The (Almost) Complete Guide to Tree Pattern Containment*

Wojciech Czerwiński
University of Warsaw

Wim Martens
Universität Bayreuth

Paweł Parys
University of Warsaw

Marcin Przybyłko
University of Warsaw

ABSTRACT

Tree pattern queries have been investigated in database theory since more than a decade ago. They are a fundamental and flexible query mechanism and have been considered in the context of querying tree structured as well as graph structured data. We revisit their containment, validity, and satisfiability problem, both with and without schema information. We present a comprehensive overview of what is known about the complexity of containment and develop new techniques which allow us to obtain tractability- and hardness results for cases that have been open since the early work on tree pattern containment. For the tree pattern queries we consider in this paper, it is known that the containment problem does not depend on whether patterns are evaluated on trees or on graphs. This means that our results also shed new light on tree pattern queries on graphs.

1. INTRODUCTION

Tree pattern queries are a fundamental building block for many query- and specification languages for tree- and graph-structured data. They have been studied under many names: tree patterns, twig patterns, twig queries, and XPath queries with child and descendant axis. In the context of tree-structured data, they form the core of XPath [6]. XPath is the main mechanism for node selection in XQuery [10] and XSLT [28], the most widely used query languages for XML. In addition, XPath is used for specifying integrity constraints in XML Schema [22], the currently de facto schema language for XML, and it is used in XLink [19] and XPointer [18] for referencing elements in external documents. In the context of graphs, languages based on tree patterns have become popular as well. Nested regular expressions [38] and Graph XPath [32], for example, are heavily inspired on tree patterns and extend them with extra navigational features, negation, or data value comparisons. Tree patterns similar to the ones we consider in this paper are also used to speed up pattern matching algorithms in graphs [14].

The containment problem for tree patterns and XPath queries has long been recognized as an important problem in databases and has been heavily investigated [5, 23, 27, 34, 36, 41, 43]. In this paper we study the tree patterns that were introduced by Miklau and Suciu [34]. These patterns have *wildcards* and allow navigation with *child* and *descendant* axes. Even though many variants and extensions of these patterns have been considered in the literature (see, e.g., [4, 8, 17, 20, 25, 31]), a complete picture of the complexity of containment for even these basic kind of tree pattern queries is still lacking.

We revisit several variants of the containment problem for tree patterns. Since there is a renewed interest in query languages for graphs, we note that the containment problem for the tree patterns we consider in this paper does not depend on whether they are evaluated on graphs or on trees [34] if no schema information is present. We look into this correspondence more deeply later in the paper. We also consider containment with schema information (in the form of Document Type Definitions or DTDs) and special cases of containment that are interesting in their own right, such as validity and satisfiability with respect to DTDs.

Since there are many problems that have a similar nature as containment, such as, for example, XPath minimization [29], consistent query answering [2], key inference [1], constraint implication [37], computing certain answers in incomplete databases [4, 24], we believe that the techniques we develop may be applicable in such closely related scenarios too.

Contribution. We denote the tree patterns as introduced by Miklau and Suciu [34] by TPQ or TPQ$(/, //, *)$. Here, TPQ abbreviates *tree pattern queries.* We consider several fragments, depending on features we allow or disallow. The four features we consider are *child edges* $(/)$, *(proper) descendant edges* $(//)$, *wildcards* $(*)$ and *branching.* We use the term *path queries* or PQ to refer to queries that do not have branching. Whenever we talk about a *fragment* of TPQs in this paper, we mean a class that is obtained from TPQs by disallowing zero or more of the four aforementioned features.

We consider containment problems of TPQs with and without schema information in the form of Document Type Definitions (DTDs) [13]. In the cases where schema information is present, we consider the variant where the schema is part of the input as well as the variant where the schema is fixed. Throughout the paper we

*This work was supported by DFG grant MA4938/2-1 and by Poland's National Science Centre grant no. UMO-2013/11/D/ST6/03075.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

consider two forms of containment, namely *strong containment*, in which the root of the pattern is required to match the root of the tree, and *weak containment*, which does not have this requirement.

Our main contributions are the following.

(1) For containment without schema information, we complete the picture that was started by Miklau and Suciu [34]. Miklau and Suciu showed, for every fragment $\mathcal{F}$ of TPQs in which child edges are present, whether containment for $\mathcal{F}$-queries is in P or coNP-complete. In addition, they also presented a number of results for cases in which the queries can be from different fragments, that is, containment of queries from fragment $\mathcal{F}_1$ in queries from fragment $\mathcal{F}_2$, but the picture was incomplete. One particular, non-trivial case that remained open since this early work is the complexity of path queries (that is, $\mathsf{PQ}(/, //, *)$) in tree pattern queries (that is, $\mathsf{TPQ}(/, //, *)$). We prove that this problem is solvable in P. In fact, we present, for every pair $\mathcal{F}_1$ and $\mathcal{F}_2$ of fragments of TPQs, whether the complexity of containment of queries from $\mathcal{F}_1$ in queries of $\mathcal{F}_2$ is in P or coNP-complete, for weak and for strong containment (Section 3). We obtain new, non-trivial polynomial time results as well as new coNP-completeness results. The aforementioned containment of path queries in tree pattern queries it the technically most involved one in the section.

(2) We then turn to problems that involve DTD information and present a complete overview of the complexity of satisfiability and validity of fragments of TPQs with schema information (Sections 4 and 5). More precisely we consider the weak and strong satisfiability or validity problem for every fragment of TPQs, with respect to a DTD or a fixed DTDs. For satisfiability, we classify every such variant of the problem either in P or we show that it is NP-complete. For validity, we show that the problem is always either in P or EXPTIME-complete. The EXPTIME-hardness goes back to weak validity of $\mathsf{TPQ}(/, *)$ [9]; all other cases are in P.

(3) We present an almost complete overview of the most general problem: containment of fragments of TPQ with respect to DTD information (Section 6). We solve a problem that has been open since a decade: containment of $\mathsf{PQ}(/)$ in $\mathsf{PQ}(/, *)$ with respect to DTDs, which we prove to be EXPTIME-complete, even when the DTD is fixed. This was listed as an open problem in [36].[1] The solution goes through a new variant of tiling problem that we call *triomino tiling* and that may be interesting in its own right. Tiling problems are a commonly used tool for proving complexity lower bounds and ask whether a region can be tiled using a set of tile types, respecting certain *horizontal* and *vertical* constraints. Triomino tiling unifies the horizontal and vertical constraints, which allows us to check both of them at the same time using a single gadget. This property of triomino tiling is crucial for our proof.

(4) We discuss connections between containment of TPQs over graphs and over trees (Section 7). When no schema information is present, it was already observed [34] that containment of TPQs over trees *is the same problem* than containment of TPQs over graphs. That is, we have that TPQ q_1 is contained in TPQ q_2 over trees if and only if it is contained over graphs. We present how this observation can be extended when schema information is present. We discuss several manners how DTDs can be used to specify meta-information on graphs so that the correspondence between trees and graphs also holds for the satisfiability problem. Unfortunately, the correspondence does not seem to carry over already for the validity problem.

Throughout the paper, we summarize our results in tables and present cases for which we prove new insights (to the best of our knowledge) in bold. We stress that the tables are optimized for looking up results and for space in the paper. They often summarize many cases in one cell, which is not optimal for clearly delineating which cases were already known and which are newly solved. Whenever we are aware of a known result that is not implied by a result we prove, we mention it in the text.

We found it striking that, when DTDs are involved, we do not see any complexity difference between the settings where the DTD is fixed or not. All our upper bounds already hold when the DTD is part of the input and all lower bounds hold for fixed DTDs. This shows that, no matter which formalism one uses for describing schema- or meta-information, the hardness results always hold as soon as the formalism is as expressive as a DTD. In fact, the hardness results also hold for DTDs defining *unordered* trees, as in [11].

Related Work. Containment and satisfiability of TPQs was first investigated by Miklau and Suciu [34] who showed, for example, that the containment problem is always in coNP and that its most general version is coNP-complete. However, when one removes wildcards, descendant axes, or branching from TPQs, the problem is in P. Satisfiability and containment of TPQs with respect to DTDs was first studied by Neven and Schwentick [36] and Wood [43]. The results of these papers are surveyed in [39].

Benedikt et al. [5] considered satisfiability of many fragments of XPath, among which also TPQs, with DTD information in many variations, two of which we also consider here ("DTD is fixed" and "DTD is part of the input"). Since some of the query fragments they consider have negation, some of their results imply upper bounds on containment too. Geerts and Fan consider satisfiability of queries with sibling axes [23], which is similar to considering TPQs on words instead of trees.

There is a strong connection between minimization of TPQs [21] and containment in the sense that a large class of TPQs can be solved by a procedure that builds on containment tests. However, it is not clear whether all TPQs can be minimized in this way [29].

Conjunctive queries on trees [25] are closely related to TPQs but can be graph-shaped instead of tree-shaped. Their containment problems are harder than for TPQs. Without schema information, the complexity jumps from coNP to Π_2^P [8] and, with schema information, from EXPTIME to 2EXPTIME [7].

Tree patterns representing XML with incomplete information [4] are also closely related to TPQs and, in fact, are more expressive. For example, they allow navigation in horizontal and vertical direction and have variables to bind data values. Computing *certain an-*

[1] Note to the reviewers: this problem was claimed to be in P in the conference version of that paper, but this was revised in the journal version.

swers over such patterns can be viewed as a form of containment [24]. For these more expressive patterns, Barceló et al. [4] embark on a quest of understanding the tractability frontier of query answering, which is a quest similar to ours. Containment of such patterns was studied in [17].

Finally, we note that the containment problem of tree-pattern-like queries is also relevant on graphs (see, e.g., [31, 44] and the references we already mentioned).

2. PRELIMINARIES

Here, we introduce the necessary definitions concerning graphs, trees, tree pattern queries, and schemas. For a finite set S, we denote by $|S|$ its number of elements.

2.1 Trees and Tree Pattern Queries

We consider trees that are node-labelled, rooted, unranked, and directed from the root downwards. When we do not say that the trees are infinite, resp., unordered, we assume that they have a finite number of nodes, resp., the children of each node are ordered from left to right. Our complexity results do not depend on whether trees are ordered or not. For an arbitrary, possibly infinite set of *labels* Λ, we denote Λ-*trees* as tuples $t = (\mathsf{Nodes}(t), \mathsf{Edges}(t), \mathsf{lab}^t)$, where $\mathsf{Nodes}(t)$ is the set of nodes $\mathsf{Edges}(t) \subseteq (\mathsf{Nodes}(t))^2$ the set of child edges and $\mathsf{lab}^t : \mathsf{Nodes}(t) \to \Lambda$ the labelling function. When t is clear from the context, we sometimes just denote lab^t by lab. The root of t will be denoted $\mathsf{root}(t)$. We define the *size of* t, denoted by $|t|$, to be the number of nodes of t. We denote a tree with root labelled a and subtrees $t_1, \ldots, t_n$ as $a(t_1, \ldots, t_n)$. By $\mathcal{T}_\Lambda$ we denote the set of all ordered, finite Λ-trees. We often simply say *trees* when Λ is clear from the context.

A *path* in tree t is a sequence of nodes $v_0 \cdots v_n$ such that, for each $i = 1, \ldots, n$, we have that $(v_{i-1}, v_i) \in \mathsf{Edges}(t)$. Paths therefore never run upwards, that is, turn towards to the root of t. We say that $v_0 \cdots v_n$ is a path *from* v_0 *to* v_n and that the *length* of the path is n. The *depth* of a node $v \in \mathsf{Nodes}(t)$ is equal to the length of the (unique) path from $\mathsf{root}(t)$ to v. The *depth* of a tree t is then defined as the maximum of the depths of all its nodes.

For a tree t and a node $v \in \mathsf{Nodes}(t)$, the *subtree of* t *at* v, denoted by $\mathsf{subtree}^t(v)$, is the tree induced by all the nodes u such that there is a (possibly empty) path from v to u. In particular, for any tree t and leaf node v, $\mathsf{subtree}^t(v) = \mathsf{lab}^t(v)$ and, for any other node u, $\mathsf{subtree}^t(u) = \mathsf{lab}^t(u)(\mathsf{subtree}^t(u_1), \ldots, \mathsf{subtree}^t(u_n))$, where $u_1, \ldots, u_n$ are the children of u from left to right.

We will use tree pattern queries with *wildcard*. We will denote the wildcard symbol by $*$. Following the standard conventions, tree pattern queries match trees that only bear labels that are different from $*$ and which we call *letters*.

Definition 2.1 (Tree Pattern Queries). Let Λ be a (possibly infinite) set of labels that contains the wildcard $*$. A *tree pattern query (with wildcards)*, or *TPQ*, over Λ is a tuple $q = (\mathsf{Nodes}(q), \mathsf{Edges}(q), \mathsf{Desc}(q), \mathsf{lab}^q)$ where $\mathsf{Nodes}(q)$ is a finite set of nodes, $\mathsf{Edges}(q) \subseteq (\mathsf{Nodes}(q))^2$ a finite set of edges, and $\mathsf{lab}^q : \mathsf{Nodes}(q) \to \Lambda$ such that $(\mathsf{Nodes}(q), \mathsf{Edges}(q), \mathsf{lab}^q)$ is a tree over Λ.

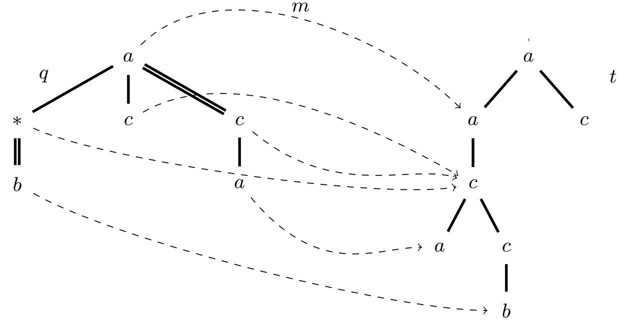


Figure 1: Mapping m is a weak embedding of the TPQ q in the tree t, but it is not a strong embedding, as $m(root(q)) \neq root(t)$. Notice that there exists also a strong embedding of q in t.

Furthermore, $\mathsf{Desc}(q) \subseteq \mathsf{Edges}(q)$ is the set of *descendant edges.*

For a node $v \in \mathsf{Nodes}(q)$, let $\mathsf{subquery}^q(v)$ be the subquery of q rooted at v, that is, the nodes of $\mathsf{subquery}^q(v)$ are precisely v and its descendants and its labels, edges, and descendant edges are inherited from q. We refrain from a formal definition.

Let t be a tree over some arbitrary set of labels Δ that does not contain the wildcard $*$. A *(weak) embedding* of a TPQ q into t is a total mapping m from $\mathsf{Nodes}(q)$ to $\mathsf{Nodes}(t)$ such that

- for every $v \in \mathsf{Nodes}(q)$ such that $\mathsf{lab}(v) \neq *$, we have $\mathsf{lab}(m(v)) = \mathsf{lab}(v)$;
- for every $v_1, v_2 \in \mathsf{Nodes}(q)$
  - if $(v_1, v_2) \notin \mathsf{Desc}$, then $(m(v_1), m(v_2)) \in \mathsf{Edges}(t)$; and
  - if $(v_1, v_2) \in \mathsf{Desc}$, then $m(v_1)$ is a proper ancestor of $m(v_2)$.

An embedding is a *strong embedding* if, additionally, it maps the root of q to the root of t.

The *(weak) language* of q is denoted $L_w(q)$ and consists of all trees t for which there is a weak embedding of q into t. The *strong language* of q is denoted $L_s(q)$ and is defined similarly but requires a strong embedding. Notice that $L_s(q) \subseteq L_w(q)$. In Figure 1, we give an example of an embedding.

The set of all tree pattern queries is denoted by TPQ or TPQ$(/, //, *)$. In this notation, we refer to $/$ as *child edges*, to $//$ as *descendant edges*, and $*$ as *wildcards.* We consider fragments of TPQs that limit the features they can use. If we omit $/$, then we only consider TPQs q where $\mathsf{Desc}(q) = \mathsf{Edges}(q)$, if we omit $//$, we assume that $\mathsf{Desc}(q) = \emptyset$, and if we omit $*$, we assume that $\mathsf{lab} : \mathsf{Nodes}(q) \to \Delta$. By PQ we denote the set of *path queries*, which are TPQs q that do not branch, that is $\mathsf{Edges}(q)$ does not contain any (v, v_1) and (v, v_2) for which $v_1 \neq v_2$. We denote fragments of PQs similar to TPQs. For example, PQ$(/, *)$ is the set of path queries that use child edges and wildcards.

2.2 Schemas

Throughout the paper, $\Sigma \subseteq \Delta$ always denotes a finite alphabet of letters. We use standard regular expressions using the operators $\cdot$ (concatenation), $+$ (disjunction),

and * (Kleene star). For a regular expression r, $L(r)$ is the language of the expression, and $\mathsf{Labels}(r)$ is the set of labels occurring in r. The *size* of a regular expression r, denoted by $|r|$, is defined as the length of its word representation.

In this paper, schemas will be mild variations on Document Type Definitions (DTDs), which we abstract as extended context-free grammars.

DEFINITION 2.2. A *DTD* is a tuple (Σ, d, S_d), where Σ is a finite alphabet, d is a function that maps Σ-symbols to regular expressions over Σ, and $S_d \subseteq \Sigma$ is the set of start symbols. For convenience we sometimes denote (Σ, d, S_d) by d and say that $a \to r$ *is a rule in* d when $d(a) = r$.

A tree t *satisfies* d if its root is labelled by an element of S_d and, for every node v with label a and children $v_1, \ldots, v_n$ from left to right, the word $\mathsf{lab}(v_1) \cdots \mathsf{lab}(v_n)$ is in the language defined by $d(a)$. By $L(d)$ we denote the language of trees satisfying d. The *size* of a DTD is $|\Sigma|+|S_d|+|d|$ where $|d|$ refers to the size of the representations of the regular string languages. Unless specified otherwise, we represent all such regular string languages by regular expressions. Hence, $|d|$ is the sum of the sizes of all expressions representing languages $d(a)$ for $a \in \Sigma$.

We assume that all DTDs d in this paper are *reduced*, which means that, for every $a \in \Sigma$ there exists a tree in $L(d)$ that contains a. It is well known that, for a DTD that is not reduced, one can find an equivalent reduced one in polynomial time.

2.3 Main Decision Problems

We will investigate the complexities of containment, satisfiability and validity of tree pattern queries, possibly with schema information.

In the following, let p and q always denote TPQs and let d denote a DTD. We say that p is weakly contained in q, if $L_w(p) \subseteq L_w(q)$. Similarly, p is strongly contained in q if $L_s(p) \subseteq L_s(q)$. We consider the following problems for TPQs:

W-CONTAINMENT	S-CONTAINMENT
Input: TPQs p, q	Input: TPQs p, q
Q: Is $L_w(p) \subseteq L_w(q)$?	Q: Is $L_s(p) \subseteq L_s(q)$?

We also consider weak and strong containment *with DTD*, in which case we also consider validity and satisfiability as special cases of containment.[2] Weak containment with respect to a DTD is formally defined as follows.

W-CONTAINMENT WITH DTD	
Input:	TPQs p, q, DTD d
Q:	Is $L_w(p) \cap L(d) \subseteq L_w(q)$?

S-CONTAINMENT is defined analogously to W-CONTAINMENT, but it uses L_s instead of L_w everywhere in the definition.

Satisfiability problems can be seen as a special case of containment. For example, we have that $L_w(p) \cap L(d) \not\subseteq \emptyset$ if and only if p is (weakly) satisfiable with respect to d. Formally, W-SATISFIABILITY with respect to a DTD asks, given a TPQ p and DTD d, whether $L_w(p) \cap L(d) \neq \emptyset$. As before, we define S-SATISFIABILITY analogously, with $L_s(p)$ instead of $L_w(p)$. The same holds true for the validity problem with DTDs, which asks whether $L(d) \subseteq L(q)$. So, formally, W-VALIDITY with respect to a DTD asks, given a TPQ q and a DTD d, whether $L(d) \subseteq L_w(q)$. Again, the corresponding S-VALIDITY problem is obtained by taking $L_s(q)$ instead of $L_w(q)$.

In the paper, we also consider variants of the problems in which the DTD is fixed. When we consider a problem with respect to a *fixed DTD* we actually refer to a set of problems. When we prove a hardness bound for such problems, it means that there exists a fixed DTD for which the problem is hard. For example, when we say that W-CONTAINMENT of $\mathsf{TPQ}(//)$ in $\mathsf{PQ}(/)$ w.r.t. fixed DTD is coNP-hard, then we mean that there exists a fixed DTD d_f for which the problem that takes as input a $p \in \mathsf{TPQ}(//)$ and $q \in \mathsf{PQ}(/)$ and asks whether $L_w(p) \cap L(d_f) \subseteq L_w(q)$ is coNP-hard. In fact, in this paper we only show *lower bounds* for problems with fixed DTD.

Weak and Strong Containment. The next observation shows that S-CONTAINMENT and W-CONTAINMENT are equivalent in the case where both queries are allowed to have descendant edges; otherwise it allows us to only present upper bounds for W-CONTAINMENT and lower bounds for S-CONTAINMENT.

OBSERVATION 2.3. *For all fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ of TPQs, there is a polynomial time reduction from* S-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ to* W-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ (with or without DTD). If $\mathcal{F}_1$ and $\mathcal{F}_2$ both contain $//$, then there is also a polynomial time reduction from* W-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ to* S-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ (with or without DTD).*

The first reduction boils down to changing the root label of both patters to an unused root symbol r (and appropriately updating the DTD). In the second reduction we attach a new r-labelled root with a descendant edge above both patterns. Details are in the appendix.

For the sake of succinctness, when we claim a result about a problem without saying that we mean the weak or strong variant, it means that we claim the result for both. Most often, this will simply be due to Observation 2.3, but sometimes it also just means that the reduction or algorithm only needs a very small adjustment.

General Upper and Lower Complexity Bounds. The containment problem of the most general class, $\mathsf{TPQ}(/, //, *)$ is known to be in coNP, which was shown by Miklau and Suciu [34]. If DTDs are involved, then containment of $\mathsf{TPQ}(/, //, *)$ is in EXPTIME [36].

Regarding lower bounds, satisfiability with respect to a DTD d is always at least as hard as the problem of computing a reduced DTD equivalent to d. Since this problem is P-hard (by a trivial reduction from the emptiness problem of context-free grammars), all decision problems in this paper that involve DTDs in the input are P-hard in general.

2.4 Queries with Output

We only consider boolean queries in this paper, i.e., queries that define a set of trees. It was shown by Miklau and Suciu that, in the case where all patterns

[2]Notice that, without DTD, validity and satisfiability of TPQs are trivial.

have the “/” operator, the containment problem of k-ary node-selecting queries reduces to containment of boolean queries (Proposition 1 in [34]). This argument was extended by Neven and Schwentick to containment problems with DTDs (Section 2.4 in [36]). Hence, all our complexity results in which the pattern fragments have “/” also hold for k-ary node-selecting TPQs on trees.

3. CONTAINMENT WITHOUT SCHEMA

Containment of TPQs without schema information was studied in the seminal paper of Miklau and Suciu [34] which gives a fairly precise picture of the tractability frontier. In particular, a main message of that paper is that containment of TPQ$(/, //, *)$ in TPQ$(/, //, *)$ is coNP-complete (see also Theorem 3.3), whereas the problem becomes tractable if we remove wildcards, descendant edges, or branching from both tree pattern queries. When considering containment of queries p in q and we allow p and q to come from different fragments, then Miklau and Suciu show the following polynomial time bounds.

Theorem 3.1 (Theorem 3 in [34]). *The following problems are in P:*
(1) S-Containment *of* TPQ$(/, *)$ *in* TPQ$(/, //, *)$*;*
(2) S-Containment *of* TPQ$(/, //, *)$ *in* TPQ$(/, *)$*;*
(3) Containment *of* TPQ$(/, //, *)$ *in* TPQ$(/, //)$*;*
(4) Containment *of* TPQ$(/, //, *)$ *in* PQ$(/, //, *)$*.*

Notation-wise, we note that Miklau and Suciu [34] assume that “/” is always present in TPQs, which is why they do not explicitly have it in their notation.

