Towards a Normal Form for Extended Relations Defined by Regular Expressions

András Benczúr and Gyula I. Szabó

Eötvös Loránd University, Faculty of Informatics,
Pázmány Péter sétány, 1/C, Budapest, 1118 Hungary
abenczur@inf.elte.hu, gyula@szaboo.de

Abstract. XML elements are described by XML schema languages such as a DTD or an XML Schema definition. The instances of these elements are semi-structured tuples. We may think of a semi-structure tuple as a sentence of a formal language, where the values are the terminal symbols and the attribute names are the nonterminal symbols. In our former work [13] we introduced the notion of the extended tuple as a sentence from a regular language generated by a grammar where the nonterminal symbols of the grammar are the attribute names of the tuple. Sets of extended tuples are the extended relations. We then introduced the dual language, which generates the tuple types allowed to occur in extended relations. We defined functional dependencies (regular FD - RFD) over extended relations. In this paper we rephrase the RFD concept by directly using regular expressions over attribute names to define extended tuples. By the help of a special vertex labeled graph associated to regular expressions the specification of substring selection for the projection operation can be defined. The normalization for regular schemas is more complex than it is in the relational model, because the schema of an extended relation can contain an infinite number of tuple types. However, we can define selection, projection and join operations on extended relations too, so a lossless-join decomposition can be performed.

1 Introduction

XML has evolved to become the de-facto standard format for data exchange over the World Wide Web. XML was originally developed to describe and present individual documents, it has also been used to build databases. Our original motivation for the introduction of the regular relational data model [13] was to find a good representation of the XML ELEMENT type declaration. The instances of a given element type in an XML document can be considered as a collection of data of complex row types. The set of attribute names in the row types are the element names occurring in the DTD declaration of the element. In the case of recursive regular expression in the element declaration, there are possibly infinite number of different row types for the element instances. The same attribute name may occur several times in a type instance. This leads to the problem of finding a formal way to define the projection operator, similar to the relational

Y. Manolopoulos et al. (Eds.): ADBIS 2014, LNCS 8716, pp. 11–24, 2014.
© Springer International Publishing Switzerland 2014

algebra, on the syntactical structure of the data type. That is necessary to define the left and right side of a functional dependency. We defined the attribute sequence by a traversal on the vertex labeled graph associated to the regular expression of the DTD. This form is also good to define attribute subsequences for the projection operator, for the selection operator and for equijoin operator. Set operations can be extended in a straightforward way, so this leads to the full extension of relational algebra operators. Using the extension of projection and equijoin (or natural join) the join dependency can be defined in the same way as in the relational model.

Motivation. Our previous model [13] could be effectively used for handling functional dependencies (FD). In the relational model FDs offer the basis for normalization (e.g. BCNF), to build non-redundant, well-defined database schema. But our model cannot handle the join operation among instances (that is used to secure lossless join decomposition) because the projection of a schema according to a set of nodes or two joined schemas would not necessarily leads to a new, valid schema. We need an improved model for regular data bases. To denote a regular language we can use regular expressions, our actual model bases upon a graph representation for regular expressions. This model is more redundant than our last one, but it is capable for handling database schema normalization.

Contributions. The main contribution of this paper is the concept of extended relations over the graph representation for regular expressions. We rephrase regular functional dependencies and also define regular join dependencies that constrain extended relations. We determine the schema of an extended relation as $(IN, \ldots, OUT)$ traversals on the graph representation for a given regular expression. We apply the classical Chase algorithm to a counterexample built on this graph. In this way, we show that the logical implication is decidable for this class of functional dependencies.

2 Related Work

As far as we know, each XML functional dependency (XFD) concept involves regular expressions or regular languages. Arenas and Libkin [2] prove different complexities for logical implication concerning their tree tuples XFD model according to the involved regular expressions. They prove quadratic time complexity in case of simple regular expressions. Our new model represents all possible instances of the regular expression at the same time and so it differs from theirs.

The notion data words has been introduced by Bouyer et al. in [4], based upon finite automata of Kaminski et al. [8]. Data words are pairs of a letter from a finite alphabet and a data from an infinite domain. Our concept differs substantially from data words: we assign data values (selected from infinite domains) to letters (from a finite alphabet) after generating a sentence by a regular expression. For data words letters and data values are processed together. Libkin and Vrgo[9] define regular expression for data words. They analyze the complexity of the main decision problems (nonemptiness, membership) for these regular

expressions. Their model is similar to ours but our point of view is differs from theirs: we view finite subsets of the set of data words and specify dependencies over them.

