Static Analysis of Navigational XPath over Graph Databases

Egor V. Kostylev

University of Oxford and University of Edinburgh

Juan L. Reutter

PUC Chile and Center for Semantic Web Research

Domagoj Vrgoč

PUC Chile and Center for Semantic Web Research

Abstract

Most query languages for graph databases rely on exploring the topological properties of the data by using paths. However, many applications require more complex patterns to be matched against the graph to obtain desired results. For this reason a version of the standard XML query language XPath has been adapted to work over graphs. In this paper we study static analysis aspects of this language, concentrating on problems such as containment, equivalence and satisfiability. We show that for the full language all of the problems are undecidable. By restricting the language we then obtain several natural fragments whose complexity ranges from PSpace-complete to ExpTime-complete.

Keywords: Graph Databases, Query Languages, XPath, Static Analysis

1. Introduction

In recent years we have witnessed a proliferation of applications that require functionalities not provided by the relational database model. A vast majority of them, such as social networks, biological databases, or the Semantic Web, have one thing in common—their underlying model is that of a graph. Because of this, managing and maintaining graph-structured data is currently one of the most active topics in the database community, and there are many big commercial vendors, such as IBM [17], Oracle [21] and Facebook [12], offering graph database products.

The most basic task for every data model, including graph databases, is query evaluation. Hence, when designing a query language one is primarily concerned with striking a good balance between expressivity and efficiency: the language should be capable of describing a wide variety of relevant queries, while at the same time having a low complexity of query answering.

To query graph-structured data one can, of course, use traditional languages and treat the model as a relational database. However, modern applications require to pose intricate navigational queries to obtain non-trivial information about the topology of the stored data—a feature that is unsupported by traditional relational databases. For these reasons several languages for querying graphs, such as regular path queries (RPQs) [11] and conjunctive regular path queries (CRPQs) [8, 10], have been proposed and extensively studied. By now we understand very well their evaluation performance, static analysis and expressive power, as well as how their extensions with backward navigation [8], nesting [3], or rational relations [2] behave. What all of these languages (with the sole exception of nested regular expressions [3]) have in common is that they rely on exploring the graph topology using paths. However, as witnessed in, for instance, XML [26], doing navigation using paths alone is often not sufficient, as more complex patterns have to be matched against a graph to obtain desired results. For this reason a graph-based adaptation of the well-studied XML language XPath has recently been proposed [19]. This language, called *graph*

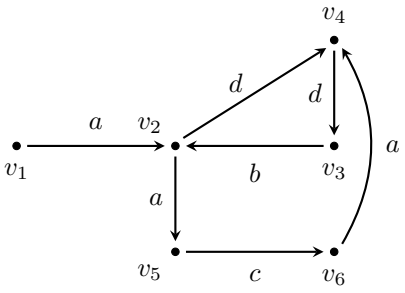

Figure 1: Example of a graph database

XPath, or GXPath for short, enriches the usual path queries with the ability to define more intricate patterns that can occur between two data points.

Evaluation properties of GXPath have been recently studied in [19, 25], and for now we can conclude that it has a good expressivity-evaluation balance for querying graph data. However, very little is known about the properties of static analysis tasks, which are important for query optimisation and building efficient execution plans. Looking to fill this gap, we provide in this paper a detailed complexity analysis of these tasks for GXPath. In what follows we will mainly focus on *query containment*, which asks to determine, given two queries, if the answer set of the first one is always contained in the answer set of the second one. However, we also show how *query equivalence* and *satisfiability* can be solved in a similar manner.

We start by showing that all three problems are undecidable for the full language of GXPath. The reason for undecidability is the presence of negation in node and path formulas, which allows to simulate powerful operations such as the complementation of binary relations. As expected, we show that the decidability of all three problems is restored (it is ExpTime-complete) once we disallow negation on path formulas. Note that this is one of the usual syntactic restrictions of the XPath language [13]. Following the design logic of XPath, our last result shows that the containment and equivalence of formulas without any form of negation is even lower (PSpace-complete).

Plan of the Paper. We formally define the data model, the query language and the static analysis problems we study in Section 2. In Section 3 we show that for the full language the problems are undecidable. In Section 4 we show how decidability can be restored using several syntactic restrictions. We conclude in Section 5.

$$
\begin{array}{rcl}
[\![\top]\!]^G & = & \{v \mid v \in V\} \\
[\![\neg\varphi]\!]^G & = & V - [\![\varphi]\!]^G \\
[\![\varphi \wedge \psi]\!]^G & = & [\![\varphi]\!]^G \cap [\![\psi]\!]^G \\
[\![\varphi \vee \psi]\!]^G & = & [\![\varphi]\!]^G \cup [\![\psi]\!]^G \\
[\![\langle\alpha\rangle]\!]^G & = & \{v \mid \exists v' \ (v, v') \in [\![\alpha]\!]^G\} \\
\\
[\![\varepsilon]\!]^G & = & \{(v, v) \mid v \in V\} \\
[\![a]\!]^G & = & \{(v, v') \mid (v, a, v') \in E\} \\
[\![a^-]\!]^G & = & \{(v', v) \mid (v, a, v') \in E\} \\
[\![[\varphi]]\!]^G & = & \{(v, v) \mid v \in [\![\varphi]\!]^G\} \\
[\![\alpha \cup \beta]\!]^G & = & [\![\alpha]\!]^G \cup [\![\beta]\!]^G \\
[\![\alpha \cdot \beta]\!]^G & = & [\![\alpha]\!]^G \circ [\![\beta]\!]^G \\
[\![\overline{\alpha}]\!]^G & = & V \times V - [\![\alpha]\!]^G \\
[\![\alpha^*]\!]^G & & \text{is reflexive transitive closure of } [\![\alpha]\!]^G
\end{array}
$$

