Fixpoint Path Queries

N. Bidoit and M. Ykhlef*

LaBRI (U.R.A. 1304 du CNRS) Université Bordeaux I
351, Cours de la Libération, F-33405 Talence
email : {Nicole.Bidoit, Mourad.Ykhlef}@labri.u-bordeaux.fr

Abstract

The paper proposes two fixpoint path query languages Path-Fixpoint and Path-While for unstructured data whose expressive power is that of Fixpoint and While queries respectively. These languages are multi-sorted like logic languages integrating fixpoint path expressions.

1 Introduction

Defining query languages for unstructured or semistructured data has received a lot of attention during the last few years [1, 7] [11, 10] [20, 3, 8] [14, 15, 16, 4, 17, 12] [13, 21, 5, 9]. The target applications are biological data management, structured documents, heterogeneous data integration, etc. Unstructured data that is data whose structure is not constrained by a schema, are usually represented in a graph-like manner. The data graph models vary slightly from one approach to an other: graphs may be restricted to trees, data may be associated to edges only, or to edges and terminal vertices, etc. In our framework, we make the choice to represent unstructured data by general multi-graphs called db-graphs whose vertices and edges both carry data or labels. This choice seems to fit in a more natural way applications such as the Web. The major focus of the paper is on defining a graph query language in a multi-sorted calculus like style. Most of the languages proposed so far have been designed as extensions of SQL which has the advantage to give them a commercial flavor. We have deliberately opted for a formal syntax and semantics. The core of our language provides a form of navigational querying based on path expressions. The approach investigated is to extend path expressions with a fixpoint operator. The motivation is the following: path expressions can be viewed as defining monadic predicates over the paths of a graph and dealing with unary properties is known to be unsatisfactory from the point of view of expressive power. Our extension increases the expressive power of most known path query languages as it provides the ability to define n-ary properties over paths for arbitrary n besides the fact of course that it introduces some form of iterated evaluation of path expressions. This feature is the main contribution of our proposal and the source of the expressive power of our language. It is interesting but not surprising to note that although we do not include the Kleene closure in the definition of elementary path expression constructs, it can be expressed by a fixpoint path expression. Moreover, the traditional transitive closure of a binary relation (which is represented by a db-graph) can also be expressed very easily in our language without introducing virtual edges or anything alike. Other interesting queries like collecting pairs of paths having same length can be simply expressed in our language without the need of build-in predicates. The main result of the paper shows that the fixpoint path calculus proposed is equivalent to Fixpoint [2] when considering its inflationary version and is equivalent to While [2] when considering its non inflationary version. These equivalences hide in someway the computation of the paths. Thus we claim that our graph query languages G-Fixpoint and G-While based respectively on the fixpoint path query sublanguages Path-Fixpoint and Path-While subsume all known existing graph query languages. Once again, this is due to the fact that none of the previously defined languages provide the ability to deal with n-ary intentionally defined predicates over paths except for monadic predicates.

Related Work: The language GraphLog [11, 10] specifies queries using path regular expression. It is shown equivalent to FO+TC or to stratified linear Datalog.
The language UnQL [8] includes the kleene closure

*This work is partially supported by the French Projects PRC BD3.

which is not sufficient to express the transitive closure of a binary relation represented by a graph. UnQL on relational data represented as a graph can express only queries in the class of FO queries. The expressive power of UnQL is strictly included in NLOGSPACE.
Lorel [3] and POQL [9] as UnQL can not compute the transitive closure of a binary relation.
WebLog [15] is a language defined in the spirit of Datalog and recursive Datalog like rules play the role of path regular expressions used in other languages. WebLog is not powerful enough to express transitive closure. The same holds for W3QS [14] and WebSQL [16]. The language STRUQL [12] allows one to express the transitive closure of a binary relation although this requires the creation of virtual edges. STRUQL is equivalent to FO+TC. The hypertext language Gram [6] is strictly less expressive than the class of FO queries when the Kleene closure is not included and express some form of transitive closure when it is added.

