mathbb{N}$ there is some $n \in \mathbb{N}$ with the following property. For every coloring*

$$f : \{(i, j) : 1 \leq i < j \leq n\} \to A$$

there is a subset $I \subseteq \{1, \ldots, n\}$ of cardinality at least k and a color $a \in A$ such that $f(i, j) = a$ for all $i < j$ from I.

If a forest F is decomposed into several parts, such as $C \cdot F'$, then we can talk of nodes (or tuples of nodes) as being in the C part or the F' part. We say tuples $u_1, \ldots, u_n$ in a tree F are segregated if there is a decomposition $F = C_1 \cdots C_n \cdot F'$ such that for every $i \in \{1, \ldots, n\}$, the tuple u_i is in the C_i part. Observe that because contexts can have their port in the root, it is possible that there are n siblings such that each one of them is in a different C_i part.

Lemma 9.5. *There is some $m \in \mathbb{N}$ such that for every productive $h \in H$, if $\bar{h}$ can select at least m segregated tuples in some forest, then $h \in H_\infty$.*

Proof. Let $h \in H$. Suppose that $\bar{h}$ selects at least m segregated tuples in some forest F. By definition of segregated tuples, this means that F can be decomposed as

$$F = C_1 \cdot C_2 \cdots C_m \cdot F'$$

such that for every $i \in \{1, \ldots, m\}$, there is at least one tuple $u_i \in \bar{h}(T)$ that is in the C_i part. For $i \in \{1, \ldots, m\}$, define $v_i \in V$ to be the equivalence class of the context C_i with the tuple u_i distinguished, and $w_i \in V$ to be the equivalence class of the context C_i with no tuple distinguished. By definition, we have $w_i \in \text{erase}(v_i)$. Also, we write $w[i..j)$ for $w_i \cdots w_{j-1}$, and similarly for $w(i..j]$ and $w(i..j)$, with square brackets standing for closed intervals and round brackets standing for open intervals. Finally, define g to be the equivalence class of the forest F' with no tuple distinguished. In terms of forest algebra, the assumption on F having m segregated tuples is stated as follows:

$$w[1..i) \cdot v_i \cdot w(i..m] \cdot g = h \qquad \text{for all } i \in \{1, \ldots, m\}.$$

Recall that each $w[i..j)$ is an element of the finite set V. If m is sufficiently large, then we can apply the Ramsey theorem to the coloring $\{i < j\} \mapsto w[i..j)$, and find a subset $I \subseteq \{1, \ldots, m\}$ with at least four elements such that $w[i..j)$ is always the same, call it $w \in V$, for every choice of $i < j \in I$. Choose four elements $i_1 < i_2 < i_3 < i_4 \in I$. Observe that w must be an idempotent, because

$$w = w[i_1..i_3) = w[i_1..i_2) \cdot w[i_2..i_3) = w \cdot w.$$

By definition, every w_i satisfies $var(w_i) = \emptyset$. It follows that $var(w) = \emptyset$. Define

$$v' = w[i_1..i_2) \cdot v_{i_2} \cdot w(i_2..i_3) \cdot w(i_3..i_4] = w \cdot v_{i_2} \cdot w(i_2..i_3) \cdot w$$

Observe that w is both a prefix and a suffix of v'. By idempotency of w, we see that

$$wv' = v' = v'w.$$

By construction $w = www \in \text{erase}(v')$. Therefore, rule (idem) says that $v' \in V_\infty$. Because

$$h = w[1..i_1) \cdot v' \cdot w(i_4..m] \cdot g$$

we can use the rule (compose) twice and then the rule (subst) to conclude that $h \in H_\infty$. □

We now present the second application of the Ramsey theorem, Lemma 9.7, which says that if there are many disjoint tuples in a forest, then many of them must be distributed regularly. A *descendant antichain* is defined to be a set of nodes that are unrelated to each other by the descendant relation. Before proving the lemma, we state a simple fact, proved for instance in [9].