Table 1: Semantics of GXPath formulas over a graph database $G = \langle V, E\rangle$ ('$-$' stands for set-theoretic difference and '$\circ$' for composition of binary relations)

The results on containment were previously presented, without proofs, in a conference paper [18]. This paper additionally contains results on satisfiability, as well as full proofs of all the statements.

2. Preliminaries

Graph Databases

Let Σ be a finite alphabet of *labels*. A *graph database* (or *data graph*) is a Σ-labelled graph $G = \langle V, E\rangle$, where

- V is a finite set of *nodes*,
- $E \subseteq V \times \Sigma \times V$ is a set of *labelled edges*.

A graphical representation of an example graph database is shown in Figure 1, where nodes $v_1, \ldots, v_6$ are connected by edges labelled by a, b, c, d.

Query Language

As in XPath, formulas of GXPath are divided into *node formulas*, returning nodes, and *path formulas*, returning pairs of nodes, which are mutually dependent on each other. We first define the general language and then restrict it to two other flavours that forbid different types of negation.

Definition 2.1. Node formulas φ, ψ *and* path formulas α, β *of* (navigational) GXPath *are expressions satisfying the grammar*

$$
\begin{array}{rcl}
\varphi, \psi & := & \top \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \langle\alpha\rangle, \\
\alpha, \beta & := & \varepsilon \mid a \mid a^- \mid [\varphi] \mid \alpha \cup \beta \mid \alpha \cdot \beta \mid \overline{\alpha} \mid \alpha^*,
\end{array}
$$

where a ranges over labels Σ.

Note that besides the operators in Definition 2.1 the original proposal of GXPath includes data value comparisons [19]. In this paper we are primarily concerned with navigational aspects of graphs, so we did not include this feature of the language in our analysis.

The formal semantics of GXPath with respect to a graph database $G = \langle V, E\rangle$ is given in Table 1: a node formula φ defines a set $[\![\varphi]\!]^G$ of nodes, and a path formula α defines a set $[\![\alpha]\!]^G$ of pairs of nodes.

To gain some intuition of the operators allowed in the language, we briefly compare it with the well-known language of regular path queries. Besides usual regular expressions underlying regular path queries, GXPath path formulas allow for the inverse a^-, which traverses edges in the opposite direction, binary negation $\overline{\alpha}$, which complements the evaluation of α, and tests $[\varphi]$, which check that the condition given by a node formula φ is satisfied. Node formulas are Boolean combinations of tests $\langle\alpha\rangle$, which check for existence of a path satisfying α that starts from this node.

Example 2.2. As an example of a GXPath formula we consider the path formula $a[b^- \wedge a^*c]d$. On the graph in Figure 1 it retrieves the pair of nodes (v_1, v_4), because there is an ad-labelled path between them such that after traversing the a-edge we get to a node that has an incoming b-edge and an outgoing ac-labelled path. However, the pair (v_6, v_3) is not in the answer, because the intermediate node in the only ad-labelled path between them does not satisfy the condition in the brackets.

We also define two different fragments of GXPath, obtained by restricting the use of negation in formulas. These fragments are inspired by well-studied fragments of XPath over XML trees [13].

Definition 2.3. *The* path-positive GXPath*, denoted* $\mathsf{GXPath}^{\text{path-pos}}$*, restricts* GXPath *by forbidding negation of the form* $\overline{\alpha}$ *in path formulas. The* positive GXPath*, denoted* $\mathsf{GXPath}^{\text{pos}}$*, further restricts the path-positive fragment by disallowing unary negation* $\neg\varphi$ *in node formulas.*

Note that the languages we study have close connections with (various variants of) propositional dynamic logic (PDL) [16]. We will use these connections in the proofs throughout the paper, giving precise explanations and references in each particular case.

Static Analysis Problems

The problems we study in this paper are query containment, equivalence and satisfiability. These problems are at the core of many static analysis tasks, such as query optimisation. Next we formally define these problems.

A (node or path) GXPath formula e_1 is *contained* in a formula e_2, written $e_1 \subseteq e_2$, if and only if for each graph database G we have that

$$[\![e_1]\!]^G \subseteq [\![e_2]\!]^G.$$

The formulas e_1 and e_2 are *equivalent* (written $e_1 \equiv e_2$) if and only if $[\![e_1]\!]^G = [\![e_2]\!]^G$ for every G. A formula e is *satisfiable* if and only if there is a graph G such that $[\![e]\!]^G \neq \emptyset$.