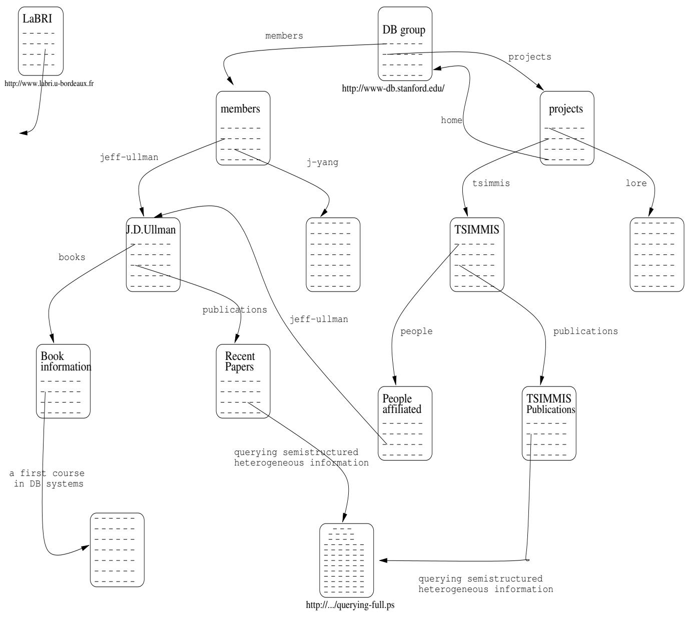
LaBRI
http://www.labri.u-bordeaux.fr
DB group
http://www-db.stanford.edu/
members
projects
members
projects
home
jeff-ullman
j-yang
tsimmis
lore
J.D.Ullman
TSIMMIS
books
publications
jeff-ullman
people
publications
Book information
Recent Papers
People affiliated
TSIMMIS Publications
querying semistructured heterogeneous information
a first course in DB systems
querying semistructured heterogeneous information
http://.../querying-full.ps

Figure 1: An example Web db-graph.

Outline of the paper: The next section presents the data model. Section 3 develops the presentation of the path calculus and its fixpoint and while extensions. Due to space limitation, the graph query languages G-Fixpoint is outlined in section 4. Section 5 investigates the expressive power of the query languages Path-Fixpoint and Path-While.

2 Data model

Definition 2.1 [**db-graph**] *Let $\mathcal{D}$ (resp. $\mathcal{L}$) be an infinite set of values (resp. labels). A db-graph $G = (N, E, \lambda_N, \lambda_E, org, dest)$ over $\mathcal{D}$ and $\mathcal{L}$ is a directed labeled multi-graph where:*

1. *N is a finite set of vertex identifiers and E is a finite set of edge identifiers,*
2. *org (resp. dest) is a mapping from E into N which gives the origin (resp. destination) of edges,*
3. *λ_N is a partial labeling mapping from N into $\mathcal{D}$,*
4. *λ_E is a partial labeling mapping from E into $\mathcal{L}$ such that:*
$$\forall e_1, e_2 \in E \mid (e_1 \neq e_2 \wedge org(e_1) = org(e_2) \wedge dest(e_1) = dest(e_2)) \Rightarrow \lambda_E(e_1) \neq \lambda_E(e_2).$$

Condition 4 says that pairs of vertices having same origin and same destination should have different labels. Although the definition allows multiple labeled edges linking two vertices, we make the restriction that only one labeled edge may link two vertices.

An edge e is totally defined by the vertices of its two extremities and its label $(org(e), dest(e), \lambda_E(e))$. A path is given by a list of edges $\langle e_1, e_2, \ldots, e_k \rangle$ such that $dest(e_i)$=$org(e_{i+1})$ for i=1..k-1. The empty path ε is an empty list of edges. A path is simple w.r.t edges if all its edges are distinct. In the remainder of the paper, a simple path is always intended to be simple w.r.t edges. The mappings *org* and *dest* defined on edges are extended to path in the usual manner.

Definition 2.2 [**Rooted and maximal subgraph**] *A source r of a graph G is a particular vertex such that each vertex of G may be except r should be reachable from r. A rooted graph is a graph with at least one source* [1]*. A subgraph G' of a graph G, rooted at r is maximal if each vertex of G reachable from r is also reachable from r in G', each edge e such that $org(e)$ is reachable from r is an edge of G'.*

A maximal graph rooted at r is totally characterized by its source. A general db-graph can be seen as the set of all its maximal rooted subgraphs. The notion of equality over graphs adopted here is that of bisimulation [19].