Fact 9.6. *Let $m \in \mathbb{N}$. There is $k \in \mathbb{N}$ such that every descendant antichain of at least k nodes contains a segregated subset of at least m nodes.*

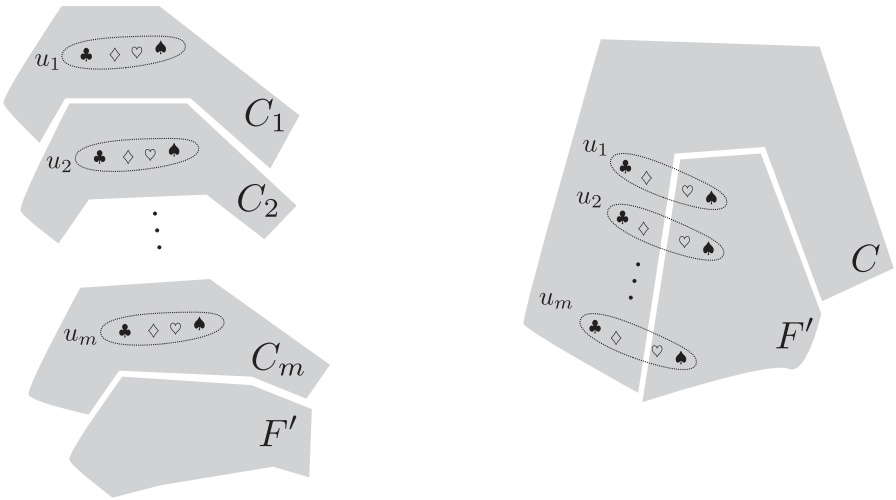


Figure 4: The two cases in Lemma 9.7. The variables are $Y = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$. In the picture on the right, which illustrates case 2 of the lemma, $Z = \{\clubsuit, \diamondsuit\}$.

Lemma 9.7. *For every $m \in \mathbb{N}$ there is some $n \in \mathbb{N}$ such that if F is a forest, and W is a set of n completely disjoint Y-tuples in F, then there are tuples $u_1, \ldots, u_m \in W$ such that either:*

1. *The tuples are segregated; or*
2. *There is a decomposition $F = C \cdot F'$ and a subset $\emptyset \subsetneq Z \subsetneq Y$ such that :*
   - *For every coordinate $z \in Z$, the nodes $u_1(z), \ldots, u_m(z)$ are in the C part.*
   - *For every coordinate $z \in Y - Z$, the nodes $u_1(z), \ldots, u_m(z)$ are in the F' part.*

The two cases are illustrated in Figure 4.

Proof. We prove the lemma for the case of binary trees, where each node has at most two children. The unranked case is then deduced by encoding an unranked tree in a binary tree.

For a tuple of nodes u, define $\wedge u$ to be the unique node that is ancestor of all nodes in u, and which is deepest in the tree for this property. For an X-tuple of nodes u, define its *spatial matrix* to be the function

$$\mathit{space}(u) : X^2 \to \{\text{ancestor}, \text{descendant}, \text{equal}, \text{none}\},$$

which says how each pair of nodes in the tuple is related in the tree.

For each tuple $u \in W$, define $\hat{u}$ be the tuple obtained from u by adding the node $\wedge u$. Apply Fact 9.6 to m, yielding some k. Without loss of generality assume also that $k > 2^{|Y|} \cdot m$. By the Ramsey theorem, if W contains enough tuples, then there are tuples $u_1, \ldots, u_k \in W$ such that the following regularity condition holds:

the matrix $\mathit{space}(\hat{u}_i, \hat{u}_j)$ does not depend on the choice of $i < j \in \{1, \ldots, k\}$.

In the above, we apply *space* to the concatenation of two tuples $\hat{u}_i$ and $\hat{u}_j$. By the regularity condition, one of the following cases must hold:

1. The nodes $\wedge u_1, \ldots, \wedge u_k$ are a descendant antichain;
2. The nodes $\wedge u_1, \ldots, \wedge u_k$ are linearly ordered by the descendant relation;
3. The nodes $\wedge u_1, \ldots, \wedge u_k$ are all equal.

We study these cases separately.

1. If the nodes $\wedge u_1, \ldots, \wedge u_k$ are a descendant antichain, then Fact 9.6 shows that at least m tuples from $u_1, \ldots, u_k$ are segregated, as required by condition 1 in the statement of the lemma.
2. Suppose that the nodes $\wedge u_1, \ldots, \wedge u_k$ are linearly ordered by the descendant relation, say in the order of increasing depth. By the regularity condition, for every variable $y \in Y$, one of the following cases must hold:
   (a) For every $i < j \in \{1, \ldots, k\}$, node $u_i(y)$ is a descendant of $\wedge u_j$. This means that all the nodes $u_1(y), \ldots, u_k(y)$ are descendants of $\wedge u_k$.
   (b) For every $i < j \in \{1, \ldots, k\}$, node $u_i(y)$ is not a descendant of $\wedge u_j$. This means that for each $i \in \{1, \ldots, k-1\}$, the node $u_i(y)$ is a descendant of $\wedge u_i$, but not of $\wedge u_{i+1}$.