In this paper, we prove the following new polynomial time bounds. One of the new results allows child edges in both patterns and, as such, also fills a remaining gap in [34]. (Indeed, in the notation of Miklau and Suciu, they solved the complexity of all containment problems p in q for all combinations of p and q coming from $\mathrm{XP}^{\{[],//,*\}}$, except the case where p is from $\mathrm{XP}^{\{//,*\}}$.)

Theorem 3.2. *The following problems are in P:*
(1) Containment *of* PQ$(/, //, *)$ *in* TPQ$(/, //, *)$*;*
(2) Containment *of* TPQ$(//, *)$ *in* TPQ$(/, //, *)$*;*
(3) Containment *of* TPQ$(/, //, *)$ *in* TPQ$(//, *)$*;*
(4) W-Containment *of* TPQ$(/, *)$ *in* TPQ$(/, //, *)$

Cases (3) and (4) are not very difficult. Case (3) is proved by a rather standard argument involving canonical trees and case (4) follows almost directly from the corresponding S-Containment problem (Theorem 3.1(1)). Cases (1) and (2) are much more involved, however. Here it is useful to see how the patterns are divided into *islands*, where an island is a part a pattern that is connected by child edges. In case (2) we first consider a tree $t \in L(p)$ in which all letters of p are separated by long paths of nodes labelled by some fresh letter. Observe that if in some island of q there are two non-wildcard nodes either on different depth or labelled by different letters then q cannot be embedded into t. Thus it remains to consider patterns q whose islands have non-wildcard nodes only on one depth and all labelled by the same letter. This case can be solved by dynamic programming. Case (1) is the most complex one. Let p^{top} and q^{top} be the topmost islands of p and q, respectively. We first check whether $L(p^{top}) \subseteq L(q^{top})$.

DTD:	fixed	not fixed
PQ	P [4.1(1)]	
TPQ(/)	**NP-c [4.2(2)]**	
TPQ(//)	**P [4.1(2)]**	NP-c [4.2(1)]
TPQ(/,//)	**NP-c [4.2(2)**,(3)]	
TPQ(/,*)	NP-c [4.2(2),(3)]	
TPQ(//,*)	**P [4.1(2)]**	NP-c [4.2(1),(3)]
TPQ(/,//,*)	NP-c [4.2(3)])	

Table 2: Complexity of satisfiability for fragments of TPQs. Results in bold are new, to the best of our knowledge.

If yes, it is easy to reduce the instance to instances for subpatterns, which we solve recursively. If no, the crux is to consider the pattern p' obtained from p by replacing all letters in its topmost island into wildcards. In this case a nontrivial (and maybe even surprising) observation is that $L(p) \subseteq L(q)$ if and only if $L(p') \subseteq L(q)$ (after the w.l.o.g. assumption that no leaf of q^{top} contains a wildcard); again, we can continue by recursion with the simpler pattern p'.

Finally, we also strengthen the coNP lower bound from Miklau and Suciu to the case where the right pattern only uses child edges, but the lower bound only holds for weak containment. For strong containment, the complexity is in P by Theorem 3.2.

Theorem 3.3. *The following problems are coNP-complete:*
(1) Containment *of* TPQ$(/, //)$ *in* TPQ$(/, //, *)$ *[34];*
(2) W-Containment *of* TPQ$(/, //)$ *in* TPQ$(/, *)$*.*

Proof sketch. The crux of the lower bound proof for case (2) is the non-trivial behavior of the patterns in Figure 2. Notice that $L_s(Y) \subseteq L_s(T) \cup L_s(F)$, and that there exists a tree in $L_s(Y)$ and $L_s(T)$ but not in $L_s(F)$ (namely, t_{true}) and there exists a tree in $L_s(Y)$ and $L_s(F)$ but not in $L_s(T)$ (namely, any tree represented by t_{false}). These properties can be used to adapt the coNP-hardness proof of Theorem 4 in [34] to show that *weak* containment of TPQ$(/, //)$ in TPQ$(/, *)$ is coNP-hard. Details are in the appendix. □

Regarding the title of the paper, when one considers TPQs to have four distinctive features (branching, wildcards, child edges, and descendant edges) and one considers both weak and strong containment, one arrives at a rather large[3] amount of different cases to consider for the containment problem. Each of these cases boils down to a theorem mentioned here. We summarize all cases in Table 1.

4. SATISFIABILITY WITH SCHEMA

We now turn to containment problems that take schema information into account. The simplest such problems are satisfiability problems. We provide two polynomial-time results.

Theorem 4.1. *The following problems are in P:*
(1) Satisfiability *of* PQ$(/, //, *)$ *w.r.t. a DTD [5];*

[3]About 2 × 144, but it depends on the definition of “different”.

<table>
<tr><th></th><th>PQ</th><th>TPQ(/,//)</th><th>TPQ(/,∗)</th><th>TPQ(//,∗)</th><th>TPQ(/,//,∗)</th></tr>
<tr><td>PQ</td><td colspan="2" rowspan="7">P [3.1(3),(4)]</td><td colspan="3" rowspan="3">P [3.1(1), 3.2(1),(2),(4)]</td></tr>
<tr><td>TPQ(/)</td></tr>
<tr><td>TPQ(//)</td></tr>
<tr><td>TPQ(/,//)</td><td>coNP-c [3.3(2)] / P [3.1(2)]</td><td rowspan="4">P [3.2(3)]</td><td>coNP-c [3.3(1)]</td></tr>
<tr><td>TPQ(/,∗)</td><td>P [3.2(4)] / P [3.1(2)]</td><td>P [3.2(4)] / P [3.1(1)]</td></tr>
<tr><td>TPQ(//,∗)</td><td>P [3.2(2)] / P [3.1(2)]</td><td>P [3.2(2)]</td></tr>
<tr><td>TPQ(/,//,∗)</td><td>coNP-c [3.3(2)] / P [3.1(2)]</td><td>coNP-c [3.3(1)]</td></tr>
</table>

Table 1: Complexities for all combinations of $\mathcal{F}_1$ in $\mathcal{F}_2$, where $\mathcal{F}_1$ and $\mathcal{F}_2$ are fragments of TPQ. When two complexities are listed, then the left one is for weak containment and the right one for strong containment. Results in bold are new, to the best of our knowledge.

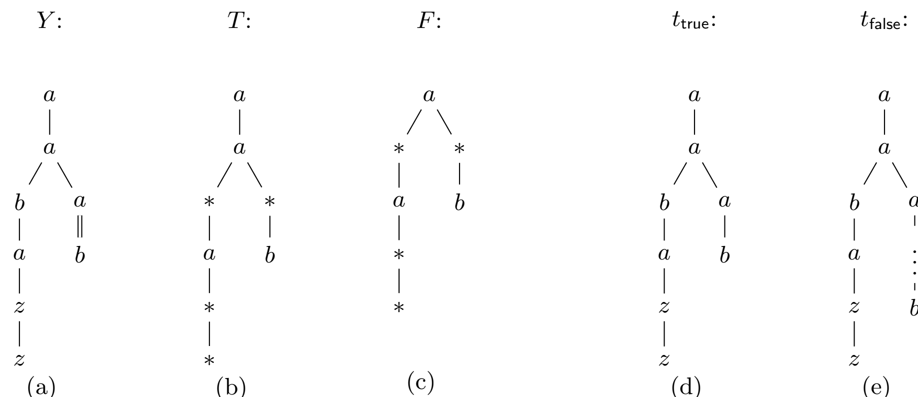


Figure 2: Gadgets for the proof of Theorem 3.3(2).

(2) Satisfiability *of TPQ(//,∗) w.r.t. a fixed DTD.*

Case (1) immediately follows from a stronger result of Benedikt et al. (Theorem 4.1 in [5]) that shows that the problem remains in P even if unions are added. This case can also be solved by a very simple intersection test of tree automata. Case (2) of the above theorem was recently proved for an *injective* semantics of TPQs over trees [16]. The proof can be adapted for the non-injective semantics we consider here.

Furthermore, in case (2) of the above Theorem, it is crucial that the DTD is fixed. Indeed, Wood [43] proved that it is NP-complete to decide whether the language of a given regular expression e over alphabet Σ contains a word that has every letter from Σ.[4] This means that it is already NP-hard to decide whether the language defined by the DTD $r \to e$, containing trees of depth one, has a tree which contains every letter from $\Sigma \setminus \{r\}$. The latter property can be easily expressed by a TPQ(//). This implies case (1) of the following theorem:

Theorem 4.2. *The following problems are NP-complete:*

(1) Satisfiability *of TPQ(//) w.r.t. a DTD [43];*
(2) Satisfiability *of TPQ(/) w.r.t. a fixed DTD;*
(3) Satisfiability *of TPQ(/,//,∗) w.r.t. a DTD [5].*

[4] In fact, Wood's result is stronger. It also holds for expressions that are *deterministic* or *one-unambiguous*, as required in DTDs in practice.

We state cases (1) and (2) here for the lower bound and case (3) for the upper bound. We note that Benedikt et al. [5] proved that satisfiability of TPQ(/,∗) with respect to a fixed DTD is NP-complete, which is close to Theorem 4.2(2). In fact, the hardness proof from Benedikt et al. can be adapted so that it does not require wildcard [35], which would also imply Theorem 4.2(2). However, the construction we present here can also be used to show Theorem 6.3, which we do not know how to do by tweaking the proof in [5]. We present a complete overview of the complexity of satisfiability in Table 2. We provide a proof sketch of case (2) of Theorem 4.2.

Proof sketch (case 2). The proof is by reduction from 3-partition, which is known to be NP-complete and is defined as follows. Given is a number $B \in \mathbb{N}$ (in unary) and a multiset S of integers strictly between $\frac{B}{2}$ and $\frac{B}{4}$ (also in unary). The question is to determine whether it is possible to partition S into $\frac{|S|}{3}$ triples so that the sum of the numbers in each triple is B. We note that the question makes sense only when $|S|$ is divisible by 3, and the sum of all numbers in S is $B \cdot \frac{|S|}{3}$.

We first reduce the problem to a problem that will be more convenient for us and which we call 4-partition. In this problem we are given a number 2^K for some $K \in \mathbb{N}$, and a multiset S' of cardinality $4 \cdot 2^L$ for some $L \in \mathbb{N}$, containing positive integers. The question to answer is whether one can partition S' into $\frac{|S'|}{4}$ sub-multisets so that the sum of numbers in each of them

	strong	weak	DTD fixed
PQ	P [5.1]		**P** **[5.1]**
TPQ(/, //)	P [5.1]		
TPQ(/, ∗)	P [5.1]	EXPTIME-c [5.2]	
TPQ(//, ∗)	P [5.1]		
TPQ(/, //, ∗)	EXPTIME-c [5.2]		

Table 3: Complexity of validity for fragments of TPQs w.r.t. a DTD.

is 2^K. In the appendix, we prove that 4-PARTITION is NP-complete.

We now reduce 4-PARTITION to our problem. To this end, consider an instance 2^K, S' (with $|S'| = 4 \cdot 2^L$) of 4-PARTITION. We construct a fixed DTD d and pattern $p \in \mathsf{TPQ}(/)$ such that p is strongly satisfiable w.r.t. d if and only if there is a solution to the given instance of 4-PARTITION.

The DTD is very simple: it just says that the alphabet is $\{a, b, c, d, e\}$ where each node labelled by a has exactly two children, and each other node is a leaf.

We define sets T_i of (unordered) trees, inductively on i, as follows. The set T_0 consists of the four trees with one node, with a label from $\{b, c, d, e\}$. When T_i is known, we define T_{i+1} to be all trees consisting of a root labelled by a, which is attached to two different unordered subtrees from T_i. Notice that each tree in T_i is perfectly balanced: each leaf is at depth exactly i. The trees in T_i can be also considered as tree patterns. Such pattern strongly embeds into exactly one unordered tree (itself) when taking the DTD restriction into account. Indeed, we cannot add any other node, since each a may have only two children and the leaf labels occur only at the leaves. Furthermore, we cannot merge any two subtrees since they always have to be different. We fix the smallest number M for which $|T_M| \geq 2^{K+L}$. We have that

$$|T_0| = 4, \qquad |T_{i+1}| = \frac{1}{2}|T_i|(|T_i| - 1).$$

Notice that, as i increases, the number $|T_i|$ is almost squared, so $|T_i|$ grows double-exponentially faster than i. This means that, for sufficiently large instances, M will be smaller than $K + L$, so we can use 2^{K+L} trees from T_M to construct our pattern, and it will be polynomial (recall that 2^K is given in unary).

We are now ready to describe the pattern p. From the root we create $|S'|$ paths, each of length L (each of them corresponds to one number from S'). At the end of the path corresponding to a number k we create k paths of length K. At the end of each such path, we attach a tree from T_M that occurs nowhere else in the whole pattern p. Figure 3 has a graphical presentation of p.

This concludes the reduction. We prove in the appendix that $L(d) \cap L_s(p) \neq \emptyset$ if and only if $L(d) \cap L_w(p) \neq \emptyset$ if and only if the instance of 4-PARTITION has a solution. □

5. VALIDITY WITH SCHEMA

For the validity problem, we do not present any new deep results, only an observation that closes a small gap

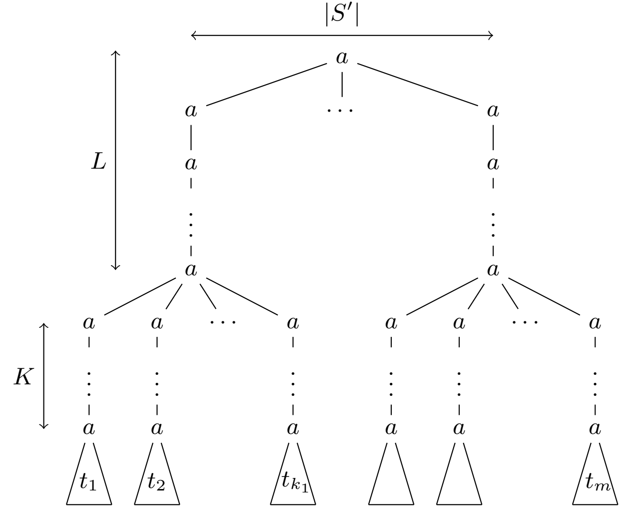


Figure 3: Structure of the pattern p in the proof of Theorem 4.2(2). All the t_j are different trees.

between the results that were presented by Hashimoto et al. [27] and Björklund et al. [9]. We summarize the strongest results here, for the sake of completeness and present a complete overview in Table 3.

THEOREM 5.1. *The following problems are in P:*
(1) VALIDITY *of* PQ(/, //, ∗) *w.r.t. a DTD [27];*
(2) VALIDITY *of* TPQ(/, //) *w.r.t. a DTD [27];*
(3) S-VALIDITY *of* TPQ(/, ∗) *w.r.t. a DTD [27];*
(4) VALIDITY *of* TPQ(//, ∗) *w.r.t. a DTD [27];*
(5) VALIDITY *of* TPQ(/, //, ∗) *w.r.t. a fixed DTD.*

Cases (1–4) are proved in Theorem 3 of [27]. Case (5) can be efficiently solved due to the constant language defined by the DTD; we prove it in the appendix. The fixed DTD in case (5) is rather crucial. Without it, weak validity for TPQ(/, ∗) immediately jumps to EXPTIME, as the following theorem shows.

THEOREM 5.2 (THEOREM 12 IN [9]). W-VALIDITY *of* TPQ(/, ∗) *w.r.t. a DTD is EXPTIME-complete.*

From this theorem, we also immediately know that S-VALIDITY of TPQ(/, //, ∗) with respect to a DTD is EXPTIME-complete.

6. CONTAINMENT WITH SCHEMA

The containment problem with schema information is the most general problem we consider. The solutions for Theorem 6.1(4) and Theorem 6.6 are technically most involved results of this paper.

6.1 Polynomial Time Fragments

THEOREM 6.1. *The following are in P:*
(1) CONTAINMENT *of* PQ(/, //, ∗) *in* PQ(/, //)*;*
(2) CONTAINMENT *of* PQ(/, //, ∗) *in* TPQ(//, ∗)*;*
(3) S-CONTAINMENT *of* PQ(/, //, ∗) *in* TPQ(/, ∗)*;*
(4) W-CONTAINMENT *of* PQ(/, //, ∗) *in* TPQ(/)*;*
all w.r.t. a DTD.

The proofs for cases (1), (2), and (3) rely on the following observation.

	PQ (/), PQ (//), PQ (/,//)	PQ (/,*)	PQ (//,*)	PQ (/,//,*)
PQ (/), PQ (//) PQ(/, //, ∗)	**P [6.1(1)]** and [36]	**EXPT.-c [6.6(1),(2)] / P [6.1(3)]**	**P [6.1(2)]**	**EXPT.-c [6.6(1),(2)] / EXPT.-c [6.6(3),(4)]**
TPQ (/) TPQ (//) TPQ(/, //, ∗)	**coNP-c [6.3,6.4(1)]**	**EXPT.-c [6.6(1),(2)] / coNP-c [6.3(1),(2), 6.4(3)]**	**coNP-c [6.3(3),(4),6.4(2)]**	

Table 4: Complexity of containment TPQs w.r.t. a DTD; right pattern is a PQ. When two complexities are listed, the first is for weak containment and the second for strong containment. The results hold for the case where the DTD is part of the input and for the case where the DTD is fixed. Fields in bold have at least one new result, to the best of our knowledge.

	TPQ (/)	TPQ (//)	TPQ (/,//)	TPQ (/,*)	TPQ (//,*)	TPQ (/,//,*)
PQ (/) PQ (//) PQ(/, //, ∗)	**P [6.1(4)] / P [6.1(3)]**	**P [6.1(2)]**	in EXPT.	**EXPT.-c [6.6(1),(2)] / P [6.1(3)]**	**P [6.1(2)]**	**EXPT.-c [6.6(1),(2)] / EXPT.-c [6.6(3),(4)]**
TPQ (/) TPQ (//) TPQ (/,//,*)	**coNP-h [6.3] / coNP-c [6.3,6.4(3)]**	**coNP-c [6.3,6.4(2)]**	**coNP-h [6.3]**	**EXPT.-c [6.6(1),(2)] / coNP-c [6.3,6.4(3)]**	**coNP-c [6.3,6.4(2)]**	

Table 5: Complexity of containment TPQs w.r.t. a DTD; right pattern is a TPQ. Notation and remarks are the same as in Table 4.

Observation 6.2. *Let Σ be a finite alphabet and let q be a query in (1) PQ$(/, //)$; (2) TPQ$(//, *)$; or in (3) TPQ$(/, *)$. Then we can construct in polynomial time a non-deterministic tree automaton for the language $\mathcal{T}_\Sigma \setminus L_s(q)$.*

Cases (1)–(3) of Theorem 6.1 now follow by reduction to the emptiness problem of non-deterministic tree automata. Indeed, for $p \in$ PQ$(/, //, *)$ and DTD d one can construct in polynomial time a non-deterministic tree automaton for $L_s(p) \cap L(d)$. This means that, in all three cases, we can obtain in polynomial time a non-deterministic tree automaton that accepts the empty language if and only if $L_s(p) \cap L(d) \subseteq L_s(q)$. Since emptiness testing of non-deterministic tree automata is in P, this gives a P solution to the problem. The P upper bounds for weak containment in cases (1) and (2) are immediate from Observation 2.3. We note that Neven and Schwentick [36] already showed that containment of PQ$(/, //)$ in PQ$(/, //)$ w.r.t. DTD is in P, which is very close to Theorem 6.1(1).

Case (4), however involves a non-trivial argument. Let $p \in$ PQ$(/, //, *)$, $q \in$ TPQ$(/)$, and let d be a DTD. Notice that the automaton recognizing $\mathcal{T}_\Sigma \setminus L_w(q)$ may be of exponential size, thus we have to proceed differently from cases (1)–(3). In a first stage, whenever in q we have two siblings with the same label, we merge them. Although this changes $L_w(q)$, it is easy to see that it does not influence the containment question. In a second stage, we remove redundant subqueries of q. Namely, it may happen that we can remove some subquery of q obtaining some q', so that whenever q' can be embedded into a tree from $L(d)$, then the whole q can be embedded as well. After this cleaning stage, we have one of two mutually exclusive situations. One possibility is that the pattern q is a path (or is very similar to a path); then we have case (1). The opposite case is that in q we have some branching. Then we can prove that the containment never holds. Indeed, to obtain a tree t from $(L_w(p) \cap L(d)) \setminus L_w(q)$ we are quite restricted only while arranging the path into which p will be embedded. However, into this path at most one path of q embeds. Outside of this path in t we can place arbitrary subtrees satisfying d, so we place there subtrees into which the rest of q cannot be embedded. Such subtrees exist, since otherwise the rest of q would be redundant, but all redundant subqueries were already removed during the second stage.

In fact the most difficult part is to check whether q is (weakly) equivalent to its part q' w.r.t. our DTD d. At first glance this looks hopeless, as this is just the W-Containment problem of TPQ(/) in TPQ(/) w.r.t. a DTD, which by Theorem 6.3 is in general coNP-hard. Hopefully, our pattern q is not arbitrary: there are no two siblings labelled the same (thanks to the first stage). The crux of the proof is to reduce the equivalence problem in this special case to multiple smaller instances of the (slightly generalized) containment question of PQ(/) in TPQ(/) w.r.t. a DTD. Thanks to that, we can proceed by dynamic programming.

6.2 Hard Fragments

We will prove that there are two ingredients that make the containment problem w.r.t. DTDs hard, even when the DTD is constant. These two ingredients are
(1) branching in the left pattern and
(2) right patterns of the form $//a/*/*/\cdots/*/b$.
Branching in the left pattern immediately renders containment with schema coNP-hard. The underlying reason is that already satisfiability of TPQ(/) with a fixed DTD is NP-complete (Theorem 4.2); and that our proof can be adapted for the case where the left pattern only has //-edges. The second source of hardness is when the right pattern can express "there is an a-node which is (exactly) k levels above a b-node". In this case the com-

plexity even jumps to EXPTIME, which is already the highest possible complexity class for TPQ containment. We found it rather surprising that such a seemingly inexpressive query makes containment so hard even when the DTD is fixed.

THEOREM 6.3. *The following are coNP-hard:*
(1) CONTAINMENT *of* TPQ(/) *in* PQ(/)*;*
(2) CONTAINMENT *of* TPQ(//) *in* PQ(/)*;*
(3) CONTAINMENT *of* TPQ(/) *in* PQ(//)*;*
(4) CONTAINMENT *of* TPQ(//) *in* PQ(//)*;*
all w.r.t. a fixed DTD.

For cases (1)–(3), it does not follow from Observation 2.3 that weak and strong containment have the same complexity, but we merged them for succinctness. The proofs are very similar. Cases (1) and (3) are immediate from Theorem 4.2(2). The other cases can be obtained from adapting the proof of Theorem 4.2(2), as follows. Recall that in this proof we were using a DTD ensuring that each internal node (but no leaf) of a tree is labelled by a, and we were reducing an instance of the 4-PARTITION problem into satisfiability of a pattern p that has each leaf on the same depth, say n. We notice that such pattern p is satisfiable if and only if $L_w(p') \cap L(d) \not\subseteq L_w(q)$, where p' is the pattern obtained from p by changing all child edges into descendant edges, and q is a path consisting of $n+1$ nodes labelled by a (connected either by child edges or by descendant edges, depending on the considered fragment). Indeed, any tree in $L_s(p) \cap L(d)$ is in $L_w(p')$ and not in $L_w(q)$ (because of the depth). On the other hand, any tree $t \in L_w(p') \cap L(d) \setminus L_w(q)$ has depth at most n, so an embedding of p' into t maps every descendant edge into a child edge, and hence p embeds into t as well. This gives a reduction to (the negation of) the problem considered in case (2) or (4).

For many cases, we can also prove coNP-completeness:

THEOREM 6.4. *The following are coNP-complete:*
(1) CONTAINMENT *of* TPQ(/, //, ∗) *in* PQ(/, //)*;*
(2) CONTAINMENT *of* TPQ(/, //, ∗) *in* TPQ(//, ∗)*;*
(3) S-CONTAINMENT *of* TPQ(/, //, ∗) *in* TPQ(/, ∗)*;*
all w.r.t. a DTD.