3 Extended Relations

Let us start with the definition of extended relation given by a regular language.

Definition 1 (Extended Relation for Regular Types). *Let L be a regular language over the set of attribute names U. Let $w = w_1 \dots w_n \in L$ a sentence, then we say that w is a regular tuple type over U. Let $dom_u; u \in U$ be sets of data values, then $\{(w_1 : a_1, \dots, w_n : a_n) \,|a_i \in dom_{w_i}\}$ is the set of possible tuples of type w. A finite subset of these tuples is an instance of the regular relation. We say that the set of these tuple types for all $w \in L$ compose the schema of a regular relation based on L.*

We have introduced the notion of the extended relation [13] for a regular language associated with its dual language. The sentences of the dual language are either the concatenated nonterminals used by generating a regular sentence or the states of the accepting automaton, visited during the acception process. Equivalently, the dual language can be given by a vertex labeled graph with a unique IN and OUT node as start and end nodes. The vertex labels along each traversal on this graph (from IN to OUT) represent a schema for the extended relation (we get the sentences of the regular language by valuation). As said in Sect. 1 the dual language model cannot handle the join operation among instances because two joined schemas would not necessarily realize a new, valid schema.

We need an improved model, based upon a suitable graph representation for regular expressions. In our new model we use regular expressions over attribute names to directly define regular relational schemes (e.g. DTD element descriptions), and create the corresponding tuples by valuation (picking data values from suitable domains) similarly to relational databases. In the next Section we present a finite graph representation for the sentences denoted by a regular expression. This graph representation should support node-selection for the projection operation.

4 Graph Representation for Regular Expressions

Definition 2 (Regular Expression Syntax). *Let Σ be a finite set of symbols (alphabet), then a regular expression RE over Σ (denoted by RE_Σ, or simply RE, if Σ is understood from the context) is recursively defined as follows:*
$RE ::= 0|1|\alpha|RE + RE|RE \circ RE|RE^*|RE^?$
where $\alpha \in \Sigma$

For a given regular expression RE we denote the set of alphabet symbols appearing in RE by $[RE]$.

There are efficient constructions of finite state automaton from a regular expression [16,7,5]. The classical algorithm of Berry and Sethi [3] constructs efficiently a DFA from a regular expression if all symbols are distinct.

Berry-Sethi's algorithm constructs a deterministic automaton with at most a quadratic number of transitions [11] and in quadratic computing time (inclusive of marking and unmarking symbols) [6] with respect to the size of the input regular expression (the number of its symbols).

Example 1. Let $G(\{S, A, B\}, \{a, b\}, S, P)$ be a regular grammar, where $P = \{S \Rightarrow a\,S, S \Rightarrow b\,S, S \Rightarrow a\,A, A \Rightarrow bB, B \Rightarrow a\}$.

The regular expression $RE = (a + b)^*\, a\; b\; a$ generates the regular language L(G) too. Fig. 1 shows the graph of the non-deterministic FSA constructed by the Berry-Sethi algorithm (BSA). The nodes represent the states of the automaton: they are distinct. Each node complies with a symbol in the regular expression (small letters), they are not distinct after unmarking. We assign the ingoing edge symbol to each node (capital letter) as vertex label. The language, generated by the vertex labels of the visited nodes, is equivalent with the dual language iff the symbols in the regular expression are distinct.

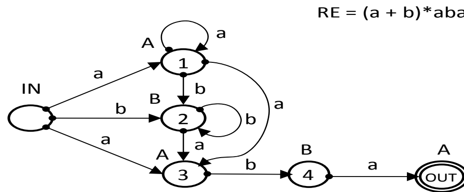

Fig. 1. Graph of the automaton for Example 1 constructed by BSA

As shown in Example 1, for a given regular expression RE we can construct a vertex labeled connected digraph G(RE), with a unique source (IN) and a unique sink (OUT) which represents RE so that that regular language denoted by RE consists of the $(IN, \ldots, OUT)$ traversals on G(RE). This graph is not too large (the number of its vertices equals to the number of symbols in RE), but it is not optimal for our aims because the different $(IN, \ldots, OUT)$ traversals have mostly common subpaths. We need another construction for the graph representation of regular expressions with disjoint $(IN, \ldots, OUT)$ traversals (regardless of IN and OUT). We will construct a graph from vertices picked from a suitably large symbol set Γ. We assume that $\{IN, OUT\} \subseteq \Gamma$ and by picking a node $v \in \Gamma$ we remove it from Γ. The vertices IN and OUT get the labels IN and OUT, respectively. We denote the empty traversal (IN, OUT) by $\triangleright\triangleleft$.