Example 2.1 Figure 1 represents a small portion of the Web as an example of a db-graph. The vertices (resp. the edges) represent HTML documents

[1]The path $\langle (n_1, n_2, l_1), (n_2, n_1, l_2) \rangle$ is a db-graph with two sources n_1 and n_2.

(resp. links between documents). Vertex identifiers are http addresses of documents. The vertex identified by *http://www.labri.u-bordeaux.fr* characterizes a maximal rooted subgraph and represents the LaBRI server. The vertex identified by *http://www-db.stanford.edu* has two outgoing edges resp. labeled by members and projects which point to subgraphs containing information about the members of the DB group at Stanford and resp. its projects.

3 Path calculus

A graph query intends to retrieve some maximal rooted subgraphs of the database graph. The retrieval process is essentially supported by specifying paths and roughly speaking, the subgraphs returned by a query are either those rooted at the destination of these paths or containing them. Thus it obviously turns out that the major component of our graph query language is a path query language. The core of our path query language is rather classical and based on path expressions. However our language is made more powerful by introducing fixpoint on path formulas which, intuitively, allows to define intentional n-ary predicates over paths. Thus fixpoint path expressions introduce much richer "contexts" than simple path expression for selecting subgraphs.

The language used to define path expressions and formulas is very similar to a multi-sorted first order language. Because in our data model, data are associated to vertices, our path expressions are slightly different from those of Lorel [20, 3] or UnQL [8]. They may contain data variables as abstractions of the content of vertices.

In the following, we assume that $\mathcal{D}$ and $\mathcal{L}$ [2] are fixed and that three sorts of variable set are given: a set of graph (resp. label, and data) variables denoted $X, Y, Z, \dots$ (resp. $x, y, z, \dots$, and $\alpha, \beta, \gamma, \dots$). For the sake of simplifying the presentation and despite the fact that we deal with three sorts of variables, we define four sorts of terms. A **graph term** is either a rooted graph or a graph variable. A **path term** is either a simple path or a graph variable (it will be clear from the context when a graph variable is a path variable and when it is a graph variable). A **label term** (resp. a **data term**) is either a label in $\mathcal{L}$ (resp. a value in $\mathcal{D}$) or a label (resp. data) variable.

Definition 3.1 [**Path expression**] *A path expression over $\mathcal{D}$ and $\mathcal{L}$ is defined recursively by:*

1. *a (elementary) path expression is an expression of one of the following forms:*
   (*a*) *a graph variable X*[3] (*b*) *a label term t.*
   X and t are pre-free and post-free path expressions.
2. *$t \lhd s$ (resp. $s \rhd t$) where s is a pre-free (resp. post-free) path expression and t is a data term.*
   $t \lhd s$ (resp. $s \rhd t$) is pre-bound (resp. post-bound).
3. *$s_1.s_2$ where s_1 and s_2 are path expressions.*
   If s_1 is pre-free (resp. pre-bound) then $s_1.s_2$ is pre-free (resp. pre-bound) and if s_2 is post-free (resp. post-bound) then $s_1.s_2$ is post-free (resp. post-bound).

The path expression *members.y.publications* intends to represent all paths of length three whose first (resp. last) edge is labeled by *members* (resp. *publications*).

The path expression "DB group"$\lhd X \rhd$"Ullman" intends to capture all paths whose origin (resp. destination) contains the data "DB group" (resp. "Ullman"). It is pre-bound because the origin is constrained (here by a value) and post-bound because the destination is also constrained.

Note that we do not include path expressions of the form s^* where * denotes the Kleene closure as it is usually done in most path query languages. We will see later on that these expressions are special cases of fixpoint path expressions. Note also that we do not make use of a special symbol # usually called "joker" and intended to abstract any label. The paths captured in a db-graph G by a path expression are now defined formally. The definition makes use of the notion of active domain of a db-graph G, *adom*(G) (not detailed here).