All the classes of formulas considered in this paper are closed under union, so the first two of these problems are easily inter-reducible: $e_1 \equiv e_2$ if and only if e_1 and e_2 contain each other, and $e_1 \subseteq e_2$ if and only if $e_1 \cup e_2 \equiv e_2$. That is why we concentrate on containment and obtain results for equivalence as immediate corollaries. Sometimes satisfiability is also reducible to one of these problems; for example, a node formula φ is satisfiable if and only if the containment $\varphi \subseteq \neg\top$ does not hold. However, this is not necessarily the case for all of the languages studied in this paper, and moreover the reduction in the other direction is usually not available. Hence, we study satisfiability separately.

Formally, we consider the following decision problems, parameterized by a class of formulas $\mathcal{Q}$.

Containment ($\mathcal{Q}$)	
Input:	Formulas e_1 and e_2 from $\mathcal{Q}$.
Question:	Is e_1 contained in e_2?

Satisfiability ($\mathcal{Q}$)	
Input:	A formula e from $\mathcal{Q}$.
Question:	Is e satisfiable?

Note that the problems for node formulas are reducible to the corresponding problems for path formulas; for example, φ is satisfiable if and only if $[\varphi]$ is satisfiable. Since our results for these two types of formulas are the same, in what follows we concentrate on node formulas when showing lower bounds and undecidability, and on path formulas in case of upper bounds.

In the rest of the paper we perform a formal complexity analysis of satisfiability and containment for the three classes of GXPath introduced above.

3. Tackling the Full Language

In this section we show that both containment and satisfiability are undecidable for full GXPath. We concentrate on satisfiability; containment follows by the simple reduction given in the previous section.

Theorem 3.1. *The problem* SATISFIABILITY (GXPath) *is undecidable.*

Proof. The proof mainly follows the lines of the undecidability proof for satisfiability in *PDL with extras* that is given in [14]. In particular, we prove undecidability by a reduction from a variant of the tiling problem, which is shown to be undecidable in [7] and [15]. We start by introducing the notation used throughout the proof.

A *tiling instance* $\mathcal{T}$ is a collection $\{T_1, \ldots, T_p\}$ of *tile types* together with two *edge relations* $\sim_h$ and $\sim_v$ (note that these relations are not necessarily symmetric). Intuitively, $T_k \sim_h T_\ell$ means that a tile of type T_ℓ can be placed to the right of a tile of type T_k in a horizontal row, while $T_k \sim_v T_\ell$ means that T_ℓ can be placed above T_k in a vertical column.

A *tiling* of the positive plane $\mathbb{N} \times \mathbb{N}$ *with* $\mathcal{T}$, for natural numbers $\mathbb{N}$, is a function $t : \mathbb{N} \times \mathbb{N} \to \mathcal{T}$ such that for all $i, j \in \mathbb{N}$

– $t(i, j) \sim_h t(i + 1, j)$, and
– $t(i, j) \sim_v t(i, j + 1)$.

Tiling t is *periodic* if there exist positive numbers n and m such that $t(i, j) = t(n+i, j) = t(i, m+j)$ for all $i, j \in \mathbb{N}$. A periodic tiling can be seen as a tiling of a torus, since column $n + 1$ and row $m + 1$ can be "glued" with the left-most column and bottom row, respectively.

Let S_{tiling} denote the set of all tiling instances that allow for tilings of the positive plane, and S_{period} denote the set of all tiling instances that allow for periodic tilings. To prove undecidability we will use the following fact.

Fact 3.2 ([7, 15])**.** *Sets* S_{tiling} *and* S_{period} *are* recursively inseparable*, that is, there is no recursive set* S *such that* $S_{\text{period}} \subseteq S \subseteq S_{\text{tiling}}$*.*

In what follows we first construct a formula $\gamma_\mathcal{T}$ for each tiling instance $\mathcal{T}$ and then show that the set

$$\Phi = \{\varphi \mid \exists G \text{ such that } [\![\varphi]\!]^G \neq \emptyset\} \qquad (1)$$

contains the set $\Gamma_{\text{period}} = \{\gamma_\mathcal{T} \mid \mathcal{T} \in \mathcal{S}_{\text{period}}\}$, and is contained in $\Gamma_{\text{tiling}} = \{\gamma_\mathcal{T} \mid \mathcal{T} \in \mathcal{S}_{\text{tiling}}\}$, which will imply, by Fact 3.2, that Φ cannot be recursive.

To define $\gamma_\mathcal{T}$, fix an alphabet of edge labels $\Sigma = \{R, L, U, D, s, a, e\}$. The intended meaning of the labels is as follows: R represents "right", L "left", U "up" and D "down", while sequences of the form sa^ke code the tile types. Note that we could work only with labels R, U, s, a and e, since it is possible to use R^- instead of L and U^- instead of D, but we opted for the extended alphabet to make the translation easier to understand.