Algorithm 1. *Construction of the Graph-Representation for a regular expression.*

Input: regular expression RE (built from the alphabet Σ),
Output: vertex labeled digraph G(RE)=(V,E) representing RE.

1. *if* $RE = 0$ *or* $RE = 1$*, then* $V = \{IN, OUT\}$ *and* $E = \{(IN, OUT)\}$.

2. *if* $RE = A, A \in \Sigma$*, then we pick a node* $v \in \Gamma$*, set* $V = \{IN, OUT, v\}$*, and* $E = \{(IN, v), (v, OUT)\}$*. We label the node* v *with* A.

3. *if* RE_1 *and* RE_2 *are regular expressions, then* $G(RE_1 + RE_2)$ *will be formed by uniting the* IN *and* OUT *nodes of* $G(RE_1)$ *and* $G(RE_2)$*, respectively.*

4. *if* RE_1 *and* RE_2 *are regular expressions, then in order to build the graph* $G(RE_1 \circ RE_2)$ *we first rename the* OUT *node of* $G(RE_1)$ *and the* IN *node of* $G(RE_2)$ *to* $JOIN$ *(Fig. 2), then unite them and connect all "left" paths with all "right" paths while eliminating the* $JOIN$ *node (Fig. 3).*

5. *if* RE *is a regular expression, then* $G\left(RE^?\right) = G(RE) \cup (IN, OUT)$.

6. *if* RE *is a regular expression, then in order to build the graph* $G(RE^*)$ *we first pick a node* $v \in \Gamma$*, then we create the graph* $G^*(RE) = G(RE) \cup \{v\}$ *(It means that* $V^* = V \cup \{v\}$*, the node* v *gets the special label* $STAR$*). Let us denote* $\{a_1, \ldots, a_n\}$ *the nodes with ingoing edge from* IN *and* $\{z_1, \ldots, z_n\}$ *the nodes with outgoing edge to* OUT*, respectively. Let us create the graph* $G_{IN}(RE, STAR) = \cup_{i=1}^{n}(v, a_i)$ *and the graph* $G_{OUT}(RE, STAR) = \cup_{i=1}^{n}(z_i, v)$*, respectively. Then* $G(RE^*) = G^*(RE) \cup G_{IN}(RE, STAR) \cup G_{OUT}(RE, STAR) \cup (IN, STAR) \cup (STAR, OUT)$.

Theorem 1. *The* $(IN, \ldots, OUT)$ *traversals on the graph representation* $G(RE)$ *for the regular expression* RE *constructed by Alg. 1 generate exactly the regular language* $L(RE)$.

Proof. Algorithm 1 constructs the representation graph so that each elementary step of building a regular expression (Def. 2) will be covered. With induction by the length of the expression and using the regular expression semantics we yield the result.

Example 2. The graph representation for the regular expression $RE = (a + b)^*$ $a\ b\ a$ constructed by the Alg. 1 may be seen on Fig. 4. This graph represents the same regular language as its counterpart on Fig. 1 but it consists of disjoint traversals.