Definition 3.2 [**Spelling**] *Let G be a db-graph. Let s be a path expression whose set of variables is $Var(s)$. Let ν be a valuation of $Var(s)$ over adom(G). The set of simple paths spelled in G by s under the valuation ν, denoted $Spell_G(\nu(s))$ is defined by:*

1. *s is X and $Spell_G(\nu(s))=\{\nu(X)\}$,*
2. *s is t and*
   $Spell_G(\nu(s))=\{\langle e\rangle \mid e \in E \text{ and } \lambda_E(e) = \nu(t)\}$,
3. *s is $t \lhd s_1$ and $Spell_G(\nu(s))=$*
   $\{p \mid p \in Spell_G(\nu(s_1)) \text{ and } \lambda_N(org(p)) = \nu(t)\}$,
4. *s is $s_1 \rhd t$ and $Spell_G(\nu(s))=$*
   $\{p \mid p \in Spell_G(\nu(s_1)) \text{ and } \lambda_N(dest(p)) = \nu(t)\}$,

[2]It is possible to consider the domain of labels identical to the domain of data i.e data over nodes and edges may be considered the same.

[3]In a path expression, a graph variable X abstracts a path.

5. *s is $s_1.s_2$ and $Spell_G(\nu(s))= \{p_1.p_2 \mid p_1 \in Spell_G(\nu(s_1))$ and $p_2 \in Spell_G(\nu(s_2))$ and $dest(p_1) = org(p_2)\}$* [4].

Example 3.1 Let G be the path

$$\langle (n_1, n_2, l_2), (n_2, n_1, l_1), (n_1, n_3, l_2)\rangle$$

and let s be the path expression $l_1.l_2$. The set of simple paths $Spell_G(\nu(s))$ is

$$\{\langle (n_2, n_1, l_1), (n_1, n_2, l_2)\rangle,\ \langle (n_2, n_1, l_1), (n_1, n_3, l_2)\rangle\}.$$

Path queries are specified by path formulae.

Definition 3.3 [**Atomic path formula**] *An atomic path formula is an expression having one of the following form:*

1. *a path expression,*
2. *$t_1 = t_2$ where t_1 and t_2 are terms of the same sort among path, label, data; $=$ is the (polymorphic) equality predicate symbol,*
3. *$t \in s$ where t is a path term, s is a path expression and $\in$ is a set membership predicate symbol.*

Note here that paths are compared using equality. Intuitively, the formula $t \in s$ intends to check whether t is one of the paths spelled by the path expression s. A general path formula is defined as usual by introducing logical connectors and quantifications.

The notion of "free" and "bound" occurrences of variables is defined in the usual manner. In the remaining, we will assume that path formulas are normalized in the sense that no distinct pair of quantifiers bind the same variable and no variable occurs both free and bound. Variable substitution may be used to normalize a path formula [2]. The set of variables having at least a free occurrence in the path formula φ is denoted by Free(φ).

Intuitively, the path formula
"DB group"$\lhd X \rhd$"Ullman" $\wedge\ \neg\exists U, V\ X \in U.projects.V$
related to the Web database, will be true for a path valuation p of X such that p connects the "*DB group*"'s page to the "*Ullman*"'s page and p does not contain any edge labeled by projects.

Path queries (simple ones) are now defined in a straightforward manner.

[4]Note that we use the same symbol to denote the concatenation over path expressions and the concatenation of paths. Recall also that $\epsilon.p = p.\epsilon = p$ for any path p where ϵ is the empty path.

Definition 3.4 [**Path Query**] *A path query is an expression of the form $\{(X_1, \ldots, X_n) \mid \varphi\}$ where φ is a path formula, Free(φ)= $\{X_1, \ldots, X_n\}$ and for $i = 1..n$, X_i is a graph (path) variable.*

Note that in general, a path query will return **tuples** of paths.

Example 3.2 [**Navigation and Content search**] The following path query, evaluated on the Web database, will return all paths linking the "DB group"'s page and the "*Ullman*"s'page.

$$\{\ (X)\ \mid\ \text{"}DB\ group\text{"} \lhd X \rhd \text{"}Ullman\text{"}\}$$

Example 3.3 [**Common edge**] The path query below is a yes/no query and will return "yes" if the db-graph queried contains two distinct paths having at least a common edge.

$$\{\ \mid\ \exists X, Y, Z, U, V, y \quad (X.Y.Z\ \wedge U.Y.V)\ \wedge\ Y \in y \\ \wedge\ (\neg(X = U)\ \vee\ \neg(Z = V))\}$$

The next definitions formally define the answer of a path query. The notion of active domain needs to be extended before (not detailed here).