These coNP upper bounds follow from two observations. The first one is Observation 6.2. The second one concerns S-SATISFIABILITY of TPQ(/, //, ∗) in the language of a tree automaton:

OBSERVATION 6.5. SATISFIABILITY *of a* TPQ(/, //, ∗) *w.r.t. a non-deterministic tree automaton is in NP.*

The proof of the observation is a rather standard "small model" argument. The result follows immediately from, e.g., Theorem 8 in [7]. We finish the proof of Theorem 6.4 as follows. Observation 6.2 says that, in all three cases, we can build a non-deterministic tree automaton for the language of all trees that satisfy the DTD and do not satisfy the right pattern. In all three cases, containment holds if and only if the left pattern is not satisfiable w.r.t. this tree automaton, which is in coNP by Observation 6.5.

The following theorem involves the technically most difficult hardness proof of the paper. The proof goes through a variant of tiling problems that we call *triomino tiling*. Whereas standard tiling problems [42] use *horizontal* and *vertical* constraints, triomino tiling unifies these to a single kind of constraint in the shape of an *L-triomino*.[5] A triomino constraint restricts how a cell, its horizontal neighbor and its vertical neighbor can be tiled. Our main reason to go through triomino tiling is because it allows us to check the horizontal and vertical constraint *at the same time, and using only the right pattern*. Since the triomino tiling problem may be useful in its own respect, we briefly present it here. The main technical difficulty in the proof is in the reduction from triomino tiling to containment, however.

More precisely, a *triomino tiling system* is a tuple $S = (T, C, t_f)$ where T is a finite set of *tile types*, $C \subseteq T^3$ a set of *triomino constraints*, and $t_f \in T$ is a *final tile*. An instance of *corridor triomino tiling* consists of a word $s \in T^*$. The question is if it is possible to find an $m \in \mathbb{N}$ such that each cell in the $|s| \times m$ rectangle can be assigned a tile type and following constraints are satisfied:

- the bottom row is tiled by s;
- the last tile is t_f; and
- for each triple (c_1, c_2, c_3) such that c_1 is a cell, c_2 its right neighbor, and c_3 its top neighbor, we have that $(t_1, t_2, t_3) \in C$, where t_i is the type that is assigned to c_i for every $i = 1, 2, 3$.

We present the problem in more detail in the appendix. In our hardness proof, we actually reduce from a two-player variant of corridor triomino tiling, tweaked appropriately for our situation. This game variant of the problem is EXPTIME-complete.

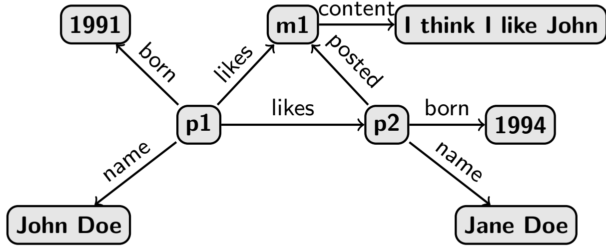
THEOREM 6.6. *The following are EXPTIME-complete:*

(1) W-CONTAINMENT *of* PQ(/) *in a* PQ(/, ∗)*;*
(2) W-CONTAINMENT *of* PQ(//) *in a* PQ(/, ∗)*;*
(3) S-CONTAINMENT *of* PQ(/) *in a* PQ(/, //, ∗)*;*
(4) S-CONTAINMENT *of* PQ(//) *in a* PQ(/, //, ∗)*;*
all w.r.t. a fixed DTD.

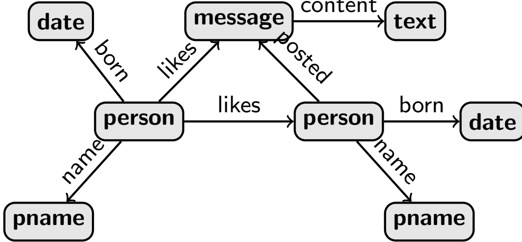
7. TREES VERSUS GRAPHS

We explain how the complexity results in the paper can be interpreted in the context of graph databases. We consider two possible abstractions of graph databases in this section. The first is a node-labelled graph and the second is a node- and edge-labelled graph. We note that the standard models of graph databases in the literature are edge-labelled rather than node-labelled (e.g., [3, 32, 33]), but this does not make a difference when transferring our kind of results. In the node- and edge-labelled model, edges will be triples consisting of two nodes and a label (as usual) and nodes will be labelled by a function called type. The intuition is that if a graph represents a social network, for example, we assume that nodes already have information on whether they represent a person, message, or photo. The relations between nodes are modelled as directed edges, e.g., *person x likes message y* is modelled as a directed edge from node x to node y labelled likes, and nodes x and y carry the type person and message respectively. An example graph and its abstraction with types is presented in Figure 4.

[5]Readers familiar with Tetris may know the term *tetromino*, which are the "blocks", consisting of four squares, that appear in the game. A triomino has three squares.



(a) A toy graph database about a social network.



(b) Abstraction of the graph database in (a), only bearing node types and edge labels.

Figure 4: A graph database 4(a) and its abstraction as a typed graph 4(b).

7.1 No Schema Information

First we explain the connection to the setting where we do not consider DTDs. Let Γ be a finite alphabet of *types*. A *graph* G over Γ is a triple ($\mathsf{Nodes}(G)$, $\mathsf{Edges}(G)$, type^G) where $\mathsf{Nodes}(G)$ is a finite set of nodes, $\mathsf{Edges}(G) \subseteq (\mathsf{Nodes}(G))^2$, and $\mathsf{type}^G : \mathsf{Nodes}(G) \to \Gamma$. When G is clear from the context, we often also denote type^G as type. It is a *rooted graph* if it has a special node $\mathsf{root}(G) \in \mathsf{Nodes}(G)$.

A *(weak) embedding* m of a TPQ q (over the letters Γ) on a graph G is defined analogously as on trees, except that we consider $\mathsf{type}(m(v))$ instead of $\mathsf{lab}(m(v))$ and, if $(v_1, v_2) \in \mathsf{Desc}(q)$, then we require that there is a directed path in G from $m(v_1)$ to $m(v_2)$. If G is a rooted graph, then m is a *strong embedding* of q if it is a weak embedding on G and $m(\mathsf{root}(q)) = \mathsf{root}(G)$. We can now define the weak and strong graph languages of q analogously as we did over trees. The small difference is that the strong graph language is a set of *rooted graphs*. Once these notions are established, the definitions of S-Containment and W-Containment of TPQs over graphs are again analogous as for trees. The next proposition, which was already observed by Miklau and Suciu [34], says that with these definitions, the W-Containment and S-Containment problems are independent of whether TPQs are interpreted over graphs or over trees.

Proposition 7.1 (Section 5.3 in [34]). *For all fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ of TPQs we consider in this paper, the* W-Containment *(resp.,* S-Containment*) problem of $\mathcal{F}_1$ in $\mathcal{F}_2$ over graphs is the same as the* W-Containment *(resp.,* S-Containment*) problem of $\mathcal{F}_1$ in $\mathcal{F}_2$ over trees.*

The proposition implies that all complexity results in Section 3 also hold for TPQs over node-labelled graphs. If one considers *graph databases* [3], that is, graphs in which the edges are triples coming from $\mathsf{Nodes}(G) \times \Sigma \times \mathsf{Nodes}(G)$, and one defines the semantics of TPQs as in Libkin et al. [32], the proposition holds as well.

7.2 Typed Graphs and Schema Information

We extend Proposition 7.1 so that it also connects to satisfiability w.r.t. DTDs. We have to adjust DTDs a little bit since they are not the most natural model for meta-information on graphs. We take some inspiration from Shape Expressions [12,40]. However, a crucial difference with [12] is that we assume that nodes already have types. That is, in a social network, nodes already have types such as person or message associated to them.

We feel that there are two natural ways of interpreting DTDs over typed graphs. In the first, the DTD only reasons about the types of nodes. In the second we consider graphs that have labelled edges and nodes with associated types and the DTD reasons about pairs consisting of node- and edge labels. We explain both variants next.

Nodes Only. In the first variant, we say that a graph G satisfies a DTD (Γ, d, S_d) *under nodes-only semantics* if, for every node u of G with $\mathsf{type}(u) = a$ and neighbors $\{v_1, \ldots, v_n\} = \{v \mid (u, v) \in \mathsf{Edges}(G)\}$, there exists a permutation σ of $\{1, \ldots, n\}$ such that $\mathsf{type}(v_{\sigma(1)}) \cdots \mathsf{type}(v_{\sigma(n)}) \in L(d(a))$. Finally, if $\mathsf{root}(G)$ exists, we require that $\mathsf{type}(\mathsf{root}(G)) \in S_d$.

Formally, this means that we consider the neighbors of a node in a graph to be unordered and, therefore, we also consider the regular expressions in DTDs to define an unordered language. Here, we need to make a brief discussion about complexity. Testing if a given word (or multiset of labels) is in the language of an unordered regular expression, that is, testing if the above permutation σ exists, is NP-complete [30]. In fact, this is a reason why unordered regular expressions do not seem suitable out of the box for schemas for unordered XML and why the research on the design of schema languages for unordered XML considers restricted regular expressions, see [11]. However, NP-completeness of this problem does not influence the complexity of TPQ containment. The reason is that, since tree pattern queries are already unordered, it does not matter for the complexity of TPQ containment whether they are interpreted on ordered or unordered structures.

We define the nodes-only semantics of TPQs to be the same one as we used for Proposition 7.1. Then we have the following correspondence between graphs and trees:

Proposition 7.2. *For all fragments $\mathcal{F}$ of TPQs we consider in this paper, under the nodes-only semantics of DTDs and TPQs on graphs, the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ with respect to DTDs over graphs is the same as the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ over trees.*

We note that this proposition requires that the DTDs are *reduced*, which is a condition we assume throughout the paper. Unfortunately there is no longer a strong correspondence for the Validity and Containment problems. Actually, we think that already for Validity the complexities may even be different. We comment on this in the conclusions.

Nodes and Edges. The second way of tweaking DTDs for graphs deals with node- and edge labels and

is a little bit more technical. First we say how we see such graphs. We let Γ be a finite alphabet of *types* as before and we additionally consider a finite alphabet Σ of *edge labels*, disjoint from Γ. A *typed graph* over (Σ, Γ) is a tuple $G = (\mathsf{Nodes}(G), \mathsf{Edges}(G), \mathsf{type})$ where $\mathsf{Nodes}(G)$ and $\mathsf{type} : \mathsf{Nodes}(G) \to \Gamma$ are as before, but $\mathsf{Edges}(G) \subseteq (\mathsf{Nodes}(G) \times \Sigma \times \mathsf{Nodes}(G))$.

For typed graphs, we consider DTDs that also reason about pairs of node- and edge labels. More precisely, it uses the alphabet $\Gamma \cup (\Sigma \times \Gamma)$ and its rules are of a restricted form. We first provide an example.

Example 7.3. Consider the rules

$$
\begin{array}{ll}
\mathsf{person} \to & (\mathsf{born}, \mathsf{date})(\mathsf{name}, \mathsf{pname})(\mathsf{posted}, \mathsf{message})^* \\
 & (\mathsf{likes}, \mathsf{message})^*(\mathsf{likes}, \mathsf{person})^* \\
(\mathsf{born}, \mathsf{date}) & \to \mathsf{date} \\
(\mathsf{name}, \mathsf{pname}) & \to \mathsf{pname} \\
(\mathsf{posted}, \mathsf{message}) & \to \mathsf{message} \\
(\mathsf{likes}, \mathsf{message}) & \to \mathsf{message} \\
(\mathsf{likes}, \mathsf{person}) & \to \mathsf{person} \\
\mathsf{message} & \to (\mathsf{content}, \mathsf{text}) \\
(\mathsf{content}, \mathsf{text}) & \to \mathsf{text}
\end{array}
$$

When we don't explicitly write a rule for a symbol s, we implicitly assume to have the rule $\mathsf{s} \to \varepsilon$. Notice the difference between rules that only have a type on the left and rules that have a pair on the left. Intuitively, the former rules define the kind of outgoing edges that are allowed from nodes in the graph and the latter rules say, for each edge, the type of node they should point to. (We have these two kinds of rules due to the limited expressive power of DTDs; a DTD can define only one right-hand-side per left-hand symbol.) So, the reason is that a DTD can only control the 1-neighborhood of a node, not the 2-neighborhood.

On a typed graph, the first six rules express that each person has an outgoing edges labelled born that points to a node of type date and an outgoing edge labelled name that points to a node of type pname. Furthermore, a person can have outgoing edges labelled likes that can point to nodes of type message or person and it can have outgoing edges labelled posted that point to nodes of type message. The last two rules express that each message has an outgoing content edge to a text-node. The typed graph in Figure 4(b) satisfies the DTD.

Formally, we say that a DTD is a *graph DTD* if it is of the form $(\Gamma \cup (\Sigma \times \Gamma), d, S_d)$ and the rules in d are only of the form $a \to r$ with $a \in \Gamma$ in $r \in \mathcal{R}(\Sigma \times \Gamma)$ or of the form $(e, a) \to a$ for $e \in \Sigma$ and $a \in \Gamma$.

A typed graph G satisfies d *under nodes/edges semantics* if all the following hold:

- for every node u of G with $\mathsf{type}(u) = a$ and incident edges $\{e_1, \ldots, e_n\} = \{(u, a_i, v_i) \mid (u, a_i, v_i) \in \mathsf{Edges}(G)\}$, there exists a permutation σ of $\{1, \ldots, n\}$ such that $(a_{\sigma(1)}, \mathsf{type}(v_{\sigma(1)})) \cdots (a_{\sigma(n)}, \mathsf{type}(v_{\sigma(n)})) \in L(d(a))$;
- for every edge $e = (u, a, v)$ of G, we have that $\mathsf{type}(v) \in L(d((a, \mathsf{type}(v))))$; and
- if $\mathsf{root}(G)$ exists, then $\mathsf{type}(\mathsf{root}(G)) \in S_d$.

We now explain when a TPQ matches a graph under nodes/edges semantics. Intuitively, this is done by translating the graph to a node-labelled graph and then take the same semantics as we already defined. Formally, let $G = (\mathsf{Nodes}(G), \mathsf{Edges}(G), \mathsf{type}^G)$. We associate to G a node-labelled graph $G_N = (\mathsf{Nodes}(G_N), \mathsf{Edges}(G_N), \mathsf{type}^{G_N})$ over $\Gamma \cup (\Sigma \times \Gamma)$, which is obtained from G as follows:

- $\mathsf{Nodes}(G_N) := \mathsf{Nodes}(G) \uplus \{n_e \mid e \in \mathsf{Edges}(G)\}$;
- $\mathsf{Edges}(G_N) := \{(u, n_e), (n_e, v) \mid e = (u, a, v) \in \mathsf{Edges}(G)\}$;
- for every $u \in Nodes(G)$, $\mathsf{type}^{G_N}(u) := \mathsf{type}^G(u)$;
- for every $n_e \in \mathsf{Nodes}(G_N)$ with $e = (u, a, v)$, we define $\mathsf{type}^{G_N}(n_e) := (a, \mathsf{type}^G(v))$; and
- if $\mathsf{root}(G)$ exists, then $\mathsf{root}(G_N) := \mathsf{root}(G)$.

We now say that G satisfies a TPQ q *under nodes/edges semantics* if G_N satisfies q. Notice that, as in graph DTDs, the TPQ q uses letters from $\Gamma \cup (\Sigma \times \Gamma)$.

Proposition 7.4. *For all fragments $\mathcal{F}$ of TPQs we consider in this paper, under the nodes/edges semantics of DTDs and TPQs on graphs, the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ with respect to DTDs over graphs is the same as the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ over trees.*

Similarly to Proposition 7.2, this proposition does not generalize to Validity or Containment. We are still working on the question which *complexity* results carry over.

8. DISCUSSION AND FUTURE WORK

We made significant progress in the investigation of the complexity of containment of TPQs over trees. The only table that still contains cases that are not yet classified as either "tractable" or complete for a complexity class containing NP or coNP is Table 5.

In Section 7 we discussed a close correspondence between the containment problem for trees and the problem for graphs. In this respect, our biggest open question is to which extent the results for trees can be carried over to graphs for the containment and validity problems with schema information. We know that the validity problem is not the same for trees than for graphs. For example, over trees, the query $a//b$ is valid for the DTD with rules $a \to a + b$ and $b \to \varepsilon$, but this is not true over graphs. The challenge consists of dealing with schemas that are recursive, i.e., allow cycles in graphs.

In this work we presented a manageable amount of existing and newly discovered results that seem to be quite powerful for classifying the complexity of tree pattern containment problems. Research is not about considering large amounts of cases and solving all of them but, only as an illustration, when one would consider all variations of TPQ fragments, weak/strong containment, with fixed DTD or not as "different", then the results mentioned in this paper classify 856 different cases out of 916 total as either in P or complete for a higher complexity class.[6] When one does not care about completeness for the higher class, then 892 cases are classified. Given the rich literature on tree pattern containment, we feel that, in addition to the newly obtained technical results, our work also contributes in making a clearer picture of the problem.

Acknowledgments. We thank Filip Murlak for pointing us to [16].

[6] OK, we admit that we are working on the insight that classifies the remaining ones.

9. REFERENCES

[1] M. Arenas, J. Daenen, F. Neven, M. Ugarte, J. Van den Bussche, and S. Vansummeren. Discovering XSD keys from XML data. In *SIGMOD*, pages 61–72, 2013.

[2] M. Arenas and L. Libkin. XML data exchange: Consistency and query answering. *J. ACM*, 55(2), 2008.

[3] P. Barceló. Querying graph databases. In *PODS*, pages 175–188, 2013.

[4] P. Barceló, L. Libkin, A. Poggi, and C. Sirangelo. XML with incomplete information. *J. ACM*, 58(1):4, 2010.

[5] M. Benedikt, W. Fan, and F. Geerts. XPath satisfiability in the presence of DTDs. *J. ACM*, 55(2), 2008.

[6] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. XML Path Language (XPath) 2.0. W3C, January 2007. http://www.w3.org/TR/2007/REC-xpath20-20070123/.

[7] H. Björklund, W. Martens, and T. Schwentick. Optimizing conjunctive queries over trees using schema information. In *MFCS*, pages 132–143, 2008.

[8] H. Björklund, W. Martens, and T. Schwentick. Conjunctive query containment over trees. *JCSS*, 2010.

[9] H. Björklund, W. Martens, and T. Schwentick. Validity of tree pattern queries with respect to schema information. In *MFCS*, pages 171–182, 2013.

[10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An XML query language. W3C, 2007. http://www.w3.org/TR/2007/REC-xquery-20070123/.

[11] I. Boneva, R. Ciucanu, and S. Staworko. Schemas for unordered XML on a DIME. *CoRR*, abs/1311.7307, 2013.

[12] I. Boneva, J. E. L. Gayo, S. Hym, E. G. Prud'hommeau, H. R. Solbrig, and S. Staworko. Validating RDF with shape expressions. *CoRR*, abs/1404.1270, 2014. To appear in ICDT 2015.

[13] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language XML 1.0 (fifth edition). W3C, 2008. http://www.w3.org/TR/2008/REC-xml-20081126/.

[14] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. In *ICDE*, pages 913–922, 2008.

[15] B. S. Chlebus. Domino-tiling games. *JCSS*, 32(3):374–392, 1986.

[16] C. David, N. Francis, and F. Murlak. Consistency of injective tree patterns. To appear in FSTTCS 2014.

[17] C. David, A. Gheerbrant, L. Libkin, and W. Martens. Containment of pattern-based queries over data trees. In *ICDT*, pages 201–212, 2013.

[18] S. DeRose, E. Maler, and R. Daniel. XML pointer language (XPointer) version 1.0. W3C, 2001.

[19] S. DeRose, E. Maler, D. Orchard, and N. Walsh. XML linking language (XLink) version 1.1. W3C, 2010.

[20] D. Figueira. *Reasoning on Words and Trees with Data*. PhD thesis, École Normale Supérieure de Cachan, 2010.

[21] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of XPath queries. *J. ACM*, 55(1), 2008.

[22] S. Gao, C. M. Sperberg-McQueen, H. Thompson, N. Mendelsohn, D. Beech, and M. Maloney. W3C XML Schema Definition Language (XSD) 1.1 part 1: Structures. W3C, 2009. http://www.w3.org/TR/2009/CR-xmlschema11-1-20090430/.

[23] F. Geerts and W. Fan. Satisfiability of XPath queries with sibling axes. In *DBPL*, pages 122–137, 2005.

[24] A. Gheerbrant, L. Libkin and C. Sirangelo. Reasoning About Pattern-Based XML Queries. In *RR*, pages 4–18, 2013.

[25] G. Gottlob, C. Koch, and K. U. Schulz. Conjunctive queries over trees. *J. ACM*, 53(2):238–272, 2006.

[26] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[27] K. Hashimoto, Y. Kusunoki, Y. Ishihara, and T. Fujiwara. Validity of positive XPath queries with wildcard in the presence of DTDs. In *DBPL*, 2011.

[28] M. Kay. XSL Transformations (XSLT) version 2.0. W3C, 2007. http://www.w3.org/TR/2007/REC-xslt20-20070123/.

[29] B. Kimelfeld and Y. Sagiv. Revisiting redundancy and minimization in an XPath fragment. In *EDBT*, pages 61–72, 2008.

[30] E. Kopczynski and A. W. To. Parikh images of grammars: Complexity and applications. In *LICS*, pages 80–89, 2010.

[31] E. V. Kostylev, J. L. Reutter, and D. Vrgoč. Containment of data graph queries. In *ICDT*, pages 131–142, 2014.

[32] L. Libkin, W. Martens, and D. Vrgoč. Querying graph databases with XPath. In *ICDT*, pages 129–140, 2013.

[33] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, 1995.

[34] G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *J. ACM*, 51(1):2–45, 2004.

[35] F. Murlak. Personal communication.

[36] F. Neven and T. Schwentick. On the complexity of XPath containment in the presence of disjunction, DTDs, and variables. *LMCS*, 2(3), 2006.

[37] M. Niewerth and T. Schwentick. Reasoning about XML constraints based on xml-to-relational mappings. In *ICDT*, pages 72–83, 2014.

[38] J. Pérez, M. Arenas, and C. Gutierrez. NSPARQL: A navigational language for RDF. *J. Web Sem.*, 8(4):255–270, 2010.

[39] T. Schwentick. XPath query containment. *Sigmod RECORD*, 33(1):101–109, 2004.

[40] Shape expressions. http://www.w3.org/2001/sw/wiki/ShEx.

[41] B. ten Cate and C. Lutz. The complexity of query containment in expressive fragments of XPath 2.0. *J. ACM*, 56(6), 2009.

[42] P. van Emde Boas. The convenience of tilings. *Complexity, Logic and Recursion Theory* pages 331–363. Marcel Dekker Inc., 1997.

[43] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003. Full version (personal communication).

[44] Q. Zeng, X. Jiang, and H. Zhuge. Adding logical operators to tree pattern queries on graph-structured data. *PVLDB*, 5(8):728–739, 2012.

APPENDIX

For an $a \in \Sigma$ and a DTD (Σ, d, S_d) we denote by d^a the DTD $(\Sigma, d, \{a\})$, so $L(d^a)$ always denotes the set of trees that satisfy the rules of d but have a root with label a.

A. PROOFS FOR SECTION 2: PRELIMINARIES

OBSERVATION 2.3. *For all fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ of TPQs, there is a polynomial time reduction from* S-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ to* W-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ (with or without DTD). If $\mathcal{F}_1$ and $\mathcal{F}_2$ both contain //, then there is also a polynomial time reduction from* W-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ to* S-CONTAINMENT *of $\mathcal{F}_1$ in $\mathcal{F}_2$ (with or without DTD).*

PROOF. We first present a reduction from W-CONTAINMENT to S-CONTAINMENT. Suppose that we have patterns p and q and a DTD d, and we want to check whether $L_w(p) \cap L(d) \subseteq L_w(q)$. Let $\top$ be a letter not appearing in the patterns nor in the DTD. Let S_d be the set of root labels allowed by d. We create new patterns p' and q' by attaching a new $\top$-labelled root above p and q, respectively, using a descendant edge (recall that by assumption // is in our fragment of TPQs). We also consider a DTD d' obtained from d by adding the rule $\top \to S_d$, and changing the set of allowed root labels into $\{\top\}$. We claim that $L_w(p) \cap L(d) \subseteq L_w(q)$ if and only if $L_s(p') \cap L(d') \subseteq L_s(q')$. It remains to prove this claim.