   Observe that condition (a) cannot hold for all coordinates $y \in Y$, because otherwise $\wedge u_k$ would be an ancestor of every node in the tuples $u_1, \ldots, u_k$, contradicting the definition of $\wedge u_1, \ldots, \wedge u_{k-1}$. Let $Z \subseteq Y$ be the set of coordinates y for which condition (b) holds. The set Z is nonempty as we have just observed. If $Z = Y$, then we have $k - 1 \geq m$ segregated tuples, as required by condition 1 in the statement of the lemma. If $Z \subsetneq Y$, then we see that condition 2 from the statement of the lemma holds, with the decomposition $F = C \cdot F'$ partitioning the nodes of F into non-descendants of $\wedge u_k$ and descendants of $\wedge u_k$.

3. Suppose that the nodes $\wedge u_1, \ldots, \wedge u_k$ are all the same node, call it x. This is the only case where we use the assumption that the tree is binary. None of the tuples $u_1, \ldots, u_k$ can contain the node x, since by the regularity condition all tuples would contain x, contradicting the assumption on complete disjointness. By definition of $\wedge u_i$, for each $i \in \{1, \ldots, k\}$, some but not all nodes in the tuple u_i are in the left subtree of x, and the remaining nodes are in the right subtree of x. Partition the tree F into $C \cdot F'$, where F' is the subtree of the left child of x. By the pigeon-hole principle and our choice of k, there must be some $\emptyset \subsetneq Z \subsetneq Y$, such that for at least m tuples, the coordinates from Z are in the F' part, and the remaining coordinates are in the C part. This establishes condition 2 in the statement of the lemma.

□

9.3. Completeness

We now prove completeness of the system in Table 1. The proof is by induction on the number of variables. Formally, we prove the following statement by induction on the size of a set $Y \subseteq X$ of variables:

(*) For every productive $h \in H$, if $\bar{h}$ is unboundedly disjoint and $var(h) \subseteq Y$, then $h \in H_\infty$. Likewise for $v \in V$.

The induction base of $Y = \emptyset$ is immediate, thanks to the rule (base).

We now focus on the induction step. Suppose that we have proved the statement for all sets smaller than Y, and we want to prove it for Y. Let m_0 be the maximal number of disjoint tuples that can be selected by a $\bar{\sigma}$ such that $\sigma \in H \cup V$ is not unboundedly disjoint. Let m be a number that is bigger than the constant from Lemma 9.5, and which is also at least $|H| \cdot |V|$ times bigger than m_0. Apply Lemma 9.7 to m, yielding some n. We will prove that if $\bar{h}$ selects at least n completely disjoint tuples in some forest F, then the proof system yields $h \in H_\infty$.

Suppose that F is a forest where $\bar{h}(F)$ contains at least n disjoint tuples. Apply Lemma 9.7, yielding a set of tuples $u_1, \ldots, u_m \in \bar{h}(T)$. Consider the two cases from the lemma:

1. If the tuples are segregated, then $h \in H_\infty$ follows from Lemma 9.5.
2. Let $F = C \cdot F'$ and Z be as in Lemma 9.7. By the lemma, for each $i \in \{1, \ldots, m\}$,

$$F \otimes u_i = (C \otimes u_{i1}) \cdot (F' \otimes u_{i2}),$$

where u_{i1} is a Z-tuple and u_{i2} is a $(Y - Z)$-tuple. Define

$$v_i = [C \otimes u_{i1}] \qquad h_i = [F' \otimes u_{i2}].$$

By construction, these elements satisfy $h = v_i h_i$. By the pigeon-hole principle, there must be some $(h_*, v_*) \in H \times V$ such that $(h_*, v_*) = (h_i, v_i)$ holds for at least m_0 values of i. By choice of m_0, both $\bar{h}_*$ and $\bar{v}_*$ are unboundedly disjoint. Because the variables of these queries are Z and $Y - Z$, both proper subsets of Y, we can use the induction assumption to show that the proof system derives $v_* \in V_\infty$ and $h_* \in H_\infty$. Using the rule (subst), we derive

$$h = v_* h_* \in H_\infty$$

This completes the proof of Proposition 9.2, which completes the proof of Theorem 9.1.