Definition 3.5 [**Path formula satisfaction**] *Let φ be a path formula and ν be a valuation of Free(φ) over adom. The db-graph G satisfies the formula φ under the valuation ν, denoted $G \models \varphi\,[\nu]$, if*

1. *φ is a path expression and $Spell_G(\varphi\,[\nu]) \neq \emptyset$.*
2. *φ is $t_1 = t_2$ and $\nu(t_1) = \nu(t_2)$ if t_1 and t_2 are both path/label/data terms.*
3. *φ is $t \in s$ and $\nu(t) \in Spell_G(\nu(s))$.*
4. *$G \models \varphi\,[\nu]$ is defined in the usual manner when φ is either of the form $(\phi \wedge \psi)$, or $(\phi \vee \psi)$, or $\neg\phi$, or $(\exists x)\ \phi$, or $(\forall x)\ \phi$.*

Note that if φ is the path expression X then there exists at least a valuation of X, namely $\nu(X) = \epsilon$, such that $Spell_G(\varphi\,[\nu]) \neq \emptyset$. To extract non empty paths through a path expression X, one needs to explicitly express it by $X\ \wedge\neg(X = \epsilon)$. As this situation may occur frequently, it could be convenient to introduce an abbreviation for such a formula. This is not done here because obvious.

Definition 3.6 [**Path query answering**] *The image of a db-graph G under the path query q specified by $\{(X_1, \ldots, X_n) \mid \varphi\}$ is $q(G)$=$\{(\nu(X_1), \ldots, \nu(X_n)) \mid \nu$* valuation of Free($\varphi$) over *adom and $G \models \varphi[\nu]\}$.*

Our path query language is very similar to the relational calculus although because of the absence of a schema we do not have relation predicate symbols. However, the reader would have noticed that path formulas define intentional n-ary relations over paths. Just as in the case of the relational calculus we now provide an inflationary extension of path queries with recursion by introducing a fixpoint operator construct allowing the iteration of path formulas evaluation up to a fixpoint. The difficulty to define such fixpoint operator resides in the absence of relation predicate symbols. It turns out that a rather simple notation convention solves this difficulty.

Definition 3.7 **[Path fixpoint operator]**
Let $\varphi(s_{X_1}, \ldots, s_{X_n}, X_1, \ldots, X_n)$ be a path formula with $2 \times n$ free variables namely $s_{X_1}, \ldots, s_{X_n}$ called distinguished free variables and $X_1, \ldots, X_n$.
$\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)$ is called a fixpoint path expression of arity n and given a db-graph G, it denotes the relation that is the limit of the inflationary sequence $\{\mathcal{S}_k\}_{k\geq 0}$ defined by:

1. *$\mathcal{S}_0 = \{(\varepsilon, \ldots, \varepsilon)\}$* [5]
2. *$\mathcal{S}_{k+1} = \mathcal{S}_k \cup \varphi(\mathcal{S}_k)$ where*

   $\varphi(\mathcal{S}_k)$ denotes the union of the evaluation of the path queries $\{(X_1, \ldots, X_n) \mid \varphi[\nu_k]\}$ [6] *where ν_k is any valuation of the distinguished variables $s_{X_1}, \ldots, s_{X_n}$ induced by $\mathcal{S}_k$.*

   A valuation ν_k is induced by $\mathcal{S}_k$ if there exists a tuple $(v_1, \ldots, v_n)$ in $\mathcal{S}_k$ such that $\nu_k(s_{X_i}) = v_i$ for $i = 1..n$.

As usual, the n-ary fixpoint expression $\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)$ can be used to build a more complex path formula. For instance, now $\mu^+_{s_{X_1},\ldots,s_{X_n}}(\varphi)(t_1, \ldots, t_n)$ where each t_i is a path term, can be considered as an atomic path formula. Nesting fixpoint operators does not raise any problem. The extension of the path calculus with the fixpoint construct μ^+ is called **Path-Fixpoint**. It is straightforward to see how a partial non inflationary extension of the path calculus can be build. Because of space limitation, we do not provide here the presentation of this extension called **Path-While**.

Note that because of the initialization of the relation $\mathcal{S}_0$ the inflationist fixpoint of a path expression is never empty as it contains at least a tuple of empty paths. It may be convenient to get rid of this tuple in the limit of the sequence $\{\mathcal{S}_k\}_{k\geq 0}$. This is not done here.