Suppose that $L_w(p) \cap L(d) \subseteq L_w(q)$ and take a tree $t \in L_s(p') \cap L(d')$. Because of the DTD, the root of t has one child, and the subtree t' rooted in that child belongs to $L(d)$. As $t \in L_s(p')$, we have $t' \in L_w(p)$, so by assumption $t' \in L_w(q)$. Then obviously $t \in L_s(q')$.

Oppositely, suppose that $L_s(p') \cap L(d') \subseteq L_s(q')$ and take a tree $t \in L_w(p) \cap L(d)$. Consider t' obtained from t by attaching a new $\top$-labelled root above t. We have $t' \in L_s(p') \cap L(d')$, so $t' \in L_s(q')$, and thus $t \in L_s(q)$.

Without DTD the reduction looks more or less the same.

Now we turn into a reduction from S-CONTAINMENT to W-CONTAINMENT. Suppose that we have patterns p and q and a DTD d, and we want to check whether $L_s(p) \cap L(d) \subseteq L_s(q)$. Let $\top$, $\top_{OK}$, $\top_{BAD}$ be letters not appearing in the patterns nor in the DTD. Let S_d be the set of root labels allowed by d. Let R_p be the set of root labels allowed by p intersected with S_d. Namely, when the root of p is labelled by a letter a, then $R_p = \{a\} \cap S_d$; if this is a wildcard, then $R_p = S_d$. Similarly, let R_q be the set of root labels allowed by q intersected with S_d. We have several cases here.

The first case is that $R_p \subseteq R_q$: whenever the root of p can be mapped into some node, then also the root of q can be mapped to this node. Then we change the root labels of both patterns into $\top$. Also the set of root labels allowed by the DTD becomes $\{\top\}$. The rule for $\top$ will be $\top \to \bigcup_{a \in R_p} L_a$, where in d the rule for a was $a \to L_a$. It is easy to see that $L_s(p) \cap L(d) \subseteq L_s(q)$ if and only if $L_s(p') \cap L(d') \subseteq L_s(q')$, where p', q', d' denote the new patterns and DTD. Moreover, due to the unique root label, the only embeddings of p' or q' into a tree in $L(d')$ are strong embeddings. Thus it is enough to check whether $L_w(p') \cap L(d') \subseteq L_w(q')$.

Another case is that $R_p \cap R_q = \emptyset$. Then the containment holds exactly when $L_s(p) \cap L(d) = \emptyset$. Now we take p' and d' as previously, but this time we change the root label of q into $\top_{BAD}$. Again we check whether $L_w(p') \cap L(d') \subseteq L_w(q')$. Recalling that all embeddings are strong, this will be the case exactly when $L_s(p') \cap L(d') = \emptyset$ (as q' can never be embedded), that is when $L_s(p) \cap L(d) = \emptyset$.

The remaining case is that $R_p \cap R_q \neq \emptyset$ and $R_p \not\subseteq R_q$. This is possible only when the root of p is a wildcard (and the root of q is labelled by some letter). Let us suppose first that child edges are allowed in $\mathcal{F}_1$. Then to create p' we attach above p an additional $\top$-labelled root, using a child edge. To create q' we change the root of q into $\top_{OK}$ (without attaching anything). In d' the set of allowed roots is $\{\top\}$; we add rules $\top \to \top_{OK} + \top_{BAD}$, and $\top_{OK} \to \bigcup_{a \in R_p \cap R_q} L_a$, and $\top_{BAD} \to \bigcup_{a \in R_p \setminus R_q} L_a$, where $a \to L_a$ were the rules of d. Now to a tree $t \in L(d)$ corresponds a tree $new(t) \in L(d')$ obtained by replacing the root label to $\top_{OK}$ if it was in $R_p \cap R_q$ and to $\top_{BAD}$ otherwise, and attaching a new $\top$-labelled node above it. We see that every tree in $L(d')$ is of this form. We notice that p strongly embeds into a tree $t \in L(d)$ exactly when p' embeds into $new(t)$ (where every embedding is strong). But also q strongly embeds into a tree $t \in L(d)$ exactly when q' embeds into $new(t)$ (in particular, when the root label of q is from $R_p \setminus R_q$, then none of q and q' can be embedded). Thus the problem is equivalent to testing whether $L_w(p') \cap L(d') \subseteq L_w(q')$.

The only remaining possibility is that the root of p is a wildcard, and p does not use child edges. Then $L_w(p) = L_s(p)$. We take $p' = p$, and to create q' we change the root of q into $\top_{OK}$. In the new DTD d' the set of allowed roots is now $\{\top_{OK}, \top_{BAD}\}$, with the rules $\top_{OK} \to \bigcup_{a \in R_p \cap R_q} L_a$, and $\top_{BAD} \to \bigcup_{a \in R_p \setminus R_q} L_a$, where $a \to L_a$ were the rules of d. This time to a tree $t \in L(d)$ corresponds a tree $new(t) \in L(d')$ obtained only by replacing the root label to $\top_{OK}$ if it was in $R_p \cap R_q$ and to $\top_{BAD}$ otherwise, without attaching anything. We see that every tree in $L(d')$ is of this form, and that q strongly embeds into a tree $t \in L(d)$ exactly when q' embeds into $new(t)$. Thus we may test whether $L_w(p') \cap L(d') \subseteq L_w(q')$.

Without DTD the reduction is much simpler. If the root labels of p and q are different letters, or the first is a wildcard and the second is a letter, then surely $L_s(p) \not\subseteq L_s(q)$. Otherwise we have the same letters in both roots, or in q we have a wildcard, and we can proceed as in the case $R_p \subseteq R_q$. $\square$

B. PROOFS FOR SECTION 3: CONTAINMENT WITHOUT SCHEMA

B.1 Theorem 3.2: P Results

THEOREM 3.2. *The following problems are in P:*
(1) CONTAINMENT *of* $\mathsf{PQ}(/,//,*)$ *in* $\mathsf{TPQ}(/,//,*)$*;*
(2) CONTAINMENT *of* $\mathsf{TPQ}(//,*)$ *in* $\mathsf{TPQ}(/,//,*)$*;*
(3) CONTAINMENT *of* $\mathsf{TPQ}(/,//,*)$ *in* $\mathsf{TPQ}(//,*)$*;*
(4) W-CONTAINMENT *of* $\mathsf{TPQ}(/,*)$ *in* $\mathsf{TPQ}(/,//,*)$*.*

B.1.1 General Notions

Let *island* be a part of a pattern that is connected by child edges. An island q is a *parent* of an island q' if there are nodes v in q and v' in q' such that there is a descendant edge from q to q'. *Ancestor* and *descendant* relations on islands are defined as the transitive closure of parent relation and the inverse of ancestor relation, respectively. The *topmost island* of a pattern p is the island containing the root of p.

We say that a pattern is *normalized* if every leaf of every island either is a root of the island or is labelled by a letter (not by a wildcard).

Notice that every pattern can be turned into an equivalent one that is normalized. Indeed, consider a pattern q and take some its node v that is a leaf of its island, is labelled by a wildcard, and is connected with its parent u via a child edge. In this situation we could equally well consider the pattern q' obtained from q by changing the child edge between u and v into a descendant edge. Of course any embedding of q in a tree t is also a correct embedding of q' in this tree. Conversely, suppose that we have an embedding f of q' into a tree t. Then as long as $f(v)$ is not a child of $f(u)$, we can move $f(v)$ to its parent and we still have a correct embedding (recall that v is a leaf of an island, so it is connected with its children only via descendant edges). Thus we can assume that $f(v)$ is a child of $f(u)$, and with this assumption f is also a correct embedding of q into the tree.

For a pattern p and for $k \in \mathbb{N}$ we denote by $*^k(p)$ the pattern obtained by appending a path of k wildcard nodes above the root of p (using child edges).

Let us now recall the notion of *canonical trees* (canonical models) from [34]. Let p be a pattern, and $\perp$ a fresh letter (not appearing in any of patterns considered at the moment). A tree t is called a canonical tree of p if it is obtained from p by replacing each descendant edge by a sequence of child edges going through new $\perp$-labelled nodes (possibly zero of them), and by replacing each wildcard by the $\perp$ letter.

The main property of canonical models is that $L_w(p) \subseteq L_w(q)$ if and only if each canonical tree of p belongs to $L_w(q)$. We implicitly use this property in our proofs.

B.1.2 Proof of Theorem 3.2(1): CONTAINMENT *of* $\mathsf{PQ}(/,//,*)$ *in* $\mathsf{TPQ}(/,//,*)$ *is in P*

Notice that each canonical tree of $p \in \mathsf{PQ}(/,//,*)$ consists of a single path. Such a tree will be called a word. For a word t, by $cut^k(t)$ we denote the subtree of t rooted in the node at depth k (we cut off k topmost nodes). Recall that $|t|$ denotes the number of nodes of t (if t is a word, this is just its length). Our algorithm is based on the following lemma.

LEMMA B.1. *Let* $p = w//p' \in \mathsf{PQ}(/,//,*)$*, let* $q \in \mathsf{TPQ}(/,//,*)$ *be normalized, and let* q_{top} *be the topmost island of* q*. Suppose that* $L_w(w) \not\subseteq L_w(q_{top})$*. Then* $L_w(p) \subseteq L_w(q)$ *if and only if* $L_w(*^{|w|}(p')) \subseteq L_w(q)$*.*

Before proving this lemma, let us see how it allows us to construct an algorithm. We will use dynamic programming.[7] For each node u of p, each node x of q (in fact it is enough to consider nodes x that are roots of islands), and for each k such that the depth of $*^k(\mathsf{subquery}^p(u))$ is not greater than the depth of p, we will compute whether $L_w(*^k(\mathsf{subquery}^p(u))) \subseteq L_w(\mathsf{subquery}^q(x))$. Let us describe a single step of this algorithm (recall that each pattern can be turned into an equivalent normalized one).

LEMMA B.2. *Let* $p \in \mathsf{PQ}(/,//,*)$*, and let* $q \in \mathsf{TPQ}(/,//,*)$ *be normalized. Suppose that we know whether* $L_w(*^k(\mathsf{subquery}^p(u))) \subseteq L_w(\mathsf{subquery}^q(x))$ *for each node* u *of* p*, each node* x *of* q*, and for each* k *such that the depth of* $*^k(\mathsf{subquery}^p(u))$ *is not greater than the depth of* p*, if either* $*^k(\mathsf{subquery}^p(u))$ *or* $\mathsf{subquery}^q(x)$ *has less islands than* p *or* q*, respectively. Then we can compute in polynomial time whether* $L_w(p) \subseteq L_w(q)$*.*

PROOF. Let w be the topmost island of p, and q_{top} the topmost island of q. Let X be the set of roots of islands of q being a child of q_{top} (in other words, nodes of q that are outside q_{top}, but their parents are in q_{top}). For a node x, let $d(x)$ denote its depth. First observe that if $p = w$ then the situation is easy, since there is only one canonical tree of p, and we can check whether it belongs to $L_w(q)$ or not. Thus assume that $p = w//p'$. Consider the unique canonical tree t_w of w. We check whether q_{top} can be embedded into t_w.

Suppose first that q_{top} cannot be embedded into t_w. Then $L_w(w) \not\subseteq L_w(q_{top})$. By Lemma B.1, $L_w(p) \subseteq L_w(q)$ if and only if $L_w(*^{|w|}(p')) \subseteq L_w(q)$. The latter is known by assumption, as $*^{|w|}(p')$ has less islands than p.

Next, suppose that q_{top} can be embedded into t_w. We concentrate on the embedding which maps the root of q_{top} to a node of t_w of the smallest possible depth; denote this depth m. For each $x \in X$ by assumption we know whether

[7] We describe the algorithm using dynamic programming, but in fact it just performs a simple recursion (each partial result is used only once).

$L_w(cut^{m+d(x)}(p)) \subseteq L_w(\mathsf{subtree}^q(x))$. Notice that $m + d(x) \le |w|$ (the whole q_{top} can be embedded into t_w starting from the node at depth m, and x is just below some node of q_{top}), so the expression $cut^{m+d(x)}(p)$ makes sense. We have two cases.

Suppose first that $L_w(cut^{m+d(x)}(p)) \subseteq L_w(\mathsf{subtree}^q(x))$ for each $x \in X$. We claim that then $L_w(p) \subseteq L_w(q)$. Indeed, take a canonical tree t of p. Since t_w is a prefix of t, the island q_{top} can be embedded into t starting exactly at the node at depth m. Moreover, for each $x \in X$ we have $cut^{m+d(x)}(t) \in L_w(cut^{m+d(x)}(p)) \subseteq L_w(\mathsf{subtree}^q(x))$, which gives an embedding of $\mathsf{subtree}^q(x)$ into t starting at the node at depth $m + d(x)$ or lower. Together all these embeddings give an embedding of q into t.

Contrarily, suppose that $L_w(cut^{m+d(x)}(p)) \not\subseteq L_w(\mathsf{subtree}^q(x))$ for some $x \in X$. We claim that in this case $L_w(p) \not\subseteq L_w(q)$. Indeed, there exists a canonical tree t of p for which $cut^{m+d(x)}(t) \notin L_w(\mathsf{subtree}^q(x))$. Since q_{top} cannot be embedded anywhere higher than starting from depth m, the whole q cannot be embedded into t. □

For the proof of Lemma B.1 we need the following auxiliary lemma.

LEMMA B.3. *Let $q \in \mathsf{TPQ}(/, //, *)$ be normalized. Let t and t' be two words of the same length. Suppose that for each node x whose labels in t and t' differ, labels of all ancestors of x in t (but not necessarily in t') do not appear in q. Suppose also that $cut^k(t) \in L_s(q)$ and $cut^m(t') \in L_s(q)$. Then as well $cut^{\max(k,m)}(t) \in L_s(q)$.*

PROOF. First let us see that the lemma holds when q is a single island. W.l.o.g. we can remove the top part of t and assume that $\min(k, m) = 0$. Moreover, for $k \ge m$ the lemma is trivial. Thus suppose that $k = 0 < m$. Let d be the greatest depth at which labels of t and t' differ. Since $t \in L_s(q)$, and since in t we do not have letters appearing in q at depths smaller than d, also in q we do not have letters at depths smaller than d (we have there only wildcards). But then in the strong embedding of q into $cut^m(t')$ only wildcards can be mapped to nodes having different labels in t' than in t. Thus labels of these nodes do not matter; we can strongly embed q also into $cut^m(t)$.

Now consider the general situation (many islands). We have two embeddings of q: f into t and g into t'. In the new embedding h we map each node x of q into the lower of the nodes $f(x)$ and $g(x)$. By the single-island case, this gives a correct embedding into t of each island separately. But obviously the descendant edges are also preserved (notice that $k < k'$ and $m < m'$ implies $\max(k, m) < \max(k', m')$ for any numbers k, k', m, m'). □

PROOF OF LEMMA B.1. We always have $L_w(p) \subseteq L_w(*^{|w|}(p'))$; the nontrivial implication is that $L_w(p) \subseteq L_w(q)$ implies $L_w(*^{|w|}(p')) \subseteq L_w(q)$. Thus suppose that $L_w(p) \subseteq L_w(q)$, and take a canonical tree t of $*^{|w|}(p')$. Our aim is to prove that q embeds into t. Now we will consider several trees, and we will analyze how p and q embed into them. This will lead to the statement that indeed q embeds into t.

Let t_w be the canonical tree of w. We construct t_1 by taking t_w, appending a long path of $\bot$-labelled nodes (of some length n greater than the depth of q), and finally appending t. We have $t_1 \in L_w(p)$, so by assumption $t_1 \in L_w(q)$. Recall that $L_w(w) \not\subseteq L_w(q_{top})$; in particular q_{top} cannot consist of a single wildcard node. Thus by normalization all leaves of q_{top} are letters, so in any embedding of q_{top} into t_1 they are mapped either inside t_w, or inside t. In the former case in fact the whole q_{top} would be mapped into t_w which would contradict with the assumption $L_w(w) \not\subseteq L_w(q_{top})$, so these leaves are indeed mapped inside t. Denote $t_2 = cut^{|w|}(t_1)$; notice that it starts by a path of n $\bot$-labelled nodes, below which we have the tree t. Because of the long distance before t, the embedding of q into t_1 does not use at all the nodes of t_w, so in fact $t_2 \in L_w(q)$.

Next, we create t_3: we take in t_2 the first $|w|$ nodes after the initial fragment of n $\bot$-labelled nodes (that is the first $|w|$ nodes of t), and we change them into t_w. Recall that this changed nodes were initially $\bot$-labelled, since t was a canonical tree of $*^{|w|}(p')$. We notice that $cut^n(t_3) \in L_w(p) \subseteq L_w(q)$. We are going to use Lemma B.3 for t_2 (as t) and t_3 (as t'). We have $cut^k(t_2) \in L_s(q)$ for some $k \ge 0$, and $cut^m(t_3) \in L_s(q)$ for some $m \ge n$, and we obtain $cut^{\max(k,m)}(t_2) \in L_s(q)$, so $cut^n(t_2) \in L_w(q)$. Recalling that $cut^n(t_2) = t$, we are done. □

B.1.3 Proof of Theorem 3.2(2): CONTAINMENT *of* $\mathsf{TPQ}(//, *)$ *in* $\mathsf{TPQ}(/, //, *)$ *is in P*

PROOF. First, suppose that in some island of $q \in \mathsf{TPQ}(/, //, *)$ we have two non-wildcard nodes that are either on different depths or are labelled by different letters. Then surely the inclusion does not hold. Indeed, consider the canonical tree t of $p \in \mathsf{TPQ}(//, *)$ in which all descendant edges are instantiated as a long chain of special letters, longer than the size of q. Then all the non-special letters in t are further from each other than the size of q. Therefore if in some two nodes of the same island of q there are letters, they have to be mapped to the same node of t. This is impossible if these two nodes are on different depth or are labelled by different letters.

From now on we assume that q is normalized, and that in each island of q all non-wildcard nodes appear at the same depth and are labelled by the same letter. To fix some terminology, let us call such pattern *singular*. Our algorithm will perform dynamic programming: for each node u of p, each node x of q (it is enough to consider nodes x that are roots of islands), and for each k such that the depth of $*^k(\mathsf{subquery}^p(u))$ is not greater than the depth of p, we will compute whether $L_w(*^k(\mathsf{subquery}^p(u))) \subseteq L_w(\mathsf{subquery}^q(x))$. For u, x being roots of the patterns and for $k = 0$ this solve the target problem. Thus it remains to prove the following claim (where as p we take any pattern of the form $*^k(\mathsf{subquery}^p(u))$ of depth not greater than the depth of p, and as q we take any subtree of q).

CLAIM B.4. *Let $p \in \mathsf{TPQ}(//, *)$, and let $q \in \mathsf{TPQ}(/, //, *)$ be singular. Let P' contain all queries of the form $*^k(\mathsf{subquery}^p(u))$ which are of depth not greater than the depth of p (where u iterates over all nodes of p). Suppose*

that for each $p' \in P'$ and for each proper subquery q' of q we know whether $L_w(p') \subseteq L_w(q')$. Then we can compute in polynomial time whether $L_w(p) \subseteq L_w(q)$.

Proof of Claim B.4. Suppose first that the topmost island of q consists of a single wildcard-labelled node. Let R denote the set of all subqueries of p rooted just below the its root. We will observe the following property:

(♠) $L_w(p) \subseteq L_w(q)$ if and only if for each child x of $\mathsf{root}(q)$ there exists a query $p' \in R$ such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$.

It is easy to check whether the right side of this property holds using the assumptions of the claim (notice that $R \subseteq P'$), thus it remains to see that this property indeed holds.

For the right-to-left implication of (♠), suppose that for each child x of $\mathsf{root}(q)$ there exists a query $p' \in R$ such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$. Take a tree $t \in L_w(p)$; by definition we have an embedding f of p into t. Let x be a child of $\mathsf{root}(q)$, and let $p' \in R$ be such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$. Then f maps p' (a proper subquery of p) into a proper subtree of t. Since $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$, the pattern $\mathsf{subquery}^q(x)$ can be also embedded into this subtree. By taking such embeddings for all children x of $\mathsf{root}(q)$, and by mapping $\mathsf{root}(q)$ into $\mathsf{root}(t)$, we obtain an embedding of q into t.

Conversely, suppose that there is a child x of $\mathsf{root}(q)$ such that for each $p' \in R$ it holds $L_w(p') \not\subseteq L_w(\mathsf{subquery}^q(x))$. For each $p' \in R$, let $t_{p'}$ be a tree from $L_w(p') \setminus L_w(\mathsf{subquery}^q(x))$. If the label of $\mathsf{root}(p)$ is not a wildcard, we denote it as a; otherwise let a be an arbitrary letter. We construct t by taking a node labelled by a, and attaching below it trees $t_{p'}$ for all $p' \in R$. Notice that $t \in L_w(p)$. On the other hand, if there is an embedding of q into t, then its part is an embedding of $\mathsf{subquery}^q(x)$ into some $t_{p'}$, which does not exist by assumption. Thus t witnesses that $L_w(p) \not\subseteq L_w(q)$.

Next, consider the more interesting case, when the the topmost island of q contains some non-wildcard nodes. By singularity they are all on the same depth n and all labelled by the same letter a. Normalization implies that each leaf of this island is labelled by a; in particular all leaves are on depth n. Recall that the depth of a node was defined so that the root of a tree is at depth 0. Let S be the set of those nodes u of p that

- have label a,
- are on depth at least n, and
- such that no ancestor of u at depth at least n has label a.

We have a property similar to (♠), but now instead of one set R we have more sets. Namely, for each $u \in S$, and each $i \in \{1, \ldots, n+1\}$, let $R_i(u)$ contain all subqueries of $*^{n+1-i}(\mathsf{subquery}^p(u))$ rooted just below its root. Note that for all $i < n+1$ the set $R_i(u)$ contains just one tree, namely $*^{n-i}(\mathsf{subquery}^p(u))$. Let also X be the set of those nodes of q which are not in the topmost island, but whose parents are already in the topmost island (in other words, in X we have roots of all islands that are children of the topmost island). Additionally, by $d(x)$ we denote the depth of a node x. For $x \in X$ we see that $d(x) \in \{1, \ldots, n+1\}$. We will observe the following property:

(♣) $L_w(p) \subseteq L_w(q)$ if and only if there exists $u \in S$ such that for each $x \in X$ there exists a query $p' \in R_{d(x)}(u)$ such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$.

As previously, it is easy to check whether the right side of this property holds using the assumptions of the claim (notice that $R_i(u) \subseteq P'$ for each $i \in \{1, \ldots, n+1\}$ and each $u \in S$), thus it remains to see that this property indeed holds.

For the right-to-left implication of (♣), fix a node $u \in S$ such that for each $x \in X$ there exists a query $p' \in R_{d(x)}(u)$ such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$. Take a tree $t \in L_w(p)$; by definition we have an embedding f of p into t. Our goal is to construct an embedding g of q into t. All a-labelled leaves of the topmost island of q will be mapped by g to $f(u)$, and the ancestors of these leaves will be mapped to appropriate ancestors of $f(u)$. Recall that u, as an element of S, has label a and is on depth at least n, hence $f(u)$ as well, thus such embedding is correct. Next, for each $x \in X$ we should define g on the subtree rooted at x. Take some $x \in X$. The assumed right side of (♣) gives us a query $p' \in R_{d(x)}(u)$ such that $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$. A part of f gives us an embedding of $\mathsf{subquery}^p(u)$ into t. But since $f(u)$ is at depth at least n (and $d(x) \geq 1$), we can as well embed the whole pattern $*^{n+1-d(x)}(\mathsf{subquery}^p(u))$ into t, by mapping the new wildcard nodes to appropriate ancestors of $f(u)$. In particular p', as a subquery of $*^{n+1-d(x)}(\mathsf{subquery}^p(u))$, is embedded into t, with its root mapped to a descendant of the $(n+1-d(x))$-ancestor[8] of $f(u)$. Since $L_w(p') \subseteq L_w(\mathsf{subquery}^q(x))$, also $\mathsf{subquery}^q(x)$ can be embedded into this subtree of t; this embedding is taken as a part of g. Notice that the descendant relation between the parent of x and x is preserved, since the parent of x is mapped to the $(n+1-d(x))$-ancestor of $f(u)$.