In the reduction below we will use the following node formulas. First, for any path formula β let

$$\texttt{loop}(\beta) = \langle \beta \cap \varepsilon \rangle \wedge \neg \langle \beta \cap \overline{\varepsilon} \rangle$$

(here and in the remainder of the proof we use $\alpha_1 \cap \alpha_2$ as a shorthand for $\overline{\overline{\alpha}_1 \cup \overline{\alpha}_2}$). This formula extracts all nodes v from the graph that have an outgoing β-path with every such path ending at v itself; formally, for any graph database G,

$$[\![\texttt{loop}(\beta)]\!]^G = \{v \in G \mid \exists v' \text{ such that } (v, v') \in [\![\beta]\!]^G, \text{ and } \forall v' \text{ if } (v, v') \in [\![\beta]\!]^G \text{ then } v = v'\}.$$

Second, for every path formula β and every node formula φ let

$$\texttt{when}(\beta, \varphi) = \neg \langle \beta[\neg\varphi] \rangle.$$

The intended meaning of this node formula is to extract all nodes v from a graph such that every β-path starting in v ends with a node belonging to $[\![\varphi]\!]^G$; formally, for any graph database G,

$$[\![\texttt{when}(\beta, \varphi)]\!]^G = \{v \in G \mid \forall v' \text{ if } (v, v') \in [\![\beta]\!]^G \text{ then } v' \in [\![\varphi]\!]^G\}.$$

Consider now a tile instance $\mathcal{T}$ with types $\{T_1, \ldots, T_p\}$, and edge relations $\sim_h$ and $\sim_v$. Based on this instance we construct a node formula $\gamma_\mathcal{T}$ that is a conjunction of two parts, γ_1 and γ_2.

We start with the definition of γ_1. This formula does not depend on the tile instance, but merely guarantees the grid structure for any graph database satisfying $\gamma_\mathcal{T}$. In particular, it enforces a "square" at any position in a database, both in clockwise and in anticlockwise direction. This is done by means of formula `square` that is defined as

the conjunction of the following two formulas:

$$\begin{aligned}\texttt{clockwise} \;=\; & \\ & \texttt{loop}(U \cdot D) \wedge \\ & \texttt{when}(U, \texttt{loop}(R \cdot L)) \wedge \\ & \texttt{when}(U \cdot R, \texttt{loop}(D \cdot U)) \wedge \\ & \texttt{when}(U \cdot R \cdot D, \texttt{loop}(L \cdot R)) \wedge \\ & \texttt{loop}(U \cdot R \cdot D \cdot L),\end{aligned}$$

$$\begin{aligned}\texttt{anticlockwise} \;=\; & \\ & \texttt{loop}(R \cdot L) \wedge \\ & \texttt{when}(R, \texttt{loop}(U \cdot D)) \wedge \\ & \texttt{when}(R \cdot U, \texttt{loop}(L \cdot R)) \wedge \\ & \texttt{when}(R \cdot U \cdot L, \texttt{loop}(D \cdot U)) \wedge \\ & \texttt{loop}(R \cdot U \cdot L \cdot D).\end{aligned}$$

Intuitively, `clockwise` allows us to define a square starting at some point in our graph, and from there going "up", then "right", then "down" and finally "left", finishing at the same point where we started. It also forces the point to be able to complete the square whenever it has an outgoing "up" arrow U. Similarly, `anticlockwise` forces a square starting with "right" and completing it in the corresponding way.

Now, γ_1 simply states that we can make a square at any point:

$$\gamma_1 = \texttt{when}(U^*, \texttt{when}(R^*, \texttt{square})).$$

Formula γ_2 is responsible for forcing the adjacent elements on the grid to agree with the edge relations. It makes use of the following node subformulas, each of which is meant to denote the placement of a tile of a type T_k at some position in the grid: for any $1 \leq k \leq p$ let $\alpha_k = \langle (s \cdot a^k \cdot e) \cap \varepsilon \rangle$, where a^k is the concatenation of k copies of a. On the basis of these formulas, let

$$\alpha \;=\; \left(\bigvee_{1 \leq k \leq p} \alpha_k \right) \;\wedge\; \left(\bigwedge_{1 \leq k \leq p} \left(\alpha_k \rightarrow \bigwedge_{\ell \neq k} \neg \alpha_\ell \right) \right)$$

(here and in the remainder of the proof we use $\varphi \rightarrow \psi$ as a shorthand for $\neg\varphi \vee \psi$). This node formula simply states that precisely one of the α_k is true.

Next, for each tile type T_k, define β_k to be the disjunction of all the α_ℓ such that $T_k \sim_h T_\ell$, that is, the disjunction of representations of the tile types that can be placed to the right of T_k. Similarly, define β^k to be the disjunction of all α_ℓ such that $T_k \sim_v T_\ell$.

Now let `tile` be the formula denoting that a tile is placed correctly in the grid; that is, formally,

$$\texttt{tile} \;= \\ \alpha \wedge \left(\bigwedge_{1 \leq k \leq p} \left(\alpha_k \rightarrow \left(\texttt{when}(R, \beta_k) \wedge \texttt{when}(U, \beta^k) \right) \right) \right).$$

Finally, let

$$\gamma_2 = \texttt{when}(U^*, \texttt{when}(R^*, \texttt{tile})).$$

Having the formula $\gamma_\mathcal{T}$ at hand, we proceed to show that, for Φ defined in (1), $\Gamma_{\text{period}} \subseteq \Phi$ and $\Phi \subseteq \Gamma_{\text{tiling}}$, starting with the second of these inclusions.