[5]$(\varepsilon, \ldots, \varepsilon)$ is the tuple of empty paths of arity n.
[6]We have taken some liberty with the writing of path queries here in order to simplify the presentation.

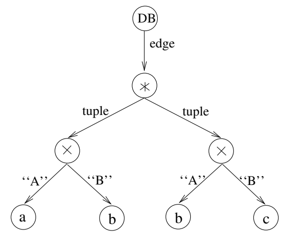

Figure 2: Representation of a relational database.

Example 3.4 The following queries are examples of path queries followed by a brief description of what each query does.

1. $\{\ (X) \mid \mu_{s_X}(\exists x, y\ \ X \in x.y\ \vee\ X \in s_X.x.y)\ (X)\}$
2. $\{\ (X) \mid \mu_{s_X}(X \in a.b \vee X \in s_X.a.b)\ (X)\}$
3. $\{\ (X) \mid \mu_{s_X}(\exists x, y\ (X \in x.y)$
   $\vee\ (X \in s_X.x.y \wedge \exists Y\ \ s_X \in Y.x.y))\ (X)\}$
4. $\{\ (X, Y) \mid \mu_{s_X,s_Y}(\exists x\ (X \in x\ \wedge\ Y \in x)$
   $\vee\ (X \in s_X.x\ \wedge\ Y \in s_Y.x))\ (X, Y)\}$
5. $\{\ (X, Y) \mid \mu_{s_X,s_Y}($
   $\exists x, y, \alpha, \beta\ \ (X \in \alpha \triangleleft x \triangleright \beta\ \wedge\ Y \in \alpha \triangleleft y \triangleright \beta)$
   $\vee\ \ (X \in s_X.x \triangleright \beta\ \wedge\ Y \in s_Y.y \triangleright \beta))\ (X, Y)\}$
6. $\mathbf{SL}(X, Y)$:=$\{\ (X, Y) \mid \mu_{s_X,s_Y}(\exists x, y\ (X \in x \wedge Y \in y)$
   $\vee\ (X \in s_X.x\ \wedge\ Y \in s_Y.y))\ (X, Y)\}$
7. $\mathbf{TC}(X, Y) := \{\ (X, Y) \mid \mu_{s_X,s_Y}($
   $\exists P\ \ P \in edge.\text{“}*\text{”} \triangleleft tuple \triangleright \text{“}\times\text{”}$
   $\wedge\ \exists \alpha, \beta\ \ ((X \in P.\text{“}A\text{”} \triangleright \alpha\ \wedge\ Y \in P.\text{“}B\text{”} \triangleright \beta)$
   $\vee\ (\exists U, V\ \ (X = s_X\ \wedge\ Y = V\ \wedge\ U \in P.\text{“}A\text{”} \triangleright \alpha$
   $\wedge\ s_Y \triangleright \alpha\ \wedge\ V \in P.\text{“}B\text{”} \triangleright \beta))))\ (X, Y)\}$

The first query returns paths of even length. This query can be expressed in most known languages using the Kleene closure of the path expression (#.#) where # is the so-called *joker* symbol. The second query is equivalent to the Kleene closure $(a.b)^*$ and demonstrates that although the Kleene closure is not included explicitly in our language, it can be expressed through a fixpoint expression. The third query is a generalization of the Kleene closure $(a.b)^*$ where a and b are not constants. The fourth (resp. fifth) query returns pairs of paths with identical sequence of edges (resp. vertices). The sixth query is rather interesting as it extracts pairs of paths having same length. This query cannot be expressed neither in Lorel, nor in UnQL. POQL can express it only by using the interpreted function length over paths. This query is rather important for sequence data. One can express a query that extracts all paths generated for example by the language $\{a^n b^n \mid n \in \mathbb{N}\}$ by using the predicate **SL** of query 6. The last query computes (besides the pair of empty paths) pairs of paths whose respective pairs of destination is the transitive closure of a binary relation R represented in a naive way by a db-graph (see Figure 2). It proves that our path query language is powerful enough to express the transitive closure without requiring the creation of new edges as it is done in [12].