Conversely, suppose that for each $u \in S$ we can find an $x \in X$ such that for each $p' \in R_{d(x)}(u)$ it holds $L_w(p') \not\subseteq L_w(\mathsf{subquery}^q(x))$. First, for each $u \in S$ we will construct a tree $t_u \in L_w(\mathsf{subquery}^p(u)) \setminus L_w(q)$ such that none of its nodes at depth at most $n-1$ has label a. Fix some $u \in S$, and take the node $x \in X$ such that for each $p' \in R_{d(x)}(u)$ it holds $L_w(p') \not\subseteq L_w(\mathsf{subquery}^q(x))$. Let b be some letter other than a.

If $d(x) = n+1$, we construct t_u as follows: for each $p' \in R_{d(x)}(u)$ we take a tree from $L_w(p') \setminus L_w(\mathsf{subquery}^q(x))$; above these trees we attach a node labelled by a, and above it we attach a path consisting of n nodes labelled by b. We observe that, in this case, $\mathsf{subquery}^p(u)$ can be embedded into t_u: the node u is mapped to the attached a-labelled

[8]As *k-ancestor* of a node u we denote the ancestor of u being at depth smaller by k.

node, and the subqueries rooted in the children of u, which are exactly the elements of $R_{d(x)}$, are mapped to the trees placed below this a-labelled node. On the other hand, any embedding of q into t_u would give an embedding of $\mathsf{subquery}^q(x)$ into some of the trees placed below our new a-labelled node, since this is the only node at depth n, and x is at depth $n+1$. By assumption such embedding does not exist, so $t_u \notin L_w(q)$.

Suppose now that $d(x) \leq n$. Now in $R_{d(x)}$ we have exactly one element, which is $p' = *^{n-d(x)}(\mathsf{subquery}^p(u))$. Take a tree $t' \in L_w(p') \setminus L_w(\mathsf{subquery}^q(x))$. We can assume that t' is a canonical tree of $L_w(p')$; thus it starts by a path of $n - d(x)$ letters other than a, under which we have an a-labelled node. We construct t_u by attaching a path of $d(x)$ nodes labelled by b above t'. Of course $\mathsf{subquery}^p(u)$ can be embedded into t_u. On the other hand, any embedding of q into t_u would give an embedding of $\mathsf{subquery}^q(x)$ into t', since t' is the only subtree which starts at depth $d(x)$. By assumption such embedding does not exist, so $t_u \notin L_w(q)$.

Till now for each $u \in S$ we have constructed a tree $t_u \in L_w(\mathsf{subquery}^p(u)) \setminus L_w(q)$ such that none of its nodes at depth at most $n-1$ has label a. Out of them we want to construct a tree $t \in L_w(p) \setminus L_w(q)$. We start from the tree of p (that is in p we treat all descendant edges as normal edges), and for each $u \in S$ we replace the subtree rooted there by t_u. Recall that no node in S is an ancestor of another node in S. It is clear that p can be embedded into t, since $\mathsf{subquery}^p(u)$ can be embedded into t_u for each $u \in S$. On the other hand, suppose that q was embedded into t. Notice that an a-labelled node from topmost island of q, being on depth n, could not be mapped to any node of t coming from the tree of p, as then this node (or some its ancestor) would be taken to S. Thus this a was mapped inside some t_u. But since at depths smaller than n in t_u we do not have any a, in fact the whole q was embedded inside t_u, which contradicts with our assumption on t_u. □

B.1.4 Proof of Theorem 3.2(3): CONTAINMENT *of* $\mathsf{TPQ}(/, //, *)$ *in* $\mathsf{TPQ}(//, *)$ *is in P*

PROOF. Let t_{min} be the smallest canonical tree of $p \in \mathsf{TPQ}(/, //, *)$. We will show that if a pattern $q \in \mathsf{TPQ}(//, *)$ embeds in t_{min} then it embeds in every canonical tree t of p. It is enough to finish the proof, as checking whether a pattern embeds into a given tree can be done in polynomial time.

Let f be an embedding of q into t_{min}. Let t be some canonical tree of p. For every node n in t_{min} there is a corresponding node in p, call it n', and for every node n' in p there is a corresponding node in t, call it n''. Let us define a function *corr* from nodes of t_{min} to nodes of t assigning for every node n the corresponding node n''. We construct embedding g of q into t as $g(n) = corr(f(n))$. Observe that if a node n_1 is an ancestor of a node n_2 in t_{min} then also $corr(n_1)$ is an ancestor of $corr(n_2)$ in t. Therefore g is a correct embedding of q into t, which finishes the proof. □

B.1.5 Proof of Theorem 3.2(4): W-CONTAINMENT *of* $\mathsf{TPQ}(/, *)$ *in* $\mathsf{TPQ}(/, //, *)$ *is in P*

PROOF. Even for weak containment, we only need to check whether $q \in \mathsf{TPQ}(/, //, *)$ (weakly) embeds in the unique canonical tree of $p \in \mathsf{TPQ}(/, *)$ This can be done in polynomial time since testing whether a tree t is in $L_w(q)$ is in P. □

B.2 Theorem 3.3: coNP-completeness

THEOREM 3.3. *The following problems are coNP-complete:*
(1) CONTAINMENT *of* $\mathsf{TPQ}(/, //, *)$ *in* $\mathsf{TPQ}(/, //, *)$ *[34];*
(2) W-CONTAINMENT *of* $\mathsf{TPQ}(/, //)$ *in* $\mathsf{TPQ}(/, *)$.

PROOF. Item (1) was already proved in [34].

We now prove (2). Membership in coNP is immediate from case (1). The coNP-hardness proof is by an adaptation of the proof of Theorem 4 in [34]. Theorem 4 in that paper claims that S-CONTAINMENT of $p_1 \in \mathsf{TPQ}(/, //)$ in $p_2 \in \mathsf{TPQ}(/, //, *)$ is coNP-complete. So we only need to show that the construction can be adapted by, at the same time, going from strong containment to weak containment and removing all descendant edges from the pattern p_2.

The pattern p_2 in the proof in [34] contains descendant edges at two locations:

- at the root and
- in the gadgets $F(y_i)$ of tree patterns C_i.

The descendant edge at the root can disappear since we consider weak containment instead of strong containment. The main trick in the present proof is to show how we can construct alternative gadgets for the Y_i, $T(y_i)$, and $F(y_i)$ in the proof of Miklau and Suciu, such that their construction still works but we do not use any descendant edge in $T(y_i)$ and $F(y_i)$. Concretely, these gadgets look as depicted in Figure 5.

To see how these gadgets work in the construction of Miklau and Suciu, observe the following facts. The gadget Y_i is responsible for generating trees that represent the truth values true and false for a propositional variable x_i, just as in the proof of Miklau and Suciu. The tree that represents true is in part (d) of Figure 5 and the trees that represent false are in part (e). We assume that there is at least one node on the dotted path in the trees in Figure 5(e).

Hence, each tree that represents true strongly matches $T(y_i)$ but does not strongly match $F(y_i)$ and each tree that represents false strongly matches $F(y_i)$ but does not strongly match $T(y_i)$.

To conclude the proof, we add that the construction in Lemma 3 of [34] does not add any wildcards to the patterns on the right, and only adds the single descendant edge at the root (which does not matter to us because we consider weak containment). Formally, this means that we perform the same construction as in Lemma 3 of [34] but we do not add the extra root nodes r and its incident edge. So we have a reduction from the complement of SAT in which $p_1 \in \mathsf{TPQ}(/, //)$ and $p_2 \in \mathsf{TPQ}(/, *)$, which concludes the hardness proof. □

Y_i: $T(y_i)$: $F(y_i)$: "$x_i = \mathsf{true}$": "$x_i = \mathsf{false}$":

(a) (b) (c) (d) (e)

Figure 5: Gadgets for the proof of Theorem 3.3.

C. PROOFS FOR SECTION 4: SATISFIABILITY WITH SCHEMA

THEOREM 4.1(2). *The following problems are in P:*
(2) SATISFIABILITY *of* $\mathsf{TPQ}(//,*)$ *w.r.t. a fixed DTD.*

PROOF. Case (2) can be shown by an adaptation of the result from [16]. Theorem 5 there states, in our terminology, that SATISFIABILITY of $\mathsf{TPQ}(//,*)$ w.r.t. a fixed DTD is in P for the injective semantics. The proof can be modified in such a way that it also works for the non injective semantics. One has to slightly modify the Definition 13 and the proof of Lemma 14, no hard problems occur on that way. □

THEOREM 4.2(2). *The following problems are NP-complete:*
(2) SATISFIABILITY *of* $\mathsf{TPQ}(/)$ *w.r.t. a fixed DTD.*

PROOF. The problems belongs to NP, since this is a less general than case (3).

For NP-hardness of (2) we continue the proof from Section 4. First, we reduce the 3-PARTITION problem to the 4-PARTITION problem. Consider an instance B, S of the 3-PARTITION problem. Out of it we create an instance of the 4-PARTITION problem as follows. As K we take the smallest number such that the sum of all the numbers in S is smaller than 2^{K-2}. Let also L be the smallest number such that $4 \cdot \frac{|S|}{3} \leq 4 \cdot 2^L$. To the multiset S' we take all the numbers from S, and $\frac{|S|}{3}$ times the number $2^K - B$, and $4 \cdot 2^L - 4 \cdot \frac{|S|}{3}$ times the number 2^{K-2}. Notice that the new instance is of polynomial size. Of course when the original instance B, S has a solution, then the instance 2^K, S' as well: to each triple we append the number $2^K - B$, and we group the numbers 2^{K-2} by four. Suppose that the instance 2^K, S' of the new problem has a solution. In this solution, the number 2^{K-2} cannot be grouped with $2^K - B$, because $2^{K-2} + 2^K - B > 2^K$ (we have $B < 2^{K-2}$). A group consisting of some numbers from S, and a number 2^{K-2} (one or two or three times) cannot exist, because the sum of numbers from S is smaller than 2^{K-2}. Thus the numbers 2^{K-2} are grouped by four. Each other group consists of some numbers from S, and the number $2^K - B$ taken exactly once. Moreover in each group we have exactly 3 numbers from S: they sum to B, and each of them is strictly between $\frac{B}{4}$ and $\frac{B}{2}$. These triples give a solution to the instance B, S of the original problem.

In Section 4 we have already created a pattern p out of an instance 2^K, S' of the 4-PARTITION problem. It remains to prove that $L(d) \cap L_s(p) \neq \emptyset$ if and only if the instance of 4-PARTITION has a solution. Assume that there exists a tree t in $L(d) \cap L_s(p)$; fix some strong embedding of p into this tree. Notice that the number of trees from T_M used in p is 2^{K+L} (which is the sum of all numbers in S'), and they are all attached on depth $K + L$. Thus in t at depth $K + L$ we have 2^{K+L} nodes (which is the maximum), to each of them we exactly one node from p is mapped. Nodes at depth L give us a partition of S' to submultisets. Each node defines one submultiset: we look at the nodes of p which are mapped to this node; they correspond to some elements of S'; we take them to the submultiset. This is indeed a partition (each element is in exactly one submultiset). Moreover the sum of elements in each submultiset is exactly 2^K: these is exactly the number nodes at depth 2^{K+L} attached below a node at depth 2^L.

Similarly, when we have a solution of our instance, we can easily create a tree t in $L(d) \cap L_s(p)$.

Let us also observe that the same reduction works for W-SATISFIABILITY, that is $L(d) \cap L_w(p) \neq \emptyset$ if and only if the instance of 4-PARTITION has a solution. Indeed, if we have a tree into which p weakly embeds, then we can restrict it to the subtree into which p strongly embeds, and obtain a tree from $L(d) \cap L_s(p)$. □

D. PROOFS FOR SECTION 5: VALIDITY WITH SCHEMA

THEOREM 5.1(5). *The following problems are in P:*

(5) VALIDITY *of TPQ(/, //, ∗) w.r.t. a fixed DTD.*

PROOF. The proof is rather easy. Consider a fixed DTD d with an alphabet Σ of size n and a pattern p. It can be easily checked in polynomial time whether $L(d)$ is empty, then clearly d is valid. From now on we assume $L(d)$ to be nonempty. Observe that if there is a path in p which has length at least n then there are at least $n+1$ nodes on that path and by pigeonhole principle some two of them are labelled equally, say by a letter a. However, there clearly exists a tree in $L(d)$ which has at most one letter a on every path. Otherwise it would mean that node labelled by a enforces a descendant labelled by a, which implies that $L(d)$ is empty. Therefore if there is any path in p of length at least n then p is not valid. Now observe that for fixed k there is only fixed number of patterns which are labelled by a fixed alphabet Σ of size n, have depth exactly k and none of their sibling subtrees are equal. Indeed, while denoting the number of such a trees by $N(k)$, we have $N(0) = n$, $N(i) = n(2^{N(i-1)} - 1)(2^{N(0)+N(1)+\ldots+N(i-2)})$. Then our polynomial algorithm works like that: it precomputes in a fixed time answers for all the trees up to depth n which do not have equal sibling subtrees. Then if p has depth n or more it is not valid; otherwise we clean up p by removing its equal sibling subtrees and check the precomputed answer for the cleaned pattern p. □

E. PROOFS FOR SECTION 6: CONTAINMENT WITH SCHEMA

E.1 Theorem 6.1: P Results

OBSERVATION 6.2: *Let Σ be a finite alphabet and let q be a query in (1) PQ(/, //); (2) TPQ(//, ∗); or in (3) TPQ(/, ∗). Then we can construct in polynomial time a non-deterministic tree automaton for the language $\mathcal{T}_\Sigma \setminus L_s(q)$.*

PROOF. For the purpose of case (1) we first argue that there exists a polynomial size DFA A accepting the language of words belonging to the $L_s(q)$. That is shown in Theorem 4.1 in [26]. Then the proof is finished by the following lemma, which is a folklore:

LEMMA E.1. *For every DFA A on words there exists a nondeterministic tree automaton B of polynomial size w.r.t. the size of A which accepts all the trees, which have no path accepted by A.*

Case (2) is based on the following observation. Suppose that we have a tree $t = a(t_1, \ldots, t_k)$, and a tree pattern query $q = b(//q_1, \ldots, //q_m)$. When $a = b$ or $b = *$, then $t \in L_s(q)$ if and only if $t \in L_w(q)$ if and only if for each j there exists i such that $t_i \in L_w(q_j)$. When $a \neq b \neq *$, then $t \notin L_s(q)$ and $t \in L_w(q)$ if and only if $t_i \in L_w(q)$ for some i. We want to negate it. When $a = b$ or $b = *$, then $t \notin L_s(q)$ (or $L_w(q)$) if and only if there exists j such that for each i it holds $t_i \notin L(q_j)$. When $a \neq b \neq *$, then $t \notin L_w(q)$ if and only if $t_i \notin L_w(q)$ for each i. This characterization allows us to construct an automaton of polynomial size. Its states are nodes of the tree pattern q plus a few auxiliary ones (when the automaton is in a state x in some node y, this means that the subtree rooted at y does not belong to the language defined by the subpattern rooted at x). The transitions can be described polynomially as well, because we always pass the same state to all children.

The proof of case (3) is pretty similar to the proof of case (2). We similarly assume $t = a(t_1, \ldots, t_k)$, and a tree pattern query $q = b(/q_1, \ldots, /q_m)$. In the case $a = b$ or $b = *$ it works the same; $t \in L_s(q)$ if and only if for each j there exists i such that $t_i \in L_s(q_j)$. In case of $a \neq b$ always $t \notin L(q)$, in contrast to case (2). Based on that observation we build an automaton for $\mathcal{T}_\Sigma \setminus L_s(q)$ similarly as in case (2). □

THEOREM 6.1: *The following are in P:*
(1) CONTAINMENT *of PQ(/, //, ∗) in PQ(/, //);*
(2) CONTAINMENT *of PQ(/, //, ∗) in TPQ(//, ∗);*
(3) S-CONTAINMENT *of PQ(/, //, ∗) in TPQ(/, ∗);*
(4) W-CONTAINMENT *of PQ(/, //, ∗) in TPQ(/);*
all w.r.t. a DTD.

Proof of Theorem 6.1(1)-(3)

As already explained in Section 6, the proof follows from Observation 6.2.

Proof of Theorem 6.1(4): W-CONTAINMENT ***of*** PQ(/, //, ∗) ***in*** TPQ(/) ***w.r.t. a DTD is in P***

Let us first observe that the automaton recognizing $\mathcal{T}_\Sigma \setminus L_w(q)$ may be of exponential size, thus we have to proceed differently from cases (1)–(3). Indeed, take for example the pattern q from Figure 6. Then, for all sets $B, C \subseteq \{0, \ldots, n-1\}$ we can consider a tree $T(B, C)$ consisting of a path of $2n$ a-labelled nodes, where additionally for each $i \in B$ the node at depth i has a b-labelled child, and for each $i \in \{0, \ldots, n-1\} \setminus C$ the node at depth $n+i$ has a c-labelled child. Notice that $T(B, B)$ for each set B belongs to $\mathcal{T}_\Sigma \setminus L_w(q)$. But suppose that for two sets B_1, B_2 of the same size the a-labelled nodes at depth n of $T(B_1, B_1)$ and $T(B_2, B_2)$ are labelled by the same state in some runs of the NFA recognizing $\mathcal{T}_\Sigma \setminus L_w(q)$. Then this NFA would also accept $T(B_1, B_2)$ which does not belong to $\mathcal{T}_\Sigma \setminus L_w(q)$. Thus the number of states required to recognize $\mathcal{T}_\Sigma \setminus L_w(q)$ cannot be smaller than the number of subsets of $\{0, \ldots, n-1\}$ of size, say, $\lfloor \frac{n}{2} \rfloor$, that is exponential in n.

Before starting the proof, let us explain its overall structure. Let $p \in$ PQ(/, //, ∗), $q \in$ TPQ(/), and let d be a DTD. In a first stage, whenever in q we have two siblings with the same label, we merge them. Although this changes $L_w(q)$, it is easy to see that it does not influence the containment question. In a second stage, we remove

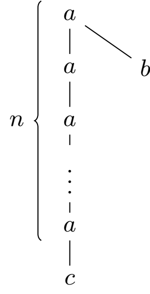


Figure 6: Pattern q for the proof of Theorem 6.1(4).

redundant subqueries of q. Namely, it may happen that we can remove some subquery of q obtaining some q', so that whenever q' can be embedded into a tree from $L(d)$, then the whole q can be embedded as well. After this cleaning stage, we have one of two mutually exclusive situations. One possibility is that the pattern q is a path, or is very similar to a path (namely, we consider a shape of patterns which we call a *brush*). Then the complement of $L_w(q)$ can be recognized by a finite automaton of polynomial size. We can construct this automaton, intersect it with an automaton for $L_w(p) \cap L(d)$, and check for emptiness, as in cases (1)-(3). The opposite case is that in q we have some branching. Then we can prove that the containment never holds. Indeed, to obtain a tree t from $L_w(p) \cap L(d)$ we are quite restricted only while arranging the path into which p will be embedded; but into this path at most one path of q embeds. Outside of this path in t we can place arbitrary subtrees (satisfying d), so we can place there subtrees into which the rest of q cannot be embedded. Such subtrees exist, since otherwise the rest of q would be redundant, so it should be removed during the second stage.

In fact the most difficult part is to check whether q is (weakly) equivalent to its part q' w.r.t. our DTD d. At first glance this looks hopeless, as this is just the W-Containment problem of $\mathsf{TPQ}(/)$ in $\mathsf{TPQ}(/)$ w.r.t. a DTD, which by Theorem 6.3 is in general coNP-hard. Hopefully, our pattern q is not arbitrary: there are no two siblings labelled the same (thanks to the first stage). The crux of the proof is to reduce the equivalence problem in this special case to multiple smaller instances of the (slightly generalized) containment question of $\mathsf{PQ}(/)$ in $\mathsf{TPQ}(/)$ w.r.t. a DTD. Thanks to that, we can proceed by dynamic programming.

In order to perform the dynamic programming, we need to slightly generalize the containment problem. In this generalization, beside of patterns $p \in \mathsf{PQ}(/, //, *)$, $q \in \mathsf{TPQ}(/)$ and a DTD d, we also have two sets F ("forbidden") and R ("required") of letters, describing the situation below the leaf of p. We define a language $L_s(p, F, R)$ containing these trees t into which p can be embedded so that no child of the image of the leaf of p is labelled by a letter from F, and for each letter from R there is a child of the image of the leaf of p labelled by this letter. We also define a language $L_>(q)$ containing trees t such that q can be embedded into a proper subtree of t. In the generalized problem we ask whether $L_s(p, F, R) \cap L(d) \subseteq L_>(q)$.

Notice that the W-Containment problem can be easily reduced to this problem. Indeed, we create a d' having all rules of d and the additional rule $r \to S_d$, where r is a fresh letter and S_d is the set of root labels allowed by d. We also take $p'_1 = r//p$. Then $L(p) \cap L(d) \subseteq L(q)$ if and only if $L_s(p', \emptyset, \emptyset) \cap L(d') \subseteq L_>(q)$: trees in $L(p) \cap L(d)$ are in one-to-one correspondence with trees in $L_s(p', \emptyset, \emptyset) \cap L(d')$ by appending an additional r-labelled root, and q embeds into an original tree exactly when it embeds into a proper subtree of the tree with an additional root. We observe that d' does not contain any requirement on the root label (only the pattern p' ensures that the root label is r). Thus in the sequel we only consider DTDs having this property.

As already mentioned, at the beginning we simplify the pattern q and the DTD d so that no node will have two children with the same label. This is possible because p describes a single path. First, we simplify the DTD. For a regular language L, by $rd(L)$ (rd stands for "remove duplicated") we denote the language of those words w such that each letter appears in w at most once and there exists a word $v \in L$ for which the set of letters appearing in w is the same as the set of letters appearing in v. In other words, $rd(L)$ contains sets of letters appearing in words from L, where a set is written in any order but each letter at most once. For a DTD d, by $rd(d)$ we denote the DTD which for each rule $a \to L$ of d has a rule $a \to rd(L)$.

Lemma E.2. *Let $p \in \mathsf{PQ}(/, //, *)$, let F and R be sets of labels, let d be a DTD, and let $q \in \mathsf{TPQ}(/)$. Then $L_s(p, F, R) \cap L(d) \subseteq L_>(q)$ if and only if $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$.*

Proof. The proof is rather easy. Suppose that $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$, and take a tree $t \in L_s(p, F, R) \cap L(d)$. Fix some embedding f witnessing that $t \in L_s(p, F, R)$. To obtain $t' \in L(rd(d))$, if a node has many children with the same label we remove all of them but one, ensuring that the image of f (that is a path) is not removed. The path p can be still embedded in the same way, and the set of labels in the children of the image of its leaf is as before, so $t' \in L_s(p, F, R)$. Then, by assumption $t' \in L_>(q)$. Since we were only removing nodes, q can be embedded in the same place in t.

The opposite direction is similar, but this time we have to duplicate subtrees: when a node according to d requires multiple children with some label, we duplicate the subtree rooted in the child with that label. If q can be embedded after such duplication, it could be embedded also before the duplication. □

It is not clear how the size of $rd(d)$ is related to the size of d, possibly it is exponential. Thus we use $rd(d)$ only in the argumentation, but all actual calculations are done using the original DTD d.

Next, we simplify the pattern $q \in \mathsf{TPQ}(/)$. Suppose that some its node has two children with the same label. Then we glue these children together (we remove the subtrees rooted in these children, and we create a new child having as subtrees all subtrees of the two removed children). We repeat this operation as long as it is possible; the obtained pattern is denoted $rd(q)$. It has the property that none of its nodes has a sibling with the same label.

Lemma E.3. *Let $p \in \mathsf{PQ}(/, //, *)$, let F and R be sets of labels, let d be a DTD, and let $q \in \mathsf{TPQ}(/)$. Then $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$ if and only if $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(rd(q))$.*

Proof. Notice that each node of a tree t satisfying $rd(d)$ has at most one child with each label, so q is always embedded into t in such a way that children with the same label are mapped to the same node. Thus for such trees q and $rd(q)$ are equivalent. □

Unlike for DTDs, we can easily compute $rd(q)$ in polynomial time, so in the sequel we can assume that in q no node has two children with the same label.