Mappings	AbCons	SolEx	SolEx($\mathcal{M}$)
bounded-depth	Π_4P	Σ_3P	in LOGSPACE
pattern-based	Π_2EXP	NEXP	in LOGSPACE
regular	decidable	decidable	in P

Mappings	Solution building
fixed	in P
bounded-depth	in EXP
pattern-based	in EXPSPACE

Figure 5: Summary of complexity results

10. Future work

A summary of complexity results is shown in Fig. 5. Notably lacking are lower bounds for regular mappings. As we have mentioned, more specialized techniques are likely to bring algorithms with better complexity.

An open direction of more theoretical interest is to explore the limits of decidability. Which kinds of logic in dependencies guarantee decidability of absolute consistency? Or even more generally, which kinds of dependencies? We believe that our notion of potential can be helpful.

A fundamental concept in data exchange is a universal solution, i.e., a solution that can be mapped homomorphically into every other solution. Universal variants of the problems we have considered are worth exploring. On the other hand, homomorphisms of ordered trees are injective, which makes a universal solution a rare bird. Developing relaxed versions of universal solutions seems an interesting research topic.

Acknowledgments. The authors thank Leonid Libkin for inspiring discussions, and Alin Deutsch and the anonymous referees of ICDT 2011 and JCSS for helpful comments.

The first author was supported by the Future and Emerging Technologies (FET) programme within the Seventh Framework Programme for Research of the European Commission, under the FET-Open grant agreement FOX, number FP7-ICT-233599.

The second and the third author were supported by the *Querying and Managing Navigational Databases* project realized within the Homing Plus programme of the Foundation for Polish Science, cofinanced by the European Union from the Regional Development Fund within the Operational Programme Innovative Economy ("Grants for Innovation").

The second author was also supported by Polish Ministry of Science and Higher Education grant no. N N201 382234.

References

[1] S. Amano, L. Libkin, and F. Murlak. XML schema mappings. In *PODS 2009*, pages 33–42.
[2] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava. Tree pattern query minimization. *VLDB J.* 11 (2002), 315–331.
[3] M. Arenas and L. Libkin. XML data exchange: consistency and query answering. *JACM*, 55(2):7:1–72, 2008.
[4] M. Benedikt, W. Fan, F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM* 55(2): (2008).
[5] P. Barceló. Logical foundations of relational data exchange. *SIGMOD Record*, 38(1):49–58, 2009.
[6] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD*, 2007.
[7] H. Björklund, W. Martens, T. Schwentick. Conjunctive query containment over trees. *DBPL'07*, pages 66–80.
[8] H. Björklund, W. Martens, T. Schwentick. Optimizing conjunctive queries over trees using schema information. *MFCS'08*, pages 132–143.
[9] M. Bojańczyk, L. Segoufin, H. Straubing. Piecewise testable tree languages *LICS'08*, pages 442–451.
[10] M. Bojańczyk, I. Walukiewicz Forest algebra *Logic and Automata*, pages 107–132.

[11] C. David. Complexity of data tree patterns over XML documents. In *MFCS'08*, pages 278–289.
[12] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Date exchange: semantics and query answering. *TCS*, 336:89–124, 2005.
[13] R. Fagin, P. G. Kolaitis, L. Popa, and W.-C. Tan. Composing schema mappings: second-order dependencies to the rescue. *ACM TODS*, 30(4):994–1055, 2005.
[14] G. Gottlob, C. Koch, K. Schulz. Conjunctive queries over trees. *J.ACM* 53(2):238–272 (2006).
[15] J. Hidders. Satisfiability of XPath expressions. In *DBPL'03*, pages 21–36.
[16] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *PODS 2005*, pages 61–75.
[17] R. Miller, M. Hernandez, L. Haas, L. Yan, C. Ho, R. Fagin, and L. Popa. The Clio project: managing heterogeneity. *SIGMOD Record*, 30:78–83, 2001.
[18] M. Murata, D. Lee, M. Mani, and K. Kawaguchi. Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology, 5(4):1-45, 2005.
[19] F. Neven. Automata Theory for XML Researchers. *SIGMOD Record* 31(3): 39-46 (2002).