4 Graph calculus

Now that path formulas (in Path-Fixpoint or Path-While) are available, it remains to explain how to combine these queries with graph constructs in order to define graph queries. Because of space limitation, the graph calculus is not formally presented. We just provide some highlights. An atomic graph formula of type $t : s$ is called an entry. Roughly speaking, $t : s$ tells that s spells at least one path of the graph t. An atomic graph formula $s[t]$ is called a selector. Roughly speaking, such a formula checks that the graph t has a source which is the destination of a path spelt by s.

Example 4.1 The following queries are examples of graph queries followed by a brief description of what each query does.

1. [**Identity query**] $\{ X \mid \exists Y \; Y = \epsilon \wedge Y[X]\}$ or $\{ X \mid \exists Y \; Y = \epsilon \wedge X : Y\}$
2. $\{ X \mid \mathit{tsimmis.publications}[X]\}$
3. $\{ X \mid \exists Y, Z$ $(\text{“}\mathit{DB\ group}\text{”} \triangleleft \mathit{Y.members.Z.books}[X])\}$
4. $\{ X \mid \forall y \; (\neg (\exists Y \; \mathit{members.y.Y.publications}[X]) \vee (\exists U \; \mathit{tsimmis.U.people.y}))\}$
5. $\{ X \mid \mathit{edge}.\text{“}*\text{”} \triangleleft \mathit{tuple} \triangleright \text{“}\times\text{”}[X]\}$
6. $\{ X \mid \exists Y, Z \; (Y[X] \wedge Z[X] \wedge \neg(Y = Z) \wedge \neg(Y = \epsilon) \wedge \neg(Z = \epsilon))\}$
7. [**Cycles**] $\{ X \mid \exists x \; X : x \wedge x[X]\}$
8. $\{ (X, Y) \mid \exists Z_1, Z_2 \; (\mathbf{TC}(Z_1, Z_2) \wedge Z_1[X] \wedge Z_2[Y] \wedge \neg(Z_1 = \epsilon) \wedge \neg(Z_2 = \epsilon))\}$

The second query retrieves all publications of Tsimmis. The third graph query retrieves the books of the Stanford DB group members. The fourth query returns publications whose authors are all members of the Tsimmis project. The fifth query returns all tuples of a binary relation R (see Figure 2). The sixth query returns all maximal subgraph reached by two non identical paths. The eighth query illustrates how our language can be generalized to generate tuples of graphs as answers. This query returns the transitive closure of the binary relation R by selecting the extremities of the pairs of paths returned by the fixpoint path query (query 7. of Example 3.4) of course discarding the pair of empty paths.

5 Expressive power

In this section, we investigate the expressive power of Path-Fixpoint and Path-While. The two results presented now are complementary and show the equivalence between Path-Fixpoint over db-graphs (resp. Path-While) and Fixpoint over relational instances (resp. While). However these equivalences hide the fact that all the languages including Fixpoint over relational instances manipulate simple path whose computation is left out of the discussion [18]. The first result maps the unstructured data framework on a specific relational structure and shows the equivalence between Path-Fixpoint (resp. Path-While) queries over db-graphs and Fixpoint (While) queries over instances of the specific relational schema. The second result goes the other way around: relational instances are mapped on db-graphs and it is showed that Fixpoint (resp. While) queries over relational instances can be expressed by Path-Fixpoint (resp. Path-While) queries over the corresponding db-graphs.

Theorem 5.1 *There exists a mapping $\mathcal{T}_{Un-to-Rel}$ from db-graphs to relational instances over a fixed schema $\mathcal{R}$ and mapping Path-Fixpoint (resp. Path-While) queries onto Fixpoint (resp. While) queries such that:*

1. *if q_{path} is a Path-Fixpoint (resp. Path-While) query and G is a db-graph then*

$$\mathcal{T}_{Un-to-Rel}(q_{path}(G)) = (\mathcal{T}_{Un-to-Rel}(q_{path}))(\mathcal{T}_{Un-to-Rel}(G)).$$

2. *if q_{rel} is a Fixpoint (resp. While) query over $\mathcal{R}$ then there exists a Path-Fixpoint (resp. Path-While) query q_{path} such that $\mathcal{T}_{Un-to-Rel}(q_{path})$ is equivalent to q_{rel}.*