Before describing how our dynamic-programming algorithm works, we say how to solve the base case of *brush* patterns. A pattern in $\mathsf{TPQ}(/)$ is called a brush if each its node either has only one child, or is a leaf, or all its children are leaves. In other words, a brush is an arbitrary long path ended by a node having arbitrarily many children being leaves.

Lemma E.4. *Let $p \in \mathsf{PQ}(/, //, *)$, let F and R be sets of labels of size at most one, let $d \in DTD$, and let $q \in \mathsf{TPQ}(/)$ be such that none of its nodes has a sibling with the same label. If q is a brush, we can decide in polynomial time whether $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$.*

Proof. The shape of q allows us to construct an automaton of polynomial size recognizing the complement of $L(q)$. Thus we can construct an automaton recognizing trees in $L_s(p, F, R) \cap L(d) \setminus L_>(q)$, and test for its emptiness (thanks to Lemma E.2 it does not matter whether we use d or $rd(d)$). □

Next, for $q \in \mathsf{TPQ}(/)$ we define a set $X(q)$ of some „subpatterns" of q. These will be the patterns to which we descend from q in a single step of our dynamic programming. For each node u of q such that each node of q other than u is an ancestor or a descendant of u,

- to $X(q)$ we take the pattern obtained from q by removing all nodes except u, ancestors of u, and children of u, and
- for each child v of u to $X(q)$ we take the pattern obtained from q by removing all nodes except v, ancestors of v, and descendants of v,
- but we do not take q itself to this set (even if it is obtained by the above rules).

We also define the transitive closure of the X operation: let $X^+(q)$ be the smallest set such that $X(q) \subseteq X^+(q)$ and $X(q') \subseteq X^+(q)$ for each $q' \in X^+(q)$. Notice that there are only polynomially many patterns in $X^+(q)$: there are patterns containing all ancestors and children of a fixed node u of q, and patterns containing all ancestors and descendants of a fixed node v of q and all paths ending in some node v of q.

We remark that the in fact our proofs would work equally well for other definitions of $X(q)$; we have somehow arbitrarily chosen this one. The only important thing is that $X^+(q)$ is of polynomial size, and that $X(q)$ is nonempty when q is not a brush.

Let us now describe the dynamic programming approach for computing whether $L_s(p, \emptyset, \emptyset) \cap L(rd(d)) \subseteq L_>(q)$. Namely, for each path p' from the root of q to some its node (a pattern in $\mathsf{PQ}(/)$), for each $q' \in X^+(q)$, and for all sets F, R of size at most one containing labels appearing somewhere in q we compute whether $L_s(p', F, R) \cap L(rd(d)) \subseteq L_>(q')$. Let us emphasize that p' is a path in q, not in p.

It remains to prove the following lemma, describing a single step of our algorithm.

Lemma E.5. *Let $p \in \mathsf{PQ}(/, //, *)$, let F and R be sets of labels of size at most one, let d be a DTD without requirements on the root label, and let $q \in \mathsf{TPQ}(/)$ be such that none of its nodes has a sibling with the same label. Suppose that for each path p' from the root of q to some its node, for each $q' \in X^+(q)$, and for all sets F', R' of size at most one containing labels appearing somewhere in q we know whether $L_s(p', F', R') \cap L(rd(d)) \subseteq L_>(q')$. Then we can compute in polynomial time whether $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$.*

Proof. We have one of four cases:

- If q is a brush, we use Lemma E.4.
- Let $\bot$ be a pattern consisting of a single node labelled by a letter not appearing in d. Using Lemma E.4 we check whether $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(\bot)$, which simply means that $L_s(p, F, R) \cap L(rd(d)) = \emptyset$. If this is the case, then surely $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$.
- Next, for each $q' \in X(q)$ we check whether $L(q') \cap L(rd(d)) = L(q) \cap L(rd(d))$ using Lemma E.6. If this is the case for some q', then as well $L_>(q') \cap L(rd(d)) = L_>(q) \cap L(rd(d))$, thus $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q)$ if and only if $L_s(p, F, R) \cap L(rd(d)) \subseteq L_>(q')$; we replace q by the smaller pattern q', and we repeat the above steps.

- Otherwise, q is not a brush, $L_s(p,F,R)\cap L(rd(d))\neq\emptyset$, and $L(q')\cap L(rd(d))\neq L(q)\cap L(rd(d))$ for each $q'\in X(q)$. In this case Lemma E.7 implies that $L_s(p,F,R)\cap L(rd(d))\not\subseteq L_>(q)$.

□

Now we have to give the two missing ingredients, that is Lemmas E.6 and E.7. Lemma E.6 contains the key idea of the solution. It says that for q and for $q'\in X(q)$ (two patterns in $\mathsf{TPQ}(/)$), the equivalence question can be reduced to multiple smaller instances of the containment question of $\mathsf{PQ}(/)$ in $\mathsf{TPQ}(/)$. This result highly depends the fact that in q no two siblings have the same label; without this assumption the equivalence question (which is in fact a containment question) would be coNP-hard by Theorem 6.3.

Lemma E.6. *Let $q\in\mathsf{TPQ}(/)$ be such that none of its nodes has a sibling with the same label, let $q'\in X(q)$, and let d be a DTD without requirements on the root label. Suppose that for each path $p\in\mathsf{PQ}(/)$ from the root of q to some its node, and for all sets F, R of size at most one containing labels appearing somewhere in q we know whether $L_s(p',F',R')\cap L(rd(d))\subseteq L_>(q')$. Then we can check in polynomial time whether $L(q')\cap L(rd(d))=L(q)\cap L(rd(d))$.*

Proof. As a first step, we check whether $L(q')\cap L(rd(d))$ is empty or not. Instead of doing this directly, we notice that $L(q')\cap L(rd(d))\neq\emptyset$ if and only if $L(q')\cap L(d)\neq\emptyset$. Indeed, for a tree in $L(q')\cap L(rd(d))$, by duplicating some of its subtrees, and reordering siblings, we can obtain a tree in $L(q')\cap L(d)$. Oppositely, take a tree in $L(q')\cap L(d)$ and some embedding of q' into t. It is important that no two siblings of q' have the same label. Thanks to that, whenever in t two siblings have the same label, we can remove a subtree rooted in one of them without removing any node in the image of the embedding. Proceeding in this way we can obtain a tree in $L(q')\cap L(rd(d))$. Knowing this, we can check whether $L(q')\cap L(rd(d))=\emptyset$ using an automaton construction: the language $L(q')\cap L(d)$ is recognized by an automaton of polynomial size; we can check for its emptiness.

It always holds $L(q')\cap L(rd(d))\supseteq L(q)\cap L(rd(d))$. Notice that if $L(q')\cap L(rd(d))=\emptyset$, the answer is immediate. Below we suppose that $L(q')\cap L(rd(d))$ is nonempty, and we consider two cases corresponding to two possible shapes of q'.

Suppose that q' was constructed by the first rule in the definition of $X(q)$; let u be the node used in this rule. We claim that $L(q')\cap L(rd(d))\neq L(q)\cap L(rd(d))$ if and only if there exists a node x present in q but not present in q' for which $L_s(p(parent(x)),\{\mathsf{lab}(x)\},\emptyset)\cap L(rd(d))\not\subseteq L_>(q')$, where $p(parent(x))$ is the path in q from its root to the parent of x. For each x this is known from the assumption of the lemma, so it remains to prove this claim.

Concentrate first on the right-to-left implication: suppose that such x exists. Let y be the parent of x, and let z be the child of u belonging to $p(y)$ (observe that y is a proper descendant of u). Take a tree $t\in L_s(p(y),\{\mathsf{lab}(x)\},\emptyset)\cap L(rd(d))\setminus L_>(q')$. Let f be the (unique) strong embedding of $p(y)$ into t. Take also a tree $t_0\in L(q')\cap L(rd(d))$ that has the smallest possible number of nodes. Then $t_0\notin L_>(q')$, as otherwise we could restrict to the proper subtree into which q' embeds and obtain a smaller tree in $L(q')\cap L(rd(d))$. Let g be the (unique) embedding of q' into t_0. We create t' as follows: in t_0 we remove the subtree rooted at $g(z)$, and in its place we put the subtree of t rooted at $f(z)$. We see that $t'\in L(rd(d))$ by construction: in a tree in $L(rd(d))$ we replace a subtree by a tree in $L(rd(d))$ rooted at a node with the same label. We also see that q' embeds into t': the entire image of g is taken to t' (only its leaf $g(z)$ is replaced by a node with the same label). Recall the nice property of trees in $L(rd(d))$ that there are no two siblings with the same label. Suppose that q could be embedded to t'. If the root of q was mapped to the root of t', then necessarily z was mapped to $g(z)/f(z)$, so x was mapped to a child of $f(y)$. This contradicts with the property that no child of $f(y)$ has label $\mathsf{lab}(x)$. If the root of q was mapped below the root of t', then this mapping in fact maps q either into t or into t_0: the first branching point of q, that is u, is not mapped above the gluing point, that is $g(z)/f(z)$ (recall that the path of t_0 from its root to $g(z)$ is labelled in the same way as the path of t from its root to $f(z)$). This would mean that either $t\in L_>(q')$ or $t_0\in L_>(q')$, which contradicts with our assumptions. Thus we have $L(q')\cap L(rd(d))\ni t'\notin L(q)\cap L(rd(d))$.

Now let us see the left-to-right implication. Take a tree $t\in L(q')\cap L(rd(d))\setminus L(q)$ that has the smallest possible number of nodes. Then $t\notin L_>(q')$, as otherwise we could restrict to the proper subtree into which q' embeds and obtain a smaller tree in $L(q')\cap L(rd(d))\setminus L(q)$. To find x we successively remove nodes of q which are not present in q'; at some moment we have a pattern which still does not embed into t, but after removing a node x it already embeds into t. Let us fix an embedding f of this pattern without x into t. In particular its part embeds the path $p(parent(x))$ into t. Moreover, no child of $f(parent(x))$ is labelled by $\mathsf{lab}(x)$, as otherwise the pattern with x would also embed into t. It means that $t\in L_s(p(parent(x)),\{\mathsf{lab}(x)\},\emptyset)$, thus $L_s(p(parent(x)),\{\mathsf{lab}(x)\},\emptyset)\cap L(rd(d))\not\subseteq L_>(q')$.

Similarly we deal with q' constructed by the second rule in the definition of $X(q)$; let u and v be the nodes used there. We claim that $L(q')\cap L(rd(d))\neq L(q)\cap L(rd(d))$ if and only if there exists a node x present in q but not present in q' such that

- x is a child of u and $L_s(p(parent(x)),\{\mathsf{lab}(x)\},\{\mathsf{lab}(v)\})\cap L(rd(d))\not\subseteq L_>(q')$, or
- x is not a child of u and $L_s(p(parent(x)),\{\mathsf{lab}(x)\},\emptyset)\cap L(rd(d))\not\subseteq L_>(q')$.

For each x this is known from the assumption of the lemma, so it remains to prove this claim.

Concentrate first on the right-to-left implication: suppose that such x exists. Take a tree $t_0\in L(q')\cap L(rd(d))$ that has the smallest possible number of nodes (we have assumed that this set is nonempty). Then $t_0\notin L_>(q')$, as otherwise we could restrict to the proper subtree into which q' embeds and obtain a smaller tree in $L(q')\cap L(rd(d))$. Let g be the (unique) embedding of q' into t_0. We have two cases; suppose first that x is a child of u. Take a tree $t\in L_s(p(u),\{\mathsf{lab}(x)\},\{\mathsf{lab}(v)\})\cap L(rd(d))\setminus L_>(q')$. Let f be the (unique) strong embedding of $p(u)$ into t. We

create t' as follows: in t_0 we remove all subtrees rooted at the children of $g(u)$ other than $g(v)$, and in their place we put all subtrees of t rooted at the children of $f(u)$ other than the one having label $\mathsf{lab}(v)$ (notice that such child exists since $\mathsf{lab}(v)$ is taken to the set R). We see that $t' \in L(rd(d))$ by construction: in t' every "triangle" of a node and all its children comes either from t or from t_0. We also see that q' embeds into t': the entire image of g is taken to t' (as it contains only v, ancestors of $g(v)$, and descendants of $g(v)$). Suppose that q could be embedded into t'. If the root of q was mapped to the root of t', then necessarily x was mapped to a child of $g(u)$. But in t' the set of labels in the children of $g(u)$ is as the set of children of $f(u)$ in t. This contradicts with the property that no child of $f(u)$ has label $\mathsf{lab}(x)$. If the root of q was mapped below the root of t', then this embedding in fact embeds q either into t or into t_0: the first branching point of q, that is u, is not mapped above the children of $g(u)$ (recall that the path of t_0 from its root to $g(u)$ is labelled in the same way as the path of t from its root to $f(u)$). But both in t and in t_0 the pattern q' does not embed anywhere below the root, so q as well. Thus we have $L(q') \cap L(rd(d)) \ni t' \notin L(q) \cap L(rd(d))$.

Next, suppose that x is not a child of u. Let y be the parent of x, and let z be the child of u belonging to $p(y)$. If $t_0 \notin L(q)$, we are done: we have $L(q') \cap L(rd(d)) \ni t_0 \notin L(q) \cap L(rd(d))$. Thus we assume that $t_0 \in L(q)$; let now g be the embedding of the whole q into t_0. Take a tree $t \in L_s(p(y), \{\mathsf{lab}(x)\}, \emptyset) \cap L(rd(d)) \setminus L_>(q')$. Let f be the (unique) strong embedding of $p(y)$ into t. We create t' as follows: in t_0 we remove the subtree rooted at $g(z)$, and in its place we put the subtree of t rooted at $f(z)$. We see that $t' \in L(rd(d))$ by construction: in a tree in $L(rd(d))$ we replace a subtree by a tree in $L(rd(d))$ rooted at a node with the same label. We also see that q' embeds into t': the entire image of q' under g is taken to t'. Suppose that q could be mapped to t'. If the root of q was mapped to the root of t', then necessarily z was mapped to $g(z)/f(z)$, so x was mapped to a child of $f(y)$. This contradicts with the property that no child of $f(y)$ has label $\mathsf{lab}(x)$. If the root of q was mapped below the root of t', then this embedding in fact embeds q either into t or into t_0: the first branching point of q, that is u, is not mapped above the gluing point, that is $g(z)/f(z)$ (recall that the path of t_0 from its root to $g(z)$ is labelled in the same way as the path of t from its root to $f(z)$). But both in t and in t_0 the pattern q' does not embed anywhere below the root, so q as well. Thus we have $L(q') \cap L(rd(d)) \ni t' \notin L(q) \cap L(rd(d))$.

Now let us see the left-to-right implication. Take a tree $t \in L(q') \cap L(rd(d)) \setminus L(q)$ that has the smallest possible number of nodes. Then $t \notin L_>(q')$, as otherwise we could restrict to the proper subtree into which q' embeds and obtain a smaller tree in $L(q') \cap L(rd(d)) \setminus L(q)$. To find x we successively remove nodes of q which are not present in q'; at some moment we have a pattern which still does not embed into t, but after removing a node x it already embeds into t. Let us fix an embedding f of this pattern without x into t. In particular it maps $p(parent(x))$ into t. Moreover, no child of $f(parent(x))$ is labelled by $\mathsf{lab}(x)$, as otherwise the pattern with x would also embed into t. It means that $t \in L_s(p(parent(x)), \{\mathsf{lab}(x)\}, \emptyset)$, thus $L_s(p(parent(x)), \{\mathsf{lab}(x)\}, \emptyset) \cap L(rd(d)) \not\subseteq L_>(q')$. In the case when $parent(x) = u$, we additionally know that a child of $f(u)$ is labelled by $\mathsf{lab}(v)$, so t belongs also to $L_s(p(parent(x)), \{\mathsf{lab}(x)\}, \{\mathsf{lab}(v)\})$. □

LEMMA E.7. *Let $p \in PQ(/, //, *)$, let F and R be sets of labels, let d be a DTD, and let $q \in TPQ(/)$ be such that none of its nodes has a sibling the same label. Suppose that q is not a brush, and $L_s(p, F, R) \cap L(rd(d)) \neq \emptyset$, and $L(q') \cap L(rd(d)) \neq L(q) \cap L(rd(d))$ for each $q' \in X(q)$. Then $L_s(p, F, R) \cap L(rd(d)) \not\subseteq L_>(q)$.*

PROOF. We will say that a pattern p can be embedded into a tree t at a node x, if it can be embedded so that its root is mapped to x.

Consider a tree $t \in L_s(p, F, R) \cap L(rd(d))$. Let f_1 be the strong embedding of p into t. If q does not embed into t, we are done. Otherwise, we will improve the tree t: we will take a lowest node x of t at which q can be embedded, and we will modify the tree so that it is still in $L_s(p, F, R) \cap L(rd(d))$, but q no longer embeds at x, and no new embedding point was introduced on the level of x or below (but possibly q embeds at some new points higher than x). Clearly this procedure terminates in a tree to which q does not embed (the following pair is a parameter which decreases lexicographically: the maximal level of a node x at which q embeds, and the number of such nodes on this level).

Thus consider a node x of t at which q embeds, being farthest from the root. Let f be the embedding of q into t, which maps the root to x. Let u be the node of q which has at least two children, but all its proper ancestors have only one child (it exists since q is not a brush). Let y_1 denote the leaf of p. We have two cases.

Suppose first that for some child v of u the node $f(v)$ belongs to the path from the root of t to $f_1(y_1)$. In this situation we consider the pattern q' obtained from q by removing all nodes except v, ancestors of v, and descendants of v. Since u (the parent of v) has at least two children, we have $q' \neq q$, so $q' \in X(q)$. By assumption, there exists a tree $t_0 \in L(q') \cap L(rd(d)) \setminus L(q)$. Let g be an embedding of q' into t_0; w.l.o.g. we assume that the root of q' is mapped to the root of t_0 (the part above the image of g can be cut off). We create t' as follows: in t we remove all subtrees rooted at children of $f(u)$ other than $f(v)$, and in their place we put subtrees of t_0 rooted at children of $g(u)$ other than $g(v)$. We see that $t' \in L(rd(d))$ by construction: in t' every "triangle" of a node and all its children comes either from t or from t_0. We also see that no node in the image of f_1 was removed; additionally the children below $f_1(y_1)$ have the same labels as in t. Thus $t' \in L_s(p, F, R)$. Suppose that q embeds into t' at x. Then for each child v' of u other than v, the subtree of q rooted at v' is mapped to a subtree of t_0 rooted at a child of $g(u)$. Together with the embedding g of q' into t_0 this gives an embedding of the whole q into t_0, which by assumption does not exists. Thus q does not embed into t' at x. Suppose that q embeds into t' at some node y being on a level greater or equal than the level of x, but $y \neq x$. Then the image of the branching point u under this embedding is neither $f(u)$ nor an ancestor of $f(u)$: either it is on a greater level, or on the same level but this is not $f(u)$. Thus if u is mapped to a node coming from t, then the whole q is mapped to such nodes, which means that q embeds

already into t at this place y. On the other hand, if u is mapped to a node coming from t_0, then all its descendants as well. Then the ancestors of u also can be mapped to appropriate nodes of t_0, since the path of t_0 from its root to $g(u)$ is labelled in the same way as the path of t from x to $f(u)$. But by assumption $t_0 \notin L(q)$, so such situation is impossible. Thus q no longer embeds at x, and no new embedding point was introduced on the level of x or below.

Next, suppose that for no child v of u, $f(v)$ belongs to the path from the root of t to $f_1(y_1)$. In this situation we consider the pattern q' obtained from q by removing all nodes except u, ancestors of u, and children of u. Since q is not a brush, we have $q' \neq q$, so $q' \in X(q)$. By assumption, there exists a tree $t_0 \in L(q') \cap L(rd(d)) \setminus L(q)$. Let g be an embedding of q' into t_0; w.l.o.g. we assume that the root of q' is mapped to the root of t_0 (the part above the image of g can be cut off). We create t' as follows: in t we remove all subtrees rooted at $f(v)$ for all children v of u (notice that there may remain some children of $f(u)$ which are not in the image of f); in their place we put subtrees of t_0 rooted at $g(v)$ for all children v of u. We see that $t' \in L(rd(d))$ by construction: in a tree in $L(rd(d))$ we replace subtrees by trees in $L(rd(d))$ rooted at a node with the same label. We also see that no node in the image of f_1 was removed; additionally the children below $f_1(y_1)$ have the same labels as in t (even if $f(u) = f_1(y_1)$, the set of labels in the children of this node remains unchanged). Thus $t' \in L_s(p, F, R)$. Notice that q no longer embeds into t' at x, as then it would be mapped to the nodes on the path from x to $f(u)$ (which are also present in t_0) and to nodes coming from t_0, so q would also embed into t_0, contrary to the assumption about t_0. Suppose that q embeds to t' at some node y being on a level greater or equal than the level of x, but $y \neq x$. Then the image of the branching point u under this mapping is neither u nor an ancestor of $f(u)$: either it is on a greater level, or on the same level but this is not $f(u)$. Thus if u is mapped to a node coming from t, then the whole q is mapped to such nodes, which means that q embeds already into t at this place y. On the other hand, if u is mapped to a node coming from t_0, then all its descendants as well. Then the ancestors of u also can be mapped to appropriate nodes of t_0, since the path of t_0 from its root to $g(u)$ is labelled in the same way as the path of t from x to $f(u)$. But by assumption $t_0 \notin L(q)$, so such situation is impossible. Thus q no longer embeds at x, and no new fitting point was introduced on the level of x or below. □

THEOREM 6.6. *The following are EXPTIME-complete:*
(1) W-CONTAINMENT *of* $\mathsf{PQ}(/)$ *in a* $\mathsf{PQ}(/, *)$*;*
(2) W-CONTAINMENT *of* $\mathsf{PQ}(//)$ *in a* $\mathsf{PQ}(/, *)$*;*
(3) S-CONTAINMENT *of* $\mathsf{PQ}(/)$ *in a* $\mathsf{PQ}(/, //, *)$*;*
(4) S-CONTAINMENT *of* $\mathsf{PQ}(//)$ *in a* $\mathsf{PQ}(/, //, *)$*;*
all w.r.t. a fixed DTD.

All the cases of Theorem 6.6 will be shown in a very similar way. We focus first on showing EXPTIME-hardness of case (1) and then we adapt the proof a bit to the other cases. For all the cases presence in EXPTIME is due to [36].

The proof of hardness will be divided into three parts. In the first of them we formulate a `Line Triomino Tiling` problem that is PSPACE-complete, and then we extend `Line Triomino Tiling` to a two-player game, showing that it becomes EXPTIME-complete. In the second part we prove PSPACE-hardness of containment by a reduction from `Line Triomino Tiling`. Then, we show how the reduction can be adapted to obtain EXPTIME-completeness.

E.1.1 Triomino Tiling

We introduce `Line Triomino Tiling` problems as a convenient variant of the well-known domino tiling problems (see, e.g., [15, 42]). Actually, our tiling problem will not be defined in the spirit of the well known *corridor tiling* problem, in which one considers tiling of a rectangle. For the purpose of our problem it is more convenient from technical point of view to consider tiling of a line, with a refined notion of constraint. The underlying idea, however, is quite similar to the known corridor tiling problem.

A `Triomino Tiling System` (TTS) is a set of *tiles* T, a set of *triomino constraints* (called also simply *constraints*) $C \subseteq T^3$ and two[9] *final tiles* $t_{f1}, t_{f2} \in T$. For a word $s = t_1 \cdots t_n \in T^*$, a *solution* of a TTS $S = (T, C, t_{f1}, t_{f2})$ with *initial row* s is a function $\lambda : \{1, \ldots, m\} \to T$, with $m \geq n$, called a *tiling*, such that

1. the initial row is indeed the prefix of the tiling, i.e., for all $i \in \{1, \ldots, n\}$ it holds $\lambda(i) = t_i$; and
2. triomino constraints are fulfilled, i.e., for all $i \in \{1, \ldots, m - n\}$ it holds $(\lambda(i), \lambda(i+1), \lambda(i+n)) \in C$.