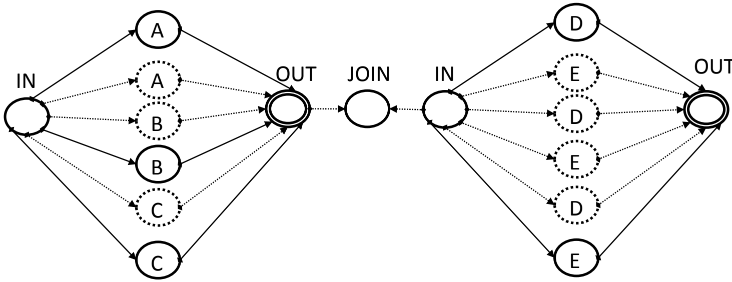

Fig. 2. First joining step of concatenation for two RE graphs

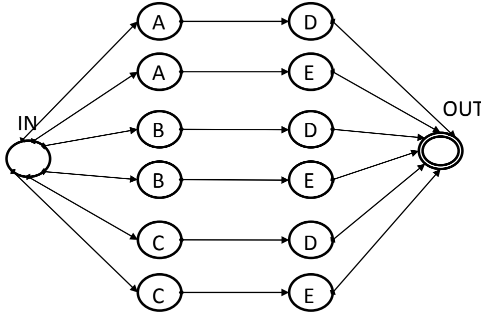

Fig. 3. Eliminating the JOIN node from the concatenation of two RE graphs

Regular expressions present a compact form for specifying regular languages. We look at the sentences of this regular language as types (schemas) of complex value tuples. We can represent these types as IN-OUT traversals on a graph constructed from the symbols in the regular expression. We say that this graph is the schemagraph for the regular expression.

Definition 3 (Schemagraph for Regular Expression). *Let RE be a regular expression built from the alphabet Σ. We say that the graph G is the schemagraph for the regular expression RE (denoted by $G(RE)$) iff*

1. *is a directed, (not necessarily strongly) connected graph,*
2. *has a unique source (IN) and a unique sink (OUT),*
3. *fulfills $OutDegree(IN) = InDegree(OUT)$,*
4. *for any two $P_A = (IN, A_1, \ldots, A_n, OUT)$, $P_B = (IN, B_1, \ldots, B_m, OUT)$ is true that $\{IN, OUT\} \subseteq P_A \cap P_B$, and if $v \in P_A \cap P_B$, then $label(v) \in \{IN, OUT, STAR\}$,*

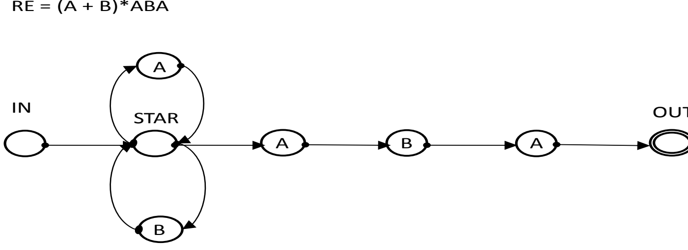

Fig. 4. Vertex labeled RE representation graph for Example 2 constructed by Alg. 1

5. *is vertex-labeled with a single symbol for each node,*
6. *each cycle of the graph involves a vertex with label $STAR$, this is the start and end node of the cycle,*
7. *each vertex v with label $STAR$ fulfills $OutDegree\left(v\right) = InDegree\left(v\right)$,*
8. *the set of vertex-labels is the set $[RE] \cup \{IN, OUT, STAR\}$,*
9. *the labels of vertices visited by an $(IN, \ldots, OUT)$ walk on $G\left(RE\right)$ set up a string generated by RE (the labels $IN, OUT, STAR$ will be ignored). Each symbol string denoted by RE can be obtained in this way.*

We say that an $(IN, \ldots, OUT)$ walk on G is a traversal on G. We denote the set of traversals on G by $T\left(G\right)$. The item (9) of Def. 3 states that for a regular expression RE $L\left(RE\right) = T\left(G\left(RE\right)\right)$.

Lemma 1. *If RE is a regular expression, then the graph $G\left(RE\right)$ generated by Alg. 1 is a schemagraph for RE.*

Proof. Starting with an empty regular expression a structural recursion by Alg. 1 gives the result. For instance, an empty RE fulfills (3) of Def. 3 and each step of Alg. 1 preserves this attribute of the graph.

Definition 4 (Schema Foundation Graph). *We say that a graph G complying with features 1-7 from Def. 3 is a schema foundation graph. We denote the set of vertex-labels for G by $Lab\left(G\right)$.*

Lemma 2. *If G is a schema foundation graph, then there exists a regular expression RE so that $G\left(RE\right) = G$ and $L\left(RE\right) = T\left(G\right)$ and $[RE] = Lab\left(G\right)$.*

5 Relational Algebra for Regular XRelation

Definition 5 (Regular XRelation for Regular Expressions). *Let RE be a regular expression and let G be a schemagraph for RE, moreover, let $w = (IN, v_1, \ldots, v_n, OUT) \in T\left(G\right)$ be a traversal on G. Let $dom_U; U \in [RE]$ be sets of data values, then $\{(v_1 : a_1, \ldots, v_n : a_n) \,|a_i \in dom_{v_i}\}$ is a tuple of type w. We say that a finite set of these tuples is a table instance of type w, and w is the type (schema) of the table instance. A regular relational instance, e.g. I, is a finite*

set of table instances. The schema of a relational instance is the set of types of its table instances. We say that the set of these tuple types for all $w \in T(G)$ compose the schema of a regular XRelation based on RE. We denote this regular XRelation by $XR(RE)$, so I is an instance of $XR(RE)$.