Theorem 5.2 *There exists a mapping $\mathcal{T}_{Rel-to-Un}$ from instances over any relational schema to db-graphs and mapping Fixpoint (resp. While) queries onto Path-Fixpoint (resp. Path-While) queries such that:*

1. *if q_{rel} is a Fixpoint (resp. While) query and r is an instance over the relational schema $\mathcal{R}$ then*

$$\mathcal{T}_{Rel-to-Un}(q_{rel}(r)) = (\mathcal{T}_{Rel-to-Un}(q_{rel}))(\mathcal{T}_{Rel-to-Un}(r)).$$

References

[1] Abiteboul, S. Querying semistructured Data. In *ICDT* (1997), pp. 1–18.

[2] Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases.* Addision-Wesley, 1995.

[3] Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. The Lorel Query Language for Semistructured Data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.

[4] Abiteboul, S., and Vianu, V. Queries and Computation on the Web. In *ICDT* (1997).

[5] Abiteboul, S., and Vianu, V. Regular Path Queries with Constraints. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Tucson, Arizona, 12–15 May 1997).

[6] Amann, B., and Scholl, M. Gram : A graph data model and query language. In *4th ACM Conf. on Hypertext and Hypermedia* (1992), pp. 138–148.

[7] Buneman, P. Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Tucson, Arizona, 12–15 May 1997), pp. 117–121.

[8] Buneman, P., Davidson, S., Hillebrand, G., and Suciu, D. A query language and optimization techniques for unstructured data. *SIGMOD Record (ACM Special Interest Group on Management of Data) 25*, 2 (1996).

[9] Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. From Structured Documents to Novel Query Facilities. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data* (Minneapolis, Minnesota, 24–27 May 1994), R. T. Snodgrass and M. Winslett, Eds., pp. 313–324.

[10] Consens, M., and Mendelzon, A. GraphLog: a Visual Formalism of Real Life Recursion. In *Proc. 9th ACM Symp. on Principles of Database Systems* (1990), pp. 404–416.

[11] Consens, M. P., and Mendelzon, A. O. Expressing Structural Hypertext Queries in GraphLog. In *ACM Hypertext'89 Proceedings* (1989), Information Retrieval II, pp. 269–292.

[12] Fernandez, M., Florescu, D., Levy, A., and Suciu, D. A Query Language for a Web-Site Management System. *SIGMOD Record (ACM Special Interest Group on Management of Data) 26*, 3 (1997), 4–11.

[13] Frohn, J., Lausen, G., and Uphoff, H. Access to Objects by Path Expressions and Rules. In *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings* (Los Altos, CA 94022, USA, 1994), J. Bocca, M. Jarke, and C. Zaniolo, Eds., Morgan Kaufmann Publishers, pp. 273–284.

[14] Konopnicki, D., and Shmueli, O. W3QS: A Query System for the World-Wide Web. In *VLDB* (1995), pp. 54–65.

[15] Lakshmanan, L. V. S., Sadri, F., and Subramanian, I. N. A declarative language for querying and restructuring the Web. In *Sixth International Workshop on Research Issues in Data Engineering: interoperability of non traditional database systems: proceedings, February 26–27, 1996, New Orleans, Louisiana* (1996), IEEE, Ed., IEEE Computer Society Press, pp. 12–21.

[16] Mendelzon, A. O., Mihaila, G. A., and Milo, T. Querying the World Wide Web. In *PDIS* (1996), pp. 80–91.

[17] Mendelzon, A. O., and Milo, T. Formal Models of Web Queries. In *Proceedings of the sixteenth ACM Symposium on Principles of Database systems* (1997), pp. 134–143.

[18] Mendelzon, A. O., and Wood, P. T. Finding Regular Simple Paths in Graph Databases. *SIAM Journal on Computing 24*, 6 (December 1995), 1235–1258.

[19] Milner, R. *Communication and concurrency.* Prentice Hall, 1989.

[20] Quass, D., Rajaraman, A., Sagiv, Y., and Ullman, J. Querying Semistructured Heterogeneous Information. *Lecture Notes in Computer Science 1013* (1995), 319–344.

[21] Van den Bussche, J., and Vossen, G. An Extension of Path Expressions to Simplify Navigation in Object-Oriented Queries. *Lecture Notes in Computer Science 760* (1993), 267–282.