We need to show that if $[\![\gamma_\mathcal{T}]\!]^G \neq \emptyset$ for some graph G, then $\mathcal{T}$ can tile the positive plane $\mathbb{N} \times \mathbb{N}$. Take any node $v_{11} \in [\![\gamma_\mathcal{T}]\!]^G$. By γ_1 the formula `square` is true at v_{11}; hence, `clockwise` and $\texttt{loop}(U \cdot D)$ are true. Therefore, there exists a node v_{12} that can be reached from v_{11} by an U-labelled edge. (Note that there also exists a D-labelled edge from v_{12} to v_{11}.) Since $\texttt{when}(U, \texttt{loop}(R \cdot L))$ is also true at v_{11}, there must be a node v_{22} with an R-labelled edge from v_{12} (and with a corresponding L-labelled edge in the opposite direction). Again, this time using the fact that $\texttt{when}(U \cdot R, \texttt{loop}(D \cdot U))$ is true at v_{11}, we get a node labelled v_{21}, connected to v_{22} by an D-labelled edge (and with an U-labelled edge connecting the latter back with v_{22}). Next, we use the fact that $\texttt{when}(U \cdot R \cdot D, \texttt{loop}(L \cdot R))$ is true at v_{11} to get a node v'_{11} to the left of v_{21}. Finally, since $\texttt{loop}(U \cdot R \cdot D \cdot L)$ is true at v_{11}, we have $v'_{11} = v_{11}$.

Similarly since `square` is true at v_{22} (as we can reach it from v_{11} by traversing an U- and then an R-labelled edge), we can find nodes v_{23}, v_{33} and v_{32} that also form a square, as shown in Fig. 2 (note that we do not claim that the nodes v_{ij} are necessarily distinct).

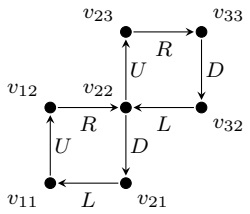

Figure 2: Squares generated from v_{11} and v_{22} by `clockwise`

Note now that since `square` is also true at v_{12}, this node must satisfy `anticlockwise`. In particular, since following an R-labelled edge and then

U-labelled one from v_{12} leads to v_{23} and since $\texttt{when}(R \cdot U, \texttt{loop}(L \cdot R))$ is true at v_{12}, there is a node v_{13} with an L-labelled edge from v_{23} (this also implies that there is an R-labelled edge from v_{13} to v_{23}). Again, since $\texttt{when}(R \cdot U \cdot L, \texttt{loop}(D \cdot U))$ is true at v_{12}, and v_{13} can be reached by $R \cdot U \cdot L$ from v_{12}, there is a node v'_{12} connected with v_{13} by an D-labelled edge (and in the other direction by an U-labelled one). But now, since v_{12} satisfies $\texttt{loop}(R \cdot U \cdot L \cdot D)$ and v'_{12} is reached from v_{12} by a path labelled $R \cdot U \cdot L \cdot D$, we have that $v'_{12} = v_{12}$. Thus there exists a square starting at v_{12} and going in an anticlockwise direction, as illustrated in Fig. 3.

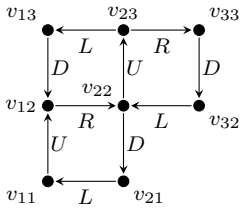

Figure 3: Extension with `anticlockwise` from v_{12}

As already noted, each edge in the construction has a corresponding edge in the opposite direction with the "dual" label (e.g., D is "dual" to U and vice versa). In particular, there is an R-edge from v_{11} to v_{21}, and, by γ_1, we can also complete a clockwise square starting at v_{21}, going through v_{22}, v_{32} and some node v_{31}, and finishing again at v_{21}.

It is straightforward to see that this process can be continued for any number of steps, starting from the main diagonal and completing the squares above it in the anticlockwise direction, while completing the ones below the diagonal in the clockwise direction. Thus we showed that γ_1 guarantees any satisfying graph database to have a square grid starting from v_{11}.

According to γ_2, every node of the grid satisfies `tile`. Hence, on the one hand, there exists a unique α_k that holds at the node. On the other hand, `tile` also guarantees that the function t, defined as $t(i,j) = T_k$ for each node v_{ij} of the grid with α_k true, is a proper tiling. That is, the types of adjacent tiles agree with the edge relations $\sim_h$ and $\sim_v$.

Thus we have shown that if the formula $\gamma_{\mathcal{T}}$ is satisfiable, then $\mathcal{T}$ can tile the positive plane $\mathbb{N} \times \mathbb{N}$. This implies the inclusion $\Phi \subseteq \Gamma_{\text{tiling}}$.

To complete the proof we need to show the inclusion $\Gamma_{\text{period}} \subseteq \Phi$. To this end, consider a periodical tiling t with a tiling instance $\mathcal{T} = \{T_1, \ldots, T_p\}$, which can tile a torus with n columns and m rows. We construct a graph database $G = \langle V, E \rangle$, with $n \times m + p + 1$ nodes, that satisfy $\gamma_{\mathcal{T}}$ as follows.

First let

$$V = \{v_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq m\} \cup \{u_0, \ldots, u_p\}.$$

The following edges in E form an $n \times m$ grid with "glued" sides:

- an R-labelled edge between v_{ij} and $v_{(i+1)j}$, and an L-labelled one in the opposite direction for $1 \leq i < n$ and $1 \leq j \leq m$;
- an R-labelled edge between v_{nj} and v_{1j}, and an L-labelled one in the opposite direction for $1 \leq j \leq m$;
- a U-labelled edge between v_{ij} and $v_{i(j+1)}$, and a D-labelled one in the opposite direction for $1 \leq i \leq n$ and $1 \leq j < m$;
- a U-labelled edge between v_{im} and v_{i1}, and a D-labelled one in the opposite direction for $1 \leq i \leq n$.