It is well known that the class of regular languages is closed under union, intersection and complement. It follows that regular XRelations possess these closure properties. That is, the set operations of relational databases are applicable for XRelations.

Let RE_1 and RE_2 be regular expressions and let I_1 and I_2 be regular relational instances for the regular XRelations $XR(RE_1)$ and $XR(RE_2)$, respectively.

Union. The union of the schemagraphs $G(RE_1) \cup G(RE_2)$ is a schemagraph too ($= G(RE_1 + RE_2)$). That is, the union $XR(RE_1) \cup XR(RE_2)$ of regular XRelations is again an XRelation and its regular instances have the form $I_1 \cup I_2$.

Intersection. The intersection $XR(RE_1) \cap XR(RE_2)$ of regular XRelations is again an XRelation and its regular instances have the form $I_1 \cap I_2$.

Difference. The set difference of two regular instances I_1 and I_2 for the regular XRelation $XR(RE_1)$ is also a regular instance for it.

5.1 Projection

Definition 6 (Node-selection). *Let RE be a regular expression and let $G(V, E)$ be a schemagraph for RE. We say that a subset $X \subset V$ is a node-selection over G iff $IN \in X$ and $OUT \in X$. If X is a node-selection, then we denote by $\overline{X}$ the complementer node-selection for X, defined by $\overline{X} = V \setminus X \cup \{IN, OUT\}$.*

Remark 1. Definition 6 presents a rigid method for fixing the scope of the projecting window. If the selected nodes belong to a cycle, then the selection chooses all occurrences from a given transversal. A more flexible selection method can be realized on an extended graph. We may add a a given number of walks (as new nodes and edges) for any (or all) cycles and select nodes on the new graph. E.g., if the RE involves the (sub)expression $(ABC)^*$, the original graph contains the nodes a, b, c (labeled with A, B, C, respectively), and the edges $(a, b), (b, c), (c, a)$. The node-selection of $\{a, b\}$ selects the labels $ABAB$ from the traversal which repeats twice the cycle. The extended graph (with two cycles) would give the new nodes and edges

$(a_1, b_1), (b_1, c_1), (c_1, a_2)$
$(a_2, b_2), (b_2, c_2)$

The labels on vertices are $ABCABC$. We can select, for instance, the nodes a_1, b_1, a_2 which brings ABA. No selection on the original graph can produce this result.

Definition 7 (Projection). *Let $G(V,E)$ be a schemagraph and let X be a node-selection over G. Let $E[X] = \{(a_1, a_n) | a_1, a_n \in X; a_2, \ldots, a_{n-1} \notin X\}$, $(a_1, a_2, \ldots, a_{n-1}, a_n) \in P(G)$, where $P(G)$ is the set of paths for G.*

We say that $G[X] = (V \setminus X \cup \{IN, OUT\}, E[X])$ is the projection graph of G onto X.

Lemma 3. *If G is a schemagraph for the regular expression RE and X a node-selection over G, then $G[X]$ is a schema foundation graph.*

Proof. $G[X]$ is the result of deleting the complement of the subgraph X from the schemagraph G and re-connecting during the deletion disconnected vertices of X. Clearly, the features 1-7 from Def. 3 of the schemagraph will be preserved. For instance, a traversal (an $(IN, \ldots, OUT)$ walk) on G will be either deleted (it contains no vertex from X) or preserved (perhaps reconnected), so the attribute (3) of Def. 3 will be preserved.

Let $RE[X]$ be a regular expression complying with the schema foundation graph $G[X]$, then we say that $RE[X]$ is the projection of RE onto X. ($Lab(X) \subseteq [RE]$, but different vertices in X can have the same label).

Definition 8 (Projection of Schema). *Let G be a schemagraph of the regular expression RE and let X be a node-selection over G.*
Let $w = (v_0, v_1, \ldots, v_n, v_{n+1}); v_0 = IN, v_{n+1} = OUT$ be a traversal on G (w is a type for RE). We denote by $w[X]$ the projection of w to X, defined as follows: $w[X] = (v_0, v_{i_1}, \ldots, v_{i_k}, v_{n+1}); v_r \in X$ for $r \in \{i_1, \ldots, i_k\}$ and $v_r \notin X$ otherwise.