As we want to prove hardness of the containment problem for a fixed DTD, we also need to consider a fixed TTS. For a TTS S we write `LineTriominoTiling`(S) ($LTT(S)$) for the set of words s such that S has a solution with initial row s.

REMARK E.8. *There exists a TTS S such that `LineTriominoTiling`(S) is PSPACE-complete.*

Remark E.8 can be shown by an uncomplicated reduction from the well known corridor tiling problem. Actually we do not use that Remark E.8, we present it only for completeness, to show that the simplified version of the proof shows PSPACE-hardness of the containment problem.

In order to prove EXPTIME-hardness we consider the game version of LTT. The `Line Triomino Tiling Game` is a two-player game, played by CONSTRUCTOR and SPOILER, whose interaction results in placing a tile. CONSTRUCTOR's goal is to build a correct tiling, and SPOILER's goal is to prevent it.

[9] We need two of them for technical reasons: as we will see below in the definition of the game, the only way for CONSTRUCTOR to win is to propose two different final tiles, out of which SPOILER will choose one.

More formally, we associate with a TTS S and an initial row s a two-player game in the following way. The tiling function λ is already defined for positions from 1 to i by the initial row s. Now values starting from the position $i+1$ are to be chosen. Players choose next tiles in the following way: CONSTRUCTOR offers exactly two different options for a tile, and then SPOILER chooses which of them is placed. At this moment SPOILER is allowed to place a tile that does not fulfil constraints, if CONSTRUCTOR offered such to him. CONSTRUCTOR wins the game if at some moment all the placed tiles fulfil constrains and one of the final tiles t_{f1} and t_{f2} is placed. Otherwise, if the play continues infinitely long (in particular if some constraints are not fulfilled), SPOILER wins. For a TTS S we write `LineTriominoTilingGame`(S) ($LTTG(S)$) for the set of words s such that in the two-player game associated with S and with initial row s player CONSTRUCTOR has a winning strategy.

THEOREM E.9. *There exists a TTS S such that LineTriominoTilingGame(S) is EXPTIME-complete.*

PROOF. We reduce from a problem of determining a winner in a tiling game of a special form. Let us recall this problem. A *tiling system* $S = (T, V, H, T_{fin})$ consists of a finite set T of tiles, two sets $V, H \subseteq T \times T$ of *vertical* and *horizontal constraints*, respectively, and a set of *final* tiles $T_{fin} \subseteq T$. With a tiling system S and a word $w = w_1 \dots w_n \in T^*$ with $n \geq 2$, called the *initial row*, we associate a 2-player game as follows. The word w induces a mapping $\tau\colon \{1, \dots, n\} \times \{1\} \to T$, where $n = |w|$. Two players, called CONSTRUCTOR and SPOILER, alternatingly choose tiles $t \in T$, implicitly defining $\tau(1,2), \tau(2,2), \dots, \tau(n,2), \tau(1,3), \dots$. A tile t is a *legal* move as $\tau(i,j)$ if it satisfies the constraints, that is $i = 1$ or $(\tau(i-1,j), \tau(i,j)) \in H$, and $\tau(i,j-1), \tau(i,j)) \in V$. Players are not allowed to play a non-legal move. CONSTRUCTOR wins the game if at some moment the players have already defined a mapping $\tau\colon \{1, \dots, n\} \times \{1, \dots, m\} \to T$ (for some $m \geq 2$), and $\tau(n,m) \in T_{fin}$. On the other hand, CONSTRUCTOR loses if at some moment one of the players cannot make a legal move, or when the game lasts infinitely long without ending a row by a final tile. For a tiling system S, we denote by `TilingWinner`(S) the set of all strings $w = t_1 \dots t_n$ such that $(t_i, t_{i+1}) \in H$ for each $i \in \{1, \dots, n-1\}$ and CONSTRUCTOR has a winning strategy for the game induced by S and w.

It is a folklore that there is tiling system S for which `TilingWinner`(S) is EXPTIME-hard. Nevertheless, it is convenient for us to give a game in which SPOILER has always exactly two moves. Such a game was for example defined in [9], as follows. Given a system S and an initial row w, a *valid rectangle* is a mapping $\tau\colon \{1, \dots, n\} \times \{1, \dots, m\} \to T$ such that the first row contains w and the constraints given by V and H are satisfied (but the last tile need not to be t_{fin}). A *tiling prefix* for S and w is a valid rectangle plus the beginning of the next row, that is a mapping $\tau\colon \{1, \dots, n\} \times \{1, \dots, m\} \cup \{1, \dots, i\} \times \{m+1\} \to T$ with $i \in \{1, \dots, n\}$. A tiling prefix for S and w is *non-blocking*[10] if the partial row can be completed to form a valid rectangle. Proposition 8 in [9] says that there is a tiling system S_r such that

- `TilingWinner`(S_r) is EXPTIME-hard,
- if $|w|$ is odd, then $w \notin$ `TilingWinner`(S_r),
- for every non-blocking prefix τ of odd length (meaning that it is SPOILER's turn) there are exactly two tiles t that are legal moves.

We will define a TTS S such that there is a (polynomial-time) reduction from `TilingWinner`(S_r) to $LTTG(S)$. Before giving a formal definition, let us first outline the reduction.

- Of course the board of the original game, that is a rectangle, will be written as a word, row after row. Then horizontal constraints have to be checked between neighboring tiles, and vertical constraints—between tiles in distance n.
- To obtain triomino constraints, we pack together horizontal and vertical constraints: we allow triples in which the tiles in distance one satisfy horizontal constraints, and the tiles in distance n satisfy vertical constraints.
- Previously, the tiles in odd columns were chosen by CONSTRUCTOR and in even columns by SPOILER. Now, in every column CONSTRUCTOR proposes two tiles and SPOILER chooses one of them. To deal with CONSTRUCTOR's columns, we create two equivalent variants of each tile; when CONSTRUCTOR presents them, SPOILER chooses one of them. Another variant of each tile is used in SPOILER's columns. By assumption on S_r, if CONSTRUCTOR plays correctly, SPOILER has exactly two legal tiles, so CONSTRUCTOR will propose them and SPOILER will choose one of them (it makes no sense for CONSTRUCTOR to propose a tile that is not legal, since then he will lose).
- There is a problem that after writing the rectangle row by row, the horizontal constrains will be checked also between the last tile in a row and the first tile in the next row. To avoid this, we add an additional column filled by some marker X_i.
- Another problem is that in the last row the horizontal constrains will not be checked: they are checked by triomino constrains only if the next row exists. To deal with this, we do not treat a final tile of the original game as final in the new game; instead we wait one more row and precisely below this tile we place a final tile of the new game.
- Moreover, in the new game a final tile is winning for CONSTRUCTOR not only at the end of row. Thus, we have to ensure using triomino constraints that we place our new final tile only when the original final tile was placed at the end of a row.

[10] In [9] such a prefix was called *valid*.

Let us now define the TTS S. Denote $S_r = (T_r, V, H, T_{fin})$. The set of tiles in S is $T = T_r \times \{\mathsf{c}_1, \mathsf{c}_2, \mathsf{s}\} \cup \{X_1, X_2, Y_1, Y_2, t_{f1}, t_{f2}\}$. The set of triomino constraints in S contains

- "Constructor triominos" $((t_1, \mathsf{c}_i), (t_2, \mathsf{s}), (t_3, \mathsf{c}_j))$ where $(t_1, t_2) \in H$, and $(t_1, t_3) \in V$, and $i, j \in \{1, 2\}$,
- "Spoiler triominos" $((t_1, \mathsf{s}), (t_2, \mathsf{c}_i), (t_3, \mathsf{s}))$ where $(t_1, t_2) \in H$, and $(t_1, t_3) \in V$, and $i \in \{1, 2\}$,
- "last-column triominos" $((t_1, \mathsf{s}), X_i, (t_3, \mathsf{s}))$ where $(t_1, t_3) \in V$, and $i \in \{1, 2\}$,
- "first-column triominos" $(X_i, (t_2, \mathsf{c}_j), X_k)$ where $t_2 \in T_r$, and $i, j, k \in \{1, 2\}$,
- "last-row Constructor triominos" $((t_1, \mathsf{c}_i), (t_2, \mathsf{s}), Y_j)$ where $(t_1, t_2) \in H$, and $i, j \in \{1, 2\}$,
- "last-row Spoiler triominos" $((t_1, \mathsf{s}), (t_2, \mathsf{c}_i), Y_j)$ where $(t_1, t_2) \in H$, and $i, j \in \{1, 2\}$,
- "final triominos" $((t_{fin}, \mathsf{s}), X_i, t_{fj})$ where $t_{fin} \in T_{fin}$, and $i, j \in \{1, 2\}$.

The reduction from $\texttt{TilingWinner}(S_r)$ to $LTTG(S)$ is as follows. If $w \in T_r^*$ is of odd length, then $w \notin \texttt{TilingWinner}(S_r)$. Otherwise, we consider the word $mark(w) \in T^*$ obtained by replacing each letter t of w by (t, c_1) if it is on odd position, or by (t, s) if it is on even position, and appending X_1 at the end. We answer positively if $mark(w) \in LTTG(S)$. Thus it remains to prove that $w \in \texttt{TilingWinner}(S_r)$ if and only if $mark(w) \in LTTG(S)$, for all words $w \in T_r^*$ of even length.

Thus consider a word $w \in \texttt{TilingWinner}(S_r)$ and fix some winning strategy of Constructor in the rectangle game induced by S_r and w. Let $n = |w|$; notice that $|mark(w)| = n + 1$ (in particular triomino constraints are checked for tiles on positions i, $i+1$, $i+n+1$). We need to show a winning strategy in the line game induced by S and $mark(w)$. This will be done in the following way. We suppose that the two games are started simultaneously. Basing on moves of Constructor in the rectangle game (that are done according to his winning strategy) we will say how Constructor should move in the line game, and basing on moves of Spoiler in the line game we will say how Spoiler should move in the rectangle game. If, supposing that the rectangle game is won by Constructor, the line game will be also won by Constructor, then the strategy defined this way is winning.

At each moment the situation in the line game will be such that at positions divisible by $n + 1$ there are X_i tiles, and after removing these tiles, cutting the line into rows of length n, and dropping the marker $(\mathsf{c}_1, \mathsf{c}_2, \mathsf{s})$ we obtain the situation in the rectangle game. Moreover, the marker is c_1 or c_2 at odd positions of rows, and s at even positions. This is the case for the initial situation. Now suppose that Constructor in the rectangle game chooses some tile t as his next move. Then in the line game we (simulating Constructor) choose tiles (t, c_1) and (t, c_2); Spoiler places one of them on the line. Now in the rectangle game it is Spoiler's turn, and by assumption he has exactly two legal moves t_1, t_2 (recall that Constructor is following a winning strategy in the rectangle game, so surely the current situation is a non-blocking prefix, as otherwise Constructor would be unable to win). In the line game it is now our (that is, Constructor's) turn; we propose the two tiles (t_1, s) and (t_2, s), and Spoiler chooses one of them (t_i, s). In the rectangle game we (as Spoiler) choose t_i; recall that this is a legal move. Additionally, if this was the last tile in a row, then in the line game we (as Constructor) propose the two tiles X_1, X_2, and Spoiler chooses one of them. If moreover $t_i \in T_{fin}$ (and this was the last tile in a row), then in the line game we (as Constructor) propose $n - 1$ times tiles Y_1, Y_2 (ignoring Spoiler's responses), and then we propose tiles t_{f1}, t_{f2}.

It remains to see that when Spoiler places one of the tiles t_{f1}, t_{f2} the situation is indeed wining for Constructor, that is that all triomino constraints are satisfied. This is more or less obvious after analyzing the set of available triominos, and recalling that all moves of the rectangle game were legal so the horizontal and vertical constraints are satisfied in the constructed rectangle.

Now consider an initial word $w \in T_r^*$ of an even length n, such that $mark(w) \in LTTG(S)$, and fix some winning strategy of Constructor in the line game induced by S and $mark(w)$. Before giving a strategy in the rectangle game induced by S_r and w, we prove some properties of the strategy in the line game. Let $\lambda\colon \{1, \ldots, m\} \to T$ (for some $m \geq n + 1$) be a situation in that game, obtained while following the winning strategy of Constructor. Then

(1) tiles of the form X_i appear on all positions divisible by $n + 1$, and only there;

if additionally in λ there are no two consecutive tiles of the form (t_{fin}, s) for $f_{fin} \in T_{fin}$ and X_i, then

(2) no tile is of the form Y_i or t_{fi};

(3) after removing tiles at positions divisible by n, and cutting the line into rows of length n, markers used at odd positions are c_i and markers used at even positions are s;

(4) after removing these markers we obtain a tiling prefix of the rectangle game that satisfies all horizontal and vertical constraints.

Indeed, surely for each $i \in \{1, \ldots, m - n - 1\}$ the tiles at positions i, $i + 1$, $i + n + 1$ satisfy triomino constraints, as otherwise Constructor would be unable to win. From the set of available triominos we see that X_i has to occur if $n + 1$ positions before there was some X_j, and cannot occur otherwise. Together with the form of the initial row, this gives us (1). Next, assume that in λ there are no two consecutive tiles of the form (t_{fin}, s) for $f_{fin} \in T_{fin}$ and X_i. Then Constructor will not win earlier than in move $m + n + 1$: he can only win by placing a final tile $n + 1$ positions after such pair. This implies that for each $i \in \{1, \ldots, m - 1\}$ there has to exist some tile t' (that will be placed at position $i + n + 1$) such that the tiles at positions i, $i + 1$, and the tile t' satisfy triomino constraints. This

gives (2), since there is no triomino with Y_i or t_{fi} on the top, and (3), since in all triominos the two kinds of markers alternate, starting with c_i after X_j, as well as (4).

Now we want give a strategy in the rectangle game induced by S_r and w. We perform a simulation as for the opposite direction, but this time we are responsible for moves of CONSTRUCTOR in the rectangle game, and moves of SPOILER in the line game. The correspondence between current situations in both games is as previously. Suppose that it is CONSTRUCTOR's turn in the rectangle game, and CONSTRUCTOR in the line game proposes two tiles. In the line game (as SPOILER) we choose the first of them. Notice that we only consider situations where earlier we have not seen tiles (t_{fin}, s) for $f_{fin} \in T_{fin}$ and X_i as two consecutive tiles, as otherwise the rectangle game would be already won by CONSTRUCTOR. Thanks to properties (1)-(4) we know that the chosen tile is of the form (t, c_i), where t is a legal move in the rectangle game. Knowing this, in the rectangle game we (as CONSTRUCTOR) choose t as the next tile.

Next, consider a moment when SPOILER has chosen some tile t in the rectangle game, and CONSTRUCTOR has proposed some two tiles in the line game. Then in the line game we (as SPOILER) choose tile (t, s); it remains to see that this is necessarily one of the tiles proposed by CONSTRUCTOR. Thanks to properties (1)-(4) (applied to the hypothetical situation after placing one of these tiles by SPOILER), we know that these tiles are of the form (t_1, s) and (t_2, s), where t_1 and t_2 are different legal moves of the rectangle game. Now we have to argue that the current tiling prefix is non-blocking. Indeed, because in the line game CONSTRUCTOR can win, we can somehow continue this game until reaching a position divisible by $n + 1$; thanks to property (1), before the position divisible by $n + 1$ we will not place any X_i, so thanks to property (3) we will obtain a valid rectangle that extends the current tiling prefix. Thus, because the tiling prefix is non-blocking, by assumption there are exactly two legal moves from it. These have to be t_1 and t_2, and t is one of them.

When the line game reaches a position divisible by $n + 1$, and CONSTRUCTOR proposes some two tiles, we choose any of them; by property (1) this has to be X_i. It remains to note that in the line game CONSTRUCTOR wins after a finite time, when a t_{fi} tile is placed, and then by property (2) we already had consecutively tiles (t_{fin}, s) for $f_{fin} \in T_{fin}$ and X_i, that is CONSTRUCTOR has already won in the rectangle game. □

E.1.2 Reduction from the LTT *to the inclusion problem*

For a given LTT(S) instance $s \in T^*$ we construct patterns p_1, q and a DTD d such that there exists a solution for s if and only if $L_w(p) \cap L(d) \not\subseteq L_w(q)$. We will gradually introduce details of the reduction and the needed terminology and define everything formally at the end. This will show PSPACE-hardness of the considered inclusion problem. We however present it only as an intermediate step, to shed a light into a more complicated construction that will reduce from LTTG(S) and show EXPTIME-hardness.

High level description. We will encode correct tilings by trees that conform to a DTD d, into which a pattern p can be weakly embedded, but into which a pattern q cannot be weakly embedded. Pattern p will ensure that the initial row is correct. The DTD together with the pattern q will ensure that a tree is indeed a correct encoding of a tiling that also fulfils the constraints and ends by a final tile.

Every node in such a tree will be either a *trunk node* or a *branch node*. The alphabet Σ of the DTD is a disjoint union of Σ_{tr}, called the *trunk alphabet* (containing *trunk letters*) and Σ_{br}, called the *branch alphabet* (containing *branch letters*). We note that Σ will be a fixed, constant-size set. The schema ensures that the root is a trunk node and, for every trunk node, its parent, if it exists, is also a trunk node and at most one of its children is a trunk node. We refer to the set of trunk nodes as the *trunk* of the tree. In every tree that conforms to the DTD the trunk is a path from the root to some (not necessarily leaf) node. An ℓ-labelled node, for any $\ell \in \Sigma$, is called an *ℓ-node.* The ancestor of a node u being at depth smaller by k is called the *k-ancestor* of u. Similarly, descendants of a node u being at depth greater by k are called k-descendants of u.

Tiles are encoded as words of length $k = |T| + 4$ written on the trunk, where T is the set of tiles from the TTS S. The whole trunk is an encoding of a tiling in the following sense: first the first tile is encoded, then the second tile is encoded, etc.; at the bottom part of the trunk the last tile of the tiling is encoded.

Recall that triomino constraints should be checked between a tile, its successor, and the tile n positions farther. The pattern q in our reduction depends only on the length of the initial row and is otherwise the same for every instance: it is the path starting with letter a, then having $kn + 2$ wildcards and at the bottom ending with letter b. Note that q does not embed into a tree t if and only if the $(kn + 3)$-ancestor of every b-node is not labelled by a.

Branch nodes are used to ensure that the encoding is correct. They enforce occurrence of many b labelled nodes, which strongly restrict the possible forms of the trees in the language of the DTD.

Encoding of a tile. Let $T = \{t_1, \ldots, t_{|T|}\}$, where $t_{f1} = t_{|T|-1}$ and $t_{f2} = t_{|T|}$. We define $\Sigma_{\text{tr}} = \{a, c_1, \ldots, c_{|T|}, d_1, \ldots, d_{|T|-1}, e_1, \ldots, e_{|T|}, f\}$. For every tile there is a unique length k word over Σ_{tr} which encodes that tile. Recall that $k = |T| + 4$. More precisely, the tile t_i, for $i \leq |T| - 2$, is encoded by the word

$$w_i = c_i\, d_{i-1} \cdots d_1\, a\, e_{k-i-3} e_{k+i-4} \cdots e_1\, aa.$$

Note that w_i always has exactly three letters a, at positions $i + 1$, $k - 1$, and k. Letters $c_1, \ldots, c_{k-4}, d_1, \ldots, d_{k-5}$ are the *first block letters*, $e_1, \ldots, e_{k-4}$ are the *second block letters*. Observe also that none of the blocks are ever empty and that, in a correct encoding, the only place where aa occurs is at the end of a tile.

There are two special situations, for $i \in \{|T| - 1, |T|\}$, that is for final tiles. The tile $t_{|T|}$ is encoded as $c_{k-4} d_{k-5} \cdots d_1\, a\, f_1$ and the tile $t_{|T|} - 1$ is encoded as $c_{k-5} d_{k-6} \cdots d_1\, a\, f_2$. Note that the difference is at the end:

there is f_1 instead of e_1aa and f_2 instead of e_2e_1aa. Intuitively letters f_i indicate that there is the end of the tiling encoding and therefore the end of the trunk. Note also that the encodings of final tiles are of length $k-2$ or $k-3$, so shorter than the others; this however does not introduce any problems, as final tiles occur only at the end of the trunk.

Ensuring the constraints. The triomino constraint $(\lambda(i), \lambda(i+1), \lambda(i+n)) \in C$ imposes that choice of a tile $\lambda(i+n)$ is restricted w.r.t. tiles $\lambda(i)$ and $\lambda(i+1)$. In the trunk encodings of $\lambda(i)$ and $\lambda(i+1)$ are, respectively, nk and $(n-1)k$ levels above the encoding of $\lambda(i+n)$.

Triomino constraints are enforced as follows. For every triple $j = (j_1, j_2, j_3)$ such that $(t_{j_1}, t_{j_2}, t_{j_3})$ is not in C, that is, $(t_{j_1}, t_{j_2}, t_{j_3})$ is forbidden by the constraints, we have a special letter $g_j \in \Sigma_{\mathrm{br}}$. Intuitively, this is the letter that is responsible for forbidding that triple. (Again, the number of such letters g_j is constant, since T as well as C are constant.) The DTD enforces that the letter g_j always occurs as a child of the letter c_{j_3}. In the DTD we have the following rule

$$g_j \to g_{j,1} + g_{j,2},$$

where $g_{j,1}, g_{j,2} \in \Sigma_{\mathrm{br}}$ are responsible for forbidding the tile t_{j_1} occurring exactly n tiles above and forbidding the tile t_{j_2} occurring exactly $n-1$ tiles above, respectively. That means that the letter $g_{j,1}$ should prevent a $(kn+2-j_1)$-ancestor labelled by a, while $g_{j,2}$ should prevent a $(k(n-1)+2-j_2)$-ancestor labelled by a. Preventing an ℓ-ancestor labelled by a is realized by having a $(kn+3-\ell)$-descendant labelled by b, since the pattern q forbids a $kn+3$-ancestor of a b labelled node to be labelled by a. Therefore each $g_{j,1}$ labelled node has a $kn+3-(kn+2-j_1) = j_1+1$-descendant labelled by b and each $g_{j,2}$ labelled node has a $kn+3-(k(n-1)+2-j_2) = k+j_2+1$-descendant labelled by b. That is realized by rules

$$g_{j,1} \to b_{j_1}, \qquad g_{j,2} \to b_{k+j_2},$$
$$b_i \to b_{i-1} \text{ for every } i > 1, \qquad b_1 \to b.$$

Notice that, since j_1, k, and j_2 are constant numbers, these rules are also constant.

Initial row. In order for a tree to be a correct encoding of the tiling instance it has to be ensured that the beginning of that tree starts from the encoding of the initial row. We enforce it by the pattern p together with the DTD. Pattern p is of the form $\#aw_{j_1}w_{j_2}\cdots w_{j_n}$, where $\#$ is a special symbol occurring nowhere else and the initial row s is of the form $t_{j_1}t_{j_2}\ldots t_{j_n}$. Furthermore, the DTD requires $\#$ to be the root of the tree, that is $S_d = \{\#\}$.

Notice that due to this unique root symbol, if p weakly embeds into a tree in $L(d)$, then it also strongly embeds into this tree.

Ensuring the correct form. The last, but not the least condition that has to be checked is that only trees that are of the form mentioned above form belong to $(L_w(p) \cap L(d)) \setminus L_w(q)$. This condition is needed to claim that $(L_w(p) \cap L(d)) \setminus L_w(q)$ is nonempty if and only if there is some correct tiling for the instance s.

Concretely, we will enforce that the letters from Σ_{tr} form a path of the form $w_{i_1}w_{i_2}\ldots w_{i_\ell}$ for some $\ell \in \mathbb{N}$ and the last letter of w_{i_ℓ} equals f_1 or f_2. The fact that the letters from Σ_{tr} form a path is already enforced by the DTD. Now we show how it is ensured that on the trunk

(a) the *aa-block*, that is, the two consecutive a letters, is repeated every k nodes,
(b) in between of every two such aa-blocks the word $c_id_{i-1}\cdots d_1ae_{k-3-i}\cdots e_1$ is written, and
(c) below the last aa-block the word $c_{k-3-i}d_{k-4-i}\cdots d_1af_i$ for $i \in \{1,2\}$ is written.

We will ensure (a), (b) and (c) by restricting positions on which letters can occur in the trunk. This will be realized by forcing the property that i-ancestors, for appropriately chosen i, of nodes labelled by some concrete letter cannot be a-nodes.