To satisfy γ_2, which is responsible for edge relations, we make use of nodes $\{u_0, \ldots, u_p\}$ as follows. First, let $u_0, u_1, \ldots, u_p$ form an a-labelled chain, that is, formally, there is an a-edge between u_k and u_{k+1} in E, for $0 \leq k < p$. Second, for each v_{ij} with $t(i,j) = T_k$ the graph contains an s-labelled edge from v_{ij} to u_0 and an e-labelled edge from u_k to v_{ij}, as illustrated in Fig. 4.

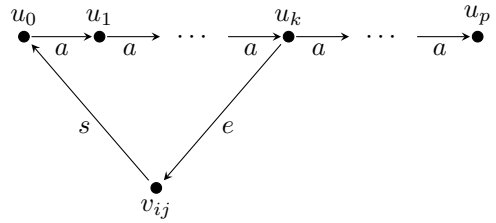

Figure 4: Representation of tile $t(i,j)$ of type T_k

The node v_{11} in the constructed graph database G satisfies $\gamma_{\mathcal{T}}$. Indeed, it satisfies γ_1, because all nodes reachable by U- and R-labelled edges from v_{11} (that is, all of the v_{ij}'s) clearly satisfy `square` by the construction. On the other hand, v_{11} satisfies γ_2, because t is a periodic tiling. In particular, at any node v_{ij} of G precisely one α_k is true, since by construction there is only one s-labelled edge leaving $v_{i,j}$, and only one e-labelled edge entering

$v_{i,j}$. Therefore, to satisfy $\alpha_k = \langle (s \cdot a^k \cdot e) \cap \varepsilon \rangle$ we must traverse this unique s-edge leaving $v_{i,j}$ and finish with the e-edge entering $v_{i,j}$. Moreover, since t is a tiling, any node with an R-labelled edge from v_{ij} satisfies β_k, and the same for a U-labelled edge and β^k.

We conclude that $\Gamma_{\text{period}} \subseteq \Phi$, which together with the previously proved inclusion $\Phi \subseteq \Gamma_{\text{tiling}}$ and Fact 3.2 implies that the set of all satisfiable GXPath node formulas Φ is not recursive, that is, the problem SATISFIABILITY(GXPath) is undecidable. □

As already mentioned, a node formula φ in full GXPath is satisfiable if and only if the containment $\varphi \subseteq \neg\top$ does not hold. Hence, Theorem 3.1 has the following immediate corollary.

Corollary 3.3. *The problem* CONTAINMENT (GXPath) *is undecidable.*

As a final remark we note that in our proof the alphabet Σ of labels does not depend on the input. Hence, the undecidability results hold even for formulas over a fixed alphabet.

4. Restoring Decidability

From the point of view of containment negation is of the most problematic features in query languages, and in GXPath this is not an exception. In this section we show how the decidability of containment is restored when we limit the negation used in formulas. We concentrate on $\mathsf{GXPath}^{\text{path-pos}}$ and $\mathsf{GXPath}^{\text{pos}}$, the two fragments of GXPath outlined in Section 2. We begin with $\mathsf{GXPath}^{\text{path-pos}}$, which forbids negation over path formulas.

Proposition 4.1. *The problem* SATISFIABILITY ($\mathsf{GXPath}^{\text{path-pos}}$) *is* EXPTIME*-complete.*

Proof. Since $\mathsf{GXPath}^{\text{path-pos}}$ is the same as PDL without variables, we can use the EXPTIME decision procedure for satisfiability of PDL formulas, developed in [16, Theorem 8.4], to solve satisfiability of $\mathsf{GXPath}^{\text{path-pos}}$ node formulas. The same bound holds for path formulas, because α is satisfiable if and only if $\langle\alpha\rangle$ is satisfiable.

The lower bound follows from a straightforward adaptation of the known EXPTIME-completeness results on satisfiability of PDL versions close to XPath. For instance, both [1, Section 4.4] and [16, Theorem 8.4] provide a reduction from the acceptance problem for deterministic Turing machines that decide a language in EXPTIME to a version of PDL satisfiability. In order to adapt these techniques to our setting we need to show how to carry out the reduction using a finite alphabet. This can be done by encoding the symbols of the unrestricted alphabet as binary strings (of unbounded length): e.g., a 4-character alphabet can be encoded as a set of strings 00, 01, 10 and 11. For a detailed description of this technique see the EXPSPACE-hardness proof in [5]. □

The next theorem shows that the same complexity bounds hold for the containment problem as well.

Theorem 4.2. *The problem* CONTAINMENT ($\mathsf{GXPath}^{\text{path-pos}}$) *is* EXPTIME*-complete.*

Proof. Since $\mathsf{GXPath}^{\text{path-pos}}$ allows for unary negation, the EXPTIME-hardness follows from Proposition 4.1. Thus in the rest of the proof we concentrate on the upper bound on the complexity. To this end, we show that the problem of containment for $\mathsf{GXPath}^{\text{path-pos}}$ path formulas can be polynomially reduced to satisfiability of $\mathsf{GXPath}^{\text{path-pos}}$ node formulas. The idea of the reduction is similar to the one used in [24], where it is shown that these two problems are inter-reducible for XPath queries on trees.