$w[X]$ is either a traversal on G or its re-connected edges belong to $G[X]$, so:

Lemma 4. *If G is a schemagraph for the regular expression RE and X a node-selection over G and w is schema for RE, then $w[X]$ is a traversal on $G[X]$.*

Definition 9 (Projection of Instance). *Let RE be a regular expression and let XR be a regular XRelation based on RE and let I be a table instance for XR with $type(I) = w$. The projection of I to X, denoted by $\pi_X(I)$, is the set of tuples $\{t[X] | t \in I; type(t[X]) = w[X]\}$ (that is, $t[X]$ is the subsequence of constants from t according to the subsequence $w[X]$ in w).*

Definition 10 (Functional Dependency). *Let G be a schemagraph of the regular expression RE and let X, Y be node-selections over G. The regular relational instance I satisfies the functional dependency (XRFD) $X \to Y$ if for any to tuples $t_1, t_2 \in I$ with $type(t_1) = w_1$ and $type(t_2) = w_2$, whenever $w_1[X] = w_2[X]$ and $t_1[X] = t_2[X]$, then $w_1[Y] = w_2[Y]$ and $t_1[Y] = t_2[Y]$.*

Example 3. Let $\mathbb{R} = (R_1, \ldots, R_n)$ be a relational database schema. The regular expression $RE = (R_1 | R_2 | \ldots | R_n)$ (if $R_i = (a, b, c, d, e)$ then for the regular expression we use the concatenation *abcde* of the attributes), then the schemagraph for RE consists of parallel, linear $(IN, \ldots, OUT)$ traversals. Each relational functional dependency over $\mathbb{R}$ can be defined on the schemagraph using Def. 10, with the restriction that both participant node-selections will be located on the same $(IN, \ldots, OUT)$ path.

5.2 Natural Join

Definition 11 (Disjunctive Natural Join). *Let G_1, G_2 be schemagraphs for the regular expressions RE_1, RE_2 and let X_1, X_2 be node-selections over G_1, G_2, respectively, so that $G_1[X_1] = G_2[X_2] = G$. Let $w_1 \in T(G_1)$ and $w_2 \in T(G_2)$ so that $w_1[X_1] = w_2[X_2] = w$. Let $(A, B) \in G$, then $(A, x, B) \in G_1$ and $(A, y, B) \in G_2$ for some paths x and y, respectively, so that AxB and AyB are subsequences of w_1 and w_2, respectively. Let I_1 and I_2 be table instances for the regular XRelations $XR(RE_1)$ and $XR(RE_2)$, respectively, so that $type(I_1) = w_1$ and $type(I_2) = w_2$, then we say that w_1 and w_2 (and also I_1 and I_2) can be joined. We define $I = I_1 \bowtie I_2$ as a (disjunctive joined) regular relational instance, for which if $t \in I$, then there exist $t_1 \in I_1$,$t_2 \in I_2$ so that $t_1[X_1] = t_2[X_2]$, then $t[u] = t_1[u] | u \in (IN, \ldots, A)$ and $t[u] = t_2[u] | u \in (B, \ldots, OUT)$. Moreover, let $t[A \bowtie B] = \{t[ApB] | ApB \in P(G_1) \cup P(G_2)\}$, then $t[A \bowtie B] = \{t_1[AxB] \cup t_2[AyB] | t_1[A] = t_2[A], t_1[B] = t_2[B]\}$.*

Remark 2. If $w_1[X_1] = w_2[X_2] = \triangleright\triangleleft$, then the disjunctive join of the two table instances I_1 and I_2 will be $I = I_1 \bowtie I_2 = I_1 \cup I_2$, moreover, $schema(I) = \{w_1, w_2\}$. The same is true for the special case $[X_1] = [X_2] = \emptyset$ as well.

Remark 3. We have defined the join operator for two table instances joined on two single attributes. We can extend this definition to joining two table instances on any number (or a single one) of attributes. We can also extend this definition to joining any (finite) number of table instances in a natural way.

Example 4. The disjunctive natural join of table instances means in fact union for the background regular expressions. Let $RE_1 = A \circ X \circ Y \circ B$ and $RE_2 = A \circ W \circ Z \circ B$ and let $X_1 = X_2 = \{A, B\}$, then the regular expression complying with the on A, B joined instances will be $A \circ ((X \circ Y) + (W \circ Z)) \circ B$.