For $x_1 \leq x_2 \leq x_3 \in \mathbb{N}$ we say that a node u is $(x_1, \hat{x}_2, x_3)$-free if none of its i-ancestors for $i \in (\{x_1,\ldots,x_3\} \setminus \{x_2\})$ is labelled by letter a. Node u in the tree is (x_1, x_3)*-1-free* if it is $(x_1, \hat{x}_2, x_3)$-free for some $x_2 \in \{x_1,\ldots,x_3\}$.

We show now how to ensure that a node is $(nk-x, nk-y)$-1-free, for constants x, y, that is, x and y do not depend on the instance of LTT. Due to the structure of q, note that it is sufficient to ensure that for all but one i such that $x+3 \leq i \leq y+3$ there is some i-descendant labelled by letter b. Consider the following rules of the DTD, which aim at making every $d_{(x,z,y)}$-node $(nk-x+1, \widehat{nk-z}+1, nk-y+1)$-free and every $d_{(x,y)}$-node $(nk-x, nk-y)$-1-free:

$$d_{(x,z,y)} \to b_{x+1}b_{x+2}\cdots b_zb_{z+2}\cdots b_yb_{y+1}$$

(where the sequence on the right is the one consisting of precisely all b_i for $i \in \{x+1,\ldots,z\} \cup \{z+2,\ldots,y+1\}$, so only $z+1$ is missing) and

$$d_{(x,y)} \to d_{(x,x,y)} + d_{(x,x+1,y)} + \cdots + d_{(x,y-1,y)} + d_{(x,y,y)}.$$

Recall that each b_i-node has an i-descendant b-node. As such, the freeness of the $d_{(x,z,y)}$-nodes and the 1-freeness $d_{(x,y)}$-nodes is immediate.

Notice that for a node labelled by a trunk letters $\#$, c_i, d_i, e_i or f_i it is determined which trunk letter will label its child. Therefore in order to stabilize the trunk we have only to demand that child of an a-node is the correct one. There are only four possibilities for a label of a node that has an a-labelled parent: letters a, c_i, e_i and f_i. Thus we demand that

(1) every c_i-node is $(nk, nk-(k-3))$-1-free,
(2) every e_i-node is $(nk-(i+2), nk-(k+i-1))$-1-free,
(3) every f_i-node is $(nk-(i+2), nk-(k+i-1))$-1-free, similarly to an e_i-node, and
(4) every a-node that has an a-labelled parent is $(nk-1, nk-(k-2))$-1-free.

We can ensure this by enforcing that c_i-nodes always have a sibling labelled by $d_{(0,k-3)}$, e_i-nodes and f_i-nodes always have a sibling labelled by $d_{(i+2,k+i-1)}$, and a-nodes that have an a-labelled parent always have a sibling labelled by $d_{(1,k-2)}$.

Notice that conditions (1)-(4) indeed assure conditions (a) and (b). The left pattern enforces that indeed a prefix of the trunk is of the form $\#aw_{i_1}w_{i_2}\dots w_{i_n}$. Then we can show that every next letter on the trunk is placed correctly. As said before, for a trunk node labelled by a letter other than a, its child being on the trunk is always correctly labelled. In the case of an a-node conditions (1)-(4) assure that the child is appropriately chosen, depending on the shift of the considered node with respect to the period of length k.

Formal definition. We now write the whole definition of the patterns p and q and the DTD.

Pattern p was, in fact, already formally defined before and is of the form $\#aw_{j_1}w_{j_2}\cdots w_{j_n}$, where $\#$ is the special symbol occurring nowhere else and the initial row s is of the form $t_{j_1}t_{j_2}\dots t_{j_n}$. Since p does not contain descendant edges or a wildcard, we have that $p \in \mathsf{PQ}(/)$.

Pattern q is of the form $a *^{(kn+2)} b$, where again all edges are short. So, $q \in \mathsf{PQ}(/,*)$.

We now turn to the definition of the DTD (Σ, d, S_d). As mentioned before, we have that $S_d = \{\#\}$. The alphabet Σ of the DTD consists of the two parts Σ_{tr} and Σ_{br}:

$$\Sigma_{\text{tr}} = \{\#, a, c_1, \dots, c_{k-4}, d_1, \dots, d_{k-5}, e_1, \dots, e_{k-4}, f_1, f_2\}$$

and

$$\begin{aligned}\Sigma_{\text{br}} &= \{b, b_1, \dots, b_{2k-4}\} \cup \{d_{(x,y)}, d_{(x,y,z)} \mid 0 \le x \le z \le y \le 2k-5\} \\ &\cup \{g_j, g_{j,1}, g_{j,2} \mid j = (j_1, j_2, j_3) \in \mathbb{N}^3 \text{ and } (t_{j_1}, t_{j_2}, t_{j_3}) \notin C\}.\end{aligned}$$

The rules in the DTD are the following:

$$\begin{array}{ll} \# \to a, & \\ d_i \to d_{i-1} \text{ for } i > 1, & d_1 \to a, \\ e_i \to e_{i-1} \text{ for } i > 1, & e_1 \to a, \\ f_1 \to \varepsilon, & f_2 \to \varepsilon. \end{array}$$

Let $s_i = g_{j^1}g_{j^2}\dots g_{j^r}$, where $j^1, \dots, j^r$ are all the triples of the form (k_1, k_2, i) for which $(t_{k_1}, t_{k_2}, t_i) \notin C$. We have also the rules:

$$c_i \to d_{i-1}s_i \text{ for } i > 1, \qquad c_1 \to as_1,$$

and

$$a \to ad_{(1,k-2)} + \Big(\bigcup_{1\le i\le k-4} c_i d_{(0,k-3)}\Big) + \Big(\bigcup_{3\le i\le k-4} e_i d_{(i+2,k+i-1)}\Big) + \Big(\bigcup_{1\le i\le 2} f_i d_{(i+2,k+i-1)}\Big).$$

The letters from Σ_{br} have the following rules:

$$\begin{array}{ll} b_i \to b_{i-1} \text{ for every } i > 1, & b_1 \to b, \\ b \to \varepsilon, & g_j \to g_{j,1} + g_{j,2}, \\ g_{j,1} \to b_{j_1}, & g_{j,2} \to b_{k+j_2}, \end{array}$$

where we assume that $j = (j_1, j_2, j_3) \in \mathbb{N}^3$. The more complicated ones are:

$$d_{(x,y,z)} \to b_{x+1}b_{x+2}\cdots b_z b_{z+2}\cdots b_y b_{y+1},$$

$$d_{(x,y)} \to d_{(x,x,y)} + d_{(x,x+1,y)} + \dots + d_{(x,y-1,y)} + d_{(x,y,y)}$$

for all $0 \le x \le z \le y \le 2k-5$ (actually not all of them are needed, but we do not aim for optimality here).

E.1.3 Reduction from LTTG *to the containment problem*

Now we will adapt the reduction from LTT to the containment problem in order to obtain a reduction from LTTG to the containment problem. For a given LTTG instance $s \in T^*$ we will construct patterns p, q and a DTD d such that CONSTRUCTOR has a winning strategy in the game for LTTG(s) if and only if $L_w(p) \cap L(d) \not\subseteq L_w(q)$. We will refer to Subsection E.1.2 as the PSPACE construction.

High level description. In the PSPACE construction a tree in $(L_w(p) \cap L(d)) \setminus L_w(q)$ corresponded to a correct tiling, a witness that there exists a solution for LTT. Now a tree in $(L_w(p) \cap L(d)) \setminus L_w(q)$ will correspond to a CONSTRUCTOR's winning strategy in LTTG, witnessing that indeed CONSTRUCTOR wins the game of LTTG(S).

On the high level the construction will be very similar. We also have *trunk nodes* and *branch nodes*. Alphabet Σ of the DTD is also a disjoint union of *trunk alphabet* Σ_{tr} (containing *trunk letters*) and *branch alphabet* Σ_{br} (containing *branch letters*) and all the notation used previously remains in force.

Tiles will be encoded exactly as previously and similarly as before written on the trunk. This time, however, the trunk is not a path. It represents a strategy tree of CONSTRUCTOR, so it is a tree of the following shape. At the top it starts with a path of length $nk+2$ that corresponds to the initial row. Then the trunk branches into two parts that correspond to the first choice that was offered by CONSTRUCTOR; the offered tiles are described by paths of length k. Below, on both sides, one more time there are branching points that correspond to next choice offered by CONSTRUCTOR. In the left branch there is a choice offered by CONSTRUCTOR in the second round after SPOILER's decision of choosing in the first round the left option. Similarly in the right branch there is a choice offered by CONSTRUCTOR in the second round in the case when SPOILER chooses the right option in the first round. Then there are paths of length k and the trunk continues like that. At the end of every trunk path there is an encoding of one of the two final tiles, which corresponds to the fact that independently of the SPOILER's moves CONSTRUCTOR can always finish the game by placing a final tile.

Branching of the trunk. As we already observed trunk is not a path any more, but it will branch every k levels, beside the upper part of the tree. This will be assured by adding a possibility for a-nodes to have two trunk-node children $c_i c_j$, for $i \neq j$, in the case of branching and also by adding a possibility for a-nodes to have only one trunk-node child c_i, but only at the top part of the tree.

The property that an a-node may have one trunk child being a c_i-node only at the top part of the tree will be assured in the following way. Note that the lowest such c_i-node will be at the beginning of the n-th tile, so at depth $2+k(n-1)$. On the other hand, at depth 1 there is always an a-node, but at the depth $1-k$ there is never an a-node (as $1-k<0$ and it does not make any sense to be at such depth). Therefore we will add a restriction that a c_i-node that is the only trunk child of an a-node has to have no a-labelled $kn+1$-ancestor. This can be assured by adding a b_2-node as a sibling of the mentioned c_i-node that does not have a sibling labelled by c_j.

Patterns p and q. Initial row is encoded by p analogously as in the PSPACE construction. There is also the same with pattern q that is of the form $a *^{kn+2} b$.

Ensuring the correct form. The correct form is ensured very similarly as in the PSPACE construction. The only difference is in the details connected with branching of the trunk. In that case for an a-node instead of options $c_i d_{(0,k-3)}$ we add options $c_i c_j d_{(0,k-3)}$ for $i \neq j$ and $c_i b_2$.

Formal definition. For the sake of clarity and completeness we present here the whole definition of the DTD. Patterns p and q have been already defined.

The DTD, of the form (Σ, d, S_d), is as follows. Similarly as in the PSPACE construction $S_d = \{\#\}$. The alphabet Σ is the disjoint union of the trunk alphabet Σ_{tr} and the branch alphabet Σ_{br}:

$$\Sigma_{\mathrm{tr}} = \{\#, a, c_1, \ldots, c_{k-4}, d_1, \ldots, d_{k-5}, e_1, \ldots, e_{k-4}, f_1, f_2\}$$

and

$$\begin{aligned} \Sigma_{\mathrm{br}} &= \{b, b_1, \ldots, b_{2k-4}\} \cup \{d_{(x,y)}, d_{(x,y,z)} \mid 0 \leq x \leq z \leq y \leq 2k-5\} \\ &\cup \{g_j, g_{j,1}, g_{j,2} \mid j = (j_1, j_2, j_3) \in \mathbb{N}^3 \text{ and } (t_{j_1}, t_{j_2}, t_{j_3}) \notin C\}. \end{aligned}$$

Rules of d for the trunk letters are the following:

$$\begin{array}{ll} \# \to a, & \\ d_i \to d_{i-1} \text{ for } i > 1, & d_1 \to a, \\ e_i \to e_{i-1} \text{ for } i > 1, & e_1 \to a, \\ f_1 \to \varepsilon, & f_2 \to \varepsilon. \end{array}$$

Let $s_i = g_{j^1} g_{j^2} \ldots g_{j^r}$, where $j^1, \ldots, j^r$ are all the triples of the form (k_1, k_2, i) for which $(t_{k_1}, t_{k_2}, t_i) \notin C$. We have also the rules:

$$c_i \to d_{i-1} s_i \text{ for } i > 1, \qquad c_1 \to a s_1,$$

and finally the most complicated one:

$$\begin{aligned} a \to a d_{(1,k-2)} &+ \Big(\bigcup_{1 \leq i < j \leq k-4} c_i c_j d_{(0,k-3)} \Big) + \Big(\bigcup_{1 \leq i \leq k-4} c_i b_2 \Big) + \\ &+ \Big(\bigcup_{3 \leq i \leq k-4} e_i d_{(i+2,k+i-1)} \Big) + \Big(\bigcup_{1 \leq i \leq 2} f_i d_{(i+2,k+i-1)} \Big). \end{aligned}$$

The letters from Σ_{br} have the following rules:

$$\begin{array}{ll} b_i \to b_{i-1} \text{ for every } i > 1, & b_1 \to b, \\ b \to \varepsilon, & g_j \to g_{j,1} + g_{j,2}, \\ g_{j,1} \to b_{j_1}, & g_{j,2} \to b_{k+j_2}, \end{array}$$

where we assume that $j = (j_1, j_2, j_3) \in \mathbb{N}^3$. The more complicated ones are:

$$d_{(x,y,z)} \to b_{x+1}b_{x+2}\cdots b_z b_{z+2}\cdots b_y b_{y+1}$$

$$d_{(x,y)} \to d_{(x,x,y)} + d_{(x,x+1,y)} + \ldots + d_{(x,y-1,y)} + d_{(x,y,y)}$$

for all $0 \le x \le z \le y \le 2k-5$.

Correctness. We do not make here a formal correctness proof, instead we argue intuitively that the construction is correct. We aim for showing that there exists a strategy for CONSTRUCTOR in the LTTG if and only if there exists a tree in $T = (L_w(p) \cap L(d)) \setminus L_w(q)$. It is enough to see that trees in T represent all the strategy trees of CONSTRUCTOR.

Formally, our construction does not forbid branching the trunk inside the initial row. This is not a problem, however. The construction above is defined so that it guaranties that the trunk of every tree in T can be restricted so that it

- represents a tree of tiles, such that the top part of the tree of depth $kn+1$ is a path representing the initial row, and below a branching occurs after every tile,
- the sequence of tiles represented on each path of the trunk satisfies triomino constraints, and
- each path of the trunk finishes with a final tile, either t_{f1} or t_{f2}.

Moreover, for every winning strategy tree represented in this way we can attach branch nodes so that it belongs to T. These properties of T imply that T is nonempty if and only if CONSTRUCTOR wins the LTTG. This finishes (the sketch of) the proof, and thus finishes the proof of case (1) of Theorem 6.6.

E.1.4 Proof of Theorem 6.6(2): W-CONTAINMENT *of* $\mathsf{PQ}(//)$ *in* $\mathsf{PQ}(/,*)$ *is EXPTIME-complete*

Here the difference w.r.t. case (1) is that the pattern p will use only descendant edges, instead of child edges, as now only descendant edges are allowed. Moreover, we add a new branch letter \$ at the end of this pattern. Namely, for an initial row $s = t_{j_1}t_{j_2}\ldots t_{j_n}$ we take

$$p = \#//a//w_{j_1}//w_{j_2}//\cdots//w_{j_n}//\$,$$

where w_i is the encoding of the tile t_i, i.e. it is of the form $c_i//d_{i-1}//\cdots//d_1//a//e_{k-i-3}//e_{k+i-4}//\cdots//e_1//a//a$. Here we assume that $i \le |T|-2$, that is the initial row does not contain final tiles.

Therefore we have to enforce that all descendant edges in p will be indeed mapped into child edges in the tree encoding the strategy. This is because we would like to really start the strategy tree from the initial row, not to have the initial row somehow spread in it. To assure this property we will enforce that \$-nodes can be present in the tree only at depths at most $kn+2$. Number $kn+2$ is just the depth of p, so this restriction indeed would imply that all descendant edges of p were mapped to the tree as child edges.

In order to achieve it we will modify the construction from case (1). First, the DTD has to allow a \$-node. This \$-node is needed as a child of an a-node that already has two c_i children. That is in the DTD rule for a we replace $c_ic_jd_{(0,k-3)}$ by $c_ic_jd_{(0,k-3)} + c_ic_jd_{(0,k-3)}\$$.

We also need to prolong the right pattern. Before it was of the form $a*^{nk+2}b$, now $q = a*^{(n+1)k+2}b$, so forbidden distance between an a-node and its descendant b-node is now $(n+1)k+3$. The reason for this modification is that by adding additional k levels we can have more control on the structure of the tree.

That modification implies many small changes in the rules of the DTD. They will however all be in the same spirit: now in order to forbid label a in the x-ancestor, we need to have a b-labelled $((n+1)k+3-x)$-descendant, instead of a b-labelled $(nk+3-x)$-descendant in the previous construction. We will however not write down all the details of these modifications, it is easy to see that they all can be done in a uniform way: whenever in the DTD we were using some b_i, now we should use b_{k+i}.

The way how we assure that any \$-node cannot be too low is the following. Observe that if a \$-node is at depth at most $kn+2$, then it has no a-labelled i-ancestor for all $i \ge kn+2$. Contrarily, suppose that a \$-node is at depth greater than $kn+2$. Looking at the DTD we notice that all ancestors of \$ are trunk nodes. Additionally, above each trunk letter (except # and the topmost a being on depth 1) there has to be some a letter in distance at most $k-3$. Indeed, while going up in the tree, the index in the label grows, and finally when the index is $k-4$ the parent has to be a-labelled. It follows that when a \$-node is at depth greater than $kn+2$, then among its i-ancestors for $i \in \{kn+2, kn+3, \ldots, kn+k-2\}$ there has to be an a-node.

Thus we will demand that every \$-node is has no a-labelled i-ancestor for all $i \in \{kn+2, kn+3, \ldots, kn+k-2\}$. That will be assured by adding the following rule to the DTD:

$$\$ \to b_4b_5\ldots b_kb_{k+1}.$$

(Note here that this construction cannot be obtained without prolonging the pattern q: we do not have nodes b_i for $i < 0$.) Then indeed the only way how a \$-node can achieve this restriction is really to be at depth at most $kn+2$.

One can easily observe that if the pattern p did not get stretched then the rest of the construction works and this finishes the proof.

E.1.5 Proof of Theorem 6.6(3,4): S-Containment *of* $\mathsf{PQ}(/)$ *(or* $\mathsf{PQ}(//)$*) in* $\mathsf{PQ}(/, //, *)$ *is EXPTIME-complete*

These cases follow by a reduction from cases (1) and (2), respectively. Having an instance of (1) or (2) we produce an instance of (3) or (4), respectively, in the following way. We create the a DTD d' by adding to the old DTD d a special new root symbol $\#_0$, which always has a child $\#$, the old root symbol. We create a new pattern p' obtained from p in the following way. In case (3) $p' = \#_0/p$, in case (4) $p' = \#_0//p$. We create the new pattern q' obtained from q in both cases in the same way: $q' = \#_0//q$. One can easily observe that in both cases

$$L_w(p) \cap L(d) \subseteq L_w(q) \iff L_w(p') \cap L(d') \subseteq L_w(q').$$

Moreover pattern p' belongs to $\mathsf{PQ}(/)$ or $\mathsf{PQ}(//)$ in cases (3) and (4), respectively. Similarly pattern q' belongs to $\mathsf{PQ}(/, //, *)$ in both cases. This finishes the reduction and thus proves EXPTIME-hardness for cases (3) and (4).

F. PROOFS FOR SECTION 7

Proposition 7.1. *For all fragments $\mathcal{F}_1$ and $\mathcal{F}_2$ of TPQs we consider in this paper, the* W-Containment *(resp.,* S-Containment*) problem of $\mathcal{F}_1$ in $\mathcal{F}_2$ over graphs is the same as the* W-Containment *(resp.,* S-Containment*) problem of $\mathcal{F}_1$ in $\mathcal{F}_2$ over trees.*

Proof. This proposition follows from Section 5.3 in [34], but to be self-contained we present a proof here.

Consider some two TPQs p and q. We will show that $L_s(p) \subseteq L_s(q)$ holds on trees if and only if it holds on graphs and similarly for the weak inclusion problem. Clearly if $L_s(p) \subseteq L_s(q)$ holds on graphs then it also holds on trees, simply because every tree is a graph. This is also true for the weak inclusion.

Therefore it is enough to show the opposite implication. First focus on the strong inclusion and assume that $L_s(p) \subseteq L_s(q)$ on trees. Assume, towards contradiction that it does not hold on graphs, so there exists some graph G such that p strongly embeds in $\mathsf{root}(G)$, but q does not. Consider now the rooted infinite tree t, which is the unfolding of G from $\mathsf{root}(G)$. Clearly p strongly embeds in t, fix some concrete embedding h. Pattern q does not strongly embed in t, as it does not strongly embed in G. Let t' consists of that nodes of t, which are in the image of h or have some descendant in the image of h. Then p strongly embeds in t', and q does not strongly embeds in t', which is a finite tree. That is in the contradiction with the fact that $L_s(p) \subseteq L_s(q)$.

For the weak inclusion we proceed analogously. Assuming that $L_w(p) \subseteq L_w(q)$ holds on trees, but not on graphs we take the counterexample graph G, unfold it and prune in order to obtain a finite tree t'. Pattern p weakly embeds in t', but q does not, which is a contradiction. That finishes the proof. □

Proposition 7.2. *For all fragments $\mathcal{F}$ of TPQs we consider in this paper, under the nodes-only semantics of DTDs and TPQs on graphs, the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ with respect to DTDs over graphs is the same as the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ over trees.*

Proof. The proof is similar to the proof of Proposition 7.1. Consider some TPQ p and a DTD d. We will show that $L_s(p) \cap L(d) \neq \emptyset$ holds on trees if and only if it holds on graphs and similarly for the weak case. Clearly if $L_s(p) \cap L(d) \neq \emptyset$ holds on trees then it also holds on graphs, simply because every tree is a graph. This is also true for the weak case.

The opposite implication is less trivial. First focus on the strong case. Assume that $L_s(p) \cap L(d) \neq \emptyset$ holds on graphs, so there is some graph $G \in L(d)$ such that p strongly embeds in G. Consider the rooted infinite tree t, which is the unfolding of G from $\mathsf{root}(G)$. Clearly t still conforms to DTD d and there is a homomorphism h from p to t such that $h(\mathsf{root}(p)) = \mathsf{root}(t)$. We will now construct a finite tree $t_{fin} \in L_s(p) \cap L(d)$, thus showing that $L_s(p) \cap L(d) \neq \emptyset$ holds also on trees.

Consider the set X of nodes of t belonging to the image of h and of their ancestors. Let t' consist of that part of t which contains nodes from X and their children. Tree t' is finite and p embeds in t', but it does not necessarily belongs to the language $L(d)$. However, the only nodes of t' in which conditions imposed by d can be not satisfied are its leafs, the places of cut. We construct the tree t_{fin} as follows. We take a tree t' and substitute every its leaf n, labelled by a letter a by a finite tree, which conforms to the DTD d and its root is also labelled by a. Such a tree exists by an assumption that the DTD d is reduced. One can easily observe that $t_{fin} \in L_s(p) \cap L(d)$.

The weak case is solved similarly with one difference. Now the pattern p embeds into a graph G that need not to be rooted. Nevertheless, the $\mathsf{root}(p)$ is mapped to some node n. Then we obtain the infinite tree t_n by unfolding G from that node n. However that tree has root labelled by the label of n, which need not to be in the set of root labels allowed by the DTD d. However, by the assumption that d is reduced there exists a finite tree t_r, which conforms to d and for some of its nodes, say n_r, it holds that $\mathsf{type}(n) = \mathsf{type}(n_r)$. Therefore we obtain t by substituting in the tree t_r the subtree rooted in n_r by the tree t_n. In that way we obtain an infinite tree t which conforms to d and such that pattern p embeds weakly into it. Further we proceed analogously as in the strong case. □

Proposition 7.4. *For all fragments $\mathcal{F}$ of TPQs we consider in this paper, under the nodes/edges semantics of DTDs and TPQs on graphs, the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ with respect to DTDs over graphs is the same as the* W-Satisfiability *(resp.,* S-Satisfiability*) problem of $\mathcal{F}$ over trees.*

Proof. The proof of Proposition 7.2 works also in this case. It is easy to observe that all the constructions in the mentioned proof work equally well in the nodes/edges semantics. □