Let α and β be $\mathsf{GXPath}^{\text{path-pos}}$ path formulas and let Γ be the alphabet of all symbols occurring in α and β plus one additional symbol b. Let now $\Gamma' = \Gamma \times \{0,1\}$, that is, Γ' contains two copies of each label decorated with either 0 or 1. Let α' and β' be formulas obtained from α and β, respectively, by replacing each occurrence of a label a in Γ by $(a,0) \cup (a,1)$. Finally, let $\mathtt{out}$ be the path formula $\bigcup_{a\in\Gamma}(a,1)$. We show that α is contained in β if and only if the formula

$$\varphi := \langle \alpha'[\mathtt{out}] \rangle \wedge \neg \langle \beta'[\mathtt{out}] \rangle$$

is not satisfiable. (To reduce notational clutter, we write $[\gamma]$ instead of $[\langle\gamma\rangle]$ for a path formula γ when checking that a node has an outgoing γ-path.)

Assume first that $\alpha \not\subseteq \beta$. Then, in particular, there is a graph database G with nodes v and v' such that $(v,v') \in [\![\alpha]\!]^G$ but $(v,v') \notin [\![\beta]\!]^G$. Furthermore, one can always find such a graph G that only uses labels from Γ: indeed, only the labels that appear in α and β are relevant, and all the other ones can be replaced by b. Let G' be a Γ'-labelled graph database obtained from G by replacing each

label a by $(a, 0)$ and adding a loop at v' labelled $(b, 1)$. Since v' is the only node in G' with an outgoing edge whose label has 1 as the second component, we get that $v \in [\![\varphi]\!]^{G'}$, as required.

For the other direction, assume that φ is satisfiable. Then there is a graph database G' over Γ' and a node v such that $v \in [\![\varphi]\!]^{G'}$. Let G be a graph obtained from G' by replacing, for any a in Γ, every edge labelled $(a, 0)$ or $(a, 1)$ by an edge labelled a On the one hand, since $v \in [\![\varphi]\!]^{G'}$, there is some v' in G' such that $(v, v') \in [\![\alpha'[\mathtt{out}]]\!]^{G'}$. Hence $(v, v') \in [\![\alpha]\!]^{G}$. On the other hand, by the same reason the node v' must have an outgoing edge with second component equal to 1. Hence, if we had that $(v, v') \in [\![\beta]\!]^{G}$, then we would also get that $(v, v') \in [\![\beta'[\mathtt{out}]]\!]^{G'}$, which contradicts the fact that $v \in [\![\varphi]\!]^{G'}$. Thus $\alpha \not\subseteq \beta$, as required. (Note that it could still be the case that $v \in [\![\langle\alpha\rangle]\!]^{G}$ and $v \in [\![\langle\beta\rangle]\!]^{G}$, but we are interested in binary containment.)

We have thus shown that containment of $\mathsf{GXPath}^{\text{path-pos}}$ path formulas is polynomially reducible to (un)satisfiability of node formulas in the same language. Application of this reduction to the result of Proposition 4.1 completes the proof of the theorem. □

The final fragment we consider is $\mathsf{GXPath}^{\text{pos}}$, which forbids all types of negation from GXPath. This fragment has in fact been considered in the literature under the name of *nested regular expressions* [3, 4], and the complexity of containment for this language has been recently established in [22].

Fact 4.3 ([22])**.** *The problem* Containment ($\mathsf{GXPath}^{\text{pos}}$) *is* PSpace*-complete.*

The last problem left to consider is satisfiability for $\mathsf{GXPath}^{\text{pos}}$. But this problem is trivial: since $\mathsf{GXPath}^{\text{pos}}$ allows for no negation, any formula is satisfiable (a witnessing graph can be constructed by induction on the structure of the formula).

Proposition 4.4. *The problem* Satisfiability ($\mathsf{GXPath}^{\text{pos}}$) *is decidable in constant time.*

5. Conclusions

In this paper we have studied static analysis aspects of the graph query language GXPath. In particular we have tackled the containment, equivalence and satisfiability problems for this language and its fragments. The results are summarised in Table 2. We have shown that for the full language we get undecidability of all three problems, mainly due to the presence of the powerful negation operator that applies to binary relations. An interesting consequence of this result is that satisfiability of PDL queries with negation is undecidable, even if they do not use propositional variables. This follows from the close connection between GXPath and PDL and strengthens the results of [14] in a non-trivial way. Although the full language is undecidable, we have shown that we can restore decidability when limiting negation in a natural way. Such restrictions result in a hierarchy of fragments that have the potential to be useful in designing practical graph query languages. Within these fragments $\mathsf{GXPath}^{\text{path-pos}}$ is the most expressive language that remains decidable. Note, however, that there may well be other fragments of GXPath that subsume $\mathsf{GXPath}^{\text{path-pos}}$ and remain decidable.

	$\mathsf{GXPath}^{\text{pos}}$	$\mathsf{GXPath}^{\text{path-pos}}$	GXPath
Satisfiability	$O(1)$	ExpTime-c	und.
Containment	PSpace-c [22]	ExpTime-c	und.
Equivalence	PSpace-c	ExpTime-c	und.