Example 5. The XML documents on Fig. 5 conform to the DTD element declarations
Courses1:

```
<!ELEMENT course (Cid,Cname,(Instid,Instn)+)>
```

and
Courses2:

```
<!ELEMENT course (Cid,(Stid,Stn)+)>
```

respectively. The disjunctive join of the two instances results in

JoinedCourses:

```
<!ELEMENT course (Cid,((Cname,(Instid,Instn)+))|((Stid,Stn)+))>
```

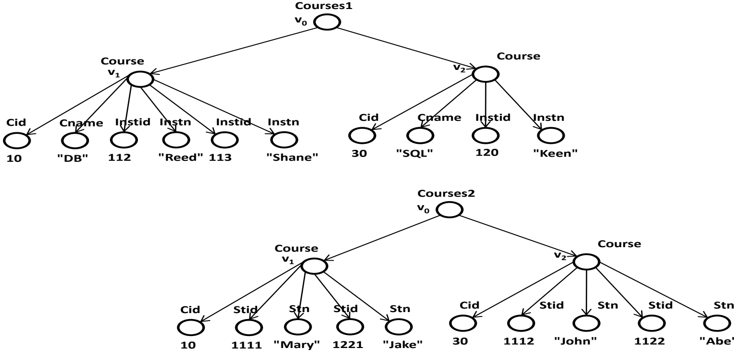

Fig. 5. Example XML documents for natural join

Definition 12 (Concatenative Natural Join). *The concatenative natural join will be defined similarly to its disjunctive counterpart, using concatenation instead of disjunction. That is, we define the regular relational instance* $I = I_1 \bowtie I_2$ *as two (concatenative) joined table instances. If* $t \in I$*, then there exist* $t_1 \in I_1$*,*$t_2 \in I_2$ *so that* $t_1[X_1] = t_2[X_2]$ *and* $t[AxyB] = t_1 \bowtie t_2[AxyB] = \{t_1[Ax] \circ t_2[yB] | t_1[A] = t_2[A], t_1[B] = t_2[B]\}$*. For the special case see Rem. 2.*

Definition 13 (Natural Join of Regular Instances). *The natural (disjunctive or concatenative) join for two regular relational instances will be defined as the set of joined member table instances. That is, if* I_1 *and* I_2 *are regular relational instances, then* $I_1 \bowtie I_2 = \{J | J = J_1 \bowtie J_2; J_1 \in I_1, J_2 \in I_2\}$*.*

Remark 4. We have defined the join operator for two regular relational instances. We can extend this definition to joining any (finite) number of regular relational instances in a natural way.

Remark 5. If $w_1[X_1] = w_2[X_2] = \triangleright\triangleleft$, then the concatenative join of the two table instances I_1 and I_2 will be $I = I_1 \bowtie I_2 = I_1 \circ I_2$, moreover, $schema(I) = (w_1 \circ w_2)$.

Example 6. The concatenative natural join of table instances means in fact concatenation for the background regular expressions. Let $RE_1 = A \circ X \circ Y \circ B$ and $RE_2 = A \circ W \circ Z \circ B$ and let $X_1 = X_2 = \{A, B\}$, then the regular expression complying with the on A, B joined instances will be $A \circ ((X \circ Y) \circ (W \circ Z)) \circ B$.

Example 7. The concatenative join of the two instances realized in the XML documents on Fig. 5 will be

JoinedCourses:

```
<!ELEMENT course (Cid,((Cname,(Instid,Instn)+)),((Stid,Stn)+))>
```

5.3 Join Dependencies, Implication Problems for Xrelations

Definition 14 (Join Dependency). *Let $G(V,E)$ be a schemagraph of the regular expression RE and let X, Y be node-selections over G so that $V = X \cup Y \cup \{IN, OUT\}$. Let I be an instance for the XRelation over RE and $\pi_X(I), \pi_Y(I)$ the projections of I to X and Y, respectively. We say that an instance I for the XRelation over RE satisfies the $\bowtie [X,Y]$ join dependency iff $I = \pi_X(I) \bowtie \pi_Y(I)$.*

Using the Definitions of functional and join dependency we can define normal form for XRelation schemas (BCNF, 4NF etc.) and describe the lossless decomposition for regular XRelations.