Table 2: Summary of the complexity results ('-c' stands for 'complete' and 'und.' for 'undecidable')

Lastly, we would like to briefly discuss how our results compare to containment and satisfiability results for XPath over trees. Note that the two problems are inter-reducible when node negation is allowed [24] and that satisfiability of the positive fragment is again trivial, so we will concentrate on containment. First, to the best of our knowledge, the decidability status of GXPath containment over trees with both node and path negation is still unresolved; however, a non-elementary lower bound was shown in [24]. Second, for the path-positive fragment the containment over trees is the same as over graphs (the upper bound follows from [9] and the lower bound from [6]). Finally, for the positive fragment the precise complexity of containment over trees is again not known, as it is usually studied in the presence of sibling axes or for unidirectional fragments of XPath. However, the problem is PSpace-hard [23], and in ExpTime [9, 20].

Acknowledgements. Reutter and Vrgoč are supported by the Millennium Nucleus Center for Semantic Web Research Grant NC120004.

References

[1] Alechina, N., Demri, S., de Rijke, M., 2003. A modal perspective on path constraints. J. Log. Comput. 13 (6), 939–956.

[2] Barceló, P., Libkin, L., Lin, A. W., Wood, P. T., 2012. Expressive languages for path queries over graph-structured data. ACM Trans. Database Syst. 38 (4).

[3] Barceló, P., Pérez, J., Reutter, J., 2012. Relative expressiveness of nested regular expressions. In: Proc. of the 6th Alberto Mendelzon Int. Workshop on Foundations of Data Management (AMW 2012). Vol. 846 of CEUR-WS. CEUR-WS.org, pp. 180–195.

[4] Barceló, P., Pérez, J., Reutter, J. L., 2013. Schema mappings and data exchange for graph databases. In: Proc. of the 16th Int. Conf. on Database Theory (ICDT 2013). ACM, pp. 189–200.

[5] Barceló, P., Reutter, J. L., Libkin, L., 2013. Parameterized regular expressions and their languages. Theor. Comput. Sci. 474, 21–45.

[6] Benedikt, M., Fan, W., Geerts, F., 2008. Xpath satisfiability in the presence of dtds. Journal of the ACM 55 (2).

[7] Börger, E., Gräedel, E., Gurevich, Y., 1997. The Classical Decision Problem. Perspectives in Mathematical Logics. Springer.

[8] Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M., 2000. Containment of conjunctive regular path queries with inverse. In: Proc. of the 7th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR 2000). pp. 176–185.

[9] Calvanese, D., De Giacomo, G., Lenzerini, M., Vardi, M. Y., 2008. Regular XPath: constraints, query containment and view-based answering for XML documents. In: Proc. of the 2008 Int. Workshop on Logic in Databases (LID 2008).

[10] Consens, M., Mendelzon, A., 1990. GraphLog: A visual formalism for real life recursion. In: Proc. of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS). ACM Press, pp. 404–416.

[11] Cruz, I., Mendelzon, A., Wood, P., 1987. A graphical query language supporting recursion. In: Proc. of the ACM Special Interest Group on Management of Data 1987 Annual Conference (SIGMOD). Vol. 16 (3) of SIGMOD Record. pp. 323–330.

[12] Facebook, 2014. Graph Search. `https://www.facebook.com/about/graphsearch`.

[13] Figueira, D., 2010. Reasoning on words and trees with data. Ph.D. thesis, ÉNS de Cachan.

[14] Goldblatt, R., Jackson, M., 2012. Well structured program equivalence is highly undecidable. ACM Trans. Comput. Log. 13 (3), 26.

[15] Gurevich, Y., Koryakov, I., 1972. Remarks on Berger's paper on the domino problem. Siberian Mathematical Journal 13 (2), 319–321.

[16] Harel, D., Kozen, D., Tiuryn, J., 2000. Dynamic Logic. MIT Press.

[17] IBM, 2014. IBM System G Native Store. `http://systemg.research.ibm.com/db-nativestore.html`.

[18] Kostylev, E. V., Reutter, J. L., Vrgoč, D., 2014. Containment of data graph queries. In: Proc. 17th Int. Conf. on Database Theory (ICDT 2014). OpenProceedings.org, pp. 131–142.

[19] Libkin, L., Martens, W., Vrgoč, D., 2013. Querying Graph Databases with XPath. In: Proc. of the 16th Int. Conf. on Database Theory (ICDT 2013). ACM, pp. 129–140.

[20] Marx, M., 2004. XPath with Conditional Axis Relations. In: Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings. pp. 477–494.

[21] Oracle, 2014. Oracle Spatial and Graph. `http://www.oracle.com/technetwork/database/options/spatialandgraph`.

[22] Reutter, J. L., 2013. Containment of nested regular expressions. CoRR abs/1304.2637.

[23] Schwentick, T., 2004. XPath query containment. SIGMOD Record 33 (1), 101–109.

[24] ten Cate, B., Lutz, C., 2009. The complexity of query containment in expressive fragments of XPath 2.0. J. ACM 56 (6), 31.

[25] Vrgoč, D., 2014. Querying graphs with data. Ph.D. thesis, School of Informatics, University of Edinburgh.

[26] XPath 2.0, 2010. XML Path Language (XPath) 2.0 (Second Edition). `www.w3.org/TR/xpath20.`