Definition 15 (Lossless Decomposition). *Let $G(V,E)$ be a schemagraph of the regular expression RE and let $X_1, \ldots, X_n$ node-selections with $\cup_{i=1}^{n} X_i \cup \{IN, OUT\} = V$. The set $X_1, \ldots, X_n$ is a lossless decomposition of G if any regular relational instance I for the XRelation over RE satisfies the join dependency $I = \pi_{X_1}(I) \bowtie \pi_{X_2}(I) \bowtie \ldots \bowtie \pi_{X_n}(I)$.*

The logical implication of functional and join dependencies for XRelations is decidable with a special form of the Chase algorithm. We present here an algorithm to decide logical implication of functional dependencies for XRelations.

Definition 16. *Let G be a schemagraph of the regular expression RE. Let Σ be a set of XRFDs and let $X \to Y$ be an XRFD over G, then Σ implies $X \to Y$ (denoted by $\Sigma \models X \to Y$) if for each (finite) regular relational instance I that satisfies Σ $I \models X \to Y$ will also be fulfilled.*

Algorithm 2. *Algorithm for checking implication of XRFDs.*
Input: schemagraph $G = (V,E)$ for an XRelation, a set Σ and $\sigma : X \to Y$ functional dependencies over G
Output: true, if $\Sigma \models \sigma$, false otherwise

1. Initialization
Create a counter example from two copies of G (G_1, G_2), the nodes of X colored green on both copies, the nodes of Y colored red on one copy and yellow on the other one.
2. $FDSET := \Sigma$;
3. $greene := X$;
4. repeat until no more dependency is applicable:
if $W \to Z \in FDSET$ and $W \subseteq greene$, then
i. $FDSET := FDSET - (W \to Z)$;
ii. $greene := greene \cup Z$;
iii. for all $v \in Z$ set color(v) := green (on both copies)

5. if the number of yellow nodes and red nodes are both zero, then output is true otherwise output is false.

Proposition 1 (Functional Dependency Implication). *Let G be a schema-graph of the regular expression RE. Let Σ be a set of XRFDs and let $X \to Y$ be an XRFD over G, then $\Sigma \models X \to Y$ if and only if the Alg. 2 with input G, Σ and $X \to Y$ returns true.*

6 Conclusion and Future Work

This paper presents regular expressions as compact database schemas and defines functional and join dependencies over them, based on the graph representation for the regular expressions. We defined extended relations on the graph representation for regular expressions and determined semantics for the dependencies on instances of extended relations. The logical implication of this kind of functional dependencies is decidable in quadratic time.

Our model offers the tools for a normal form of XRelation. We think that the logical implication for the join dependency, defined here, is decidable similarly to Alg. 2.

We would like to find the connection between our model and data words, that is, to define a register automaton that accepts those data words that satisfy a given functional dependency specified for the corresponding XRelation.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
2. Arenas, M., Libkin, L.: A normal form for XML documents. ACM Transactions on Database Systems 29(1), 195–232 (2004)
3. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoretical Computer Science 48(3), 117–126 (1986)
4. Bouyer, P., Petit, A., Thrien, D.: An algebraic approach to data languages and timed languages. Information and Computation 182(2), 137–162 (2003)
5. Brzozowski, J.A.: Derivatives of regular expressions. Journal of the ACM 11(4), 481–494 (1964)
6. Champarnaud, J.-M., Ziadi, D.: Canonical derivatives, partial derivatives and finite automaton constructions. Theoretical Computer Science 289(1), 137–163 (2002)
7. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys 16, 1–53 (1961)
8. Kaminski, M., Francez, N.: Finite-memory automata. Theoretical Computer Science 134(2), 329–363 (1994)
9. Libkin, L., Vrgoč, D.: Regular expressions for data words. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 274–288. Springer, Heidelberg (2012)
10. Murata, M., Lee, D., Mani, M., Kawaguchi, K.: Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology 5(4), 660–704 (2005)
11. Nicaud, C., Pivoteau, C., Razet, B.: Average Analysis of Glushkov Automata under a BST-Like Model. In: Proc. FSTTCS, pp. 388–399 (2010)
12. Sperberg-McQueen, C.M., Thompson, H.: XML Schema. Technical report, World Wide Web Consortium (2005), http://www.w3.org/XML/Schema

13. Szabó, G. I., Benczúr, A.: Functional Dependencies on Extended Relations Defined by Regular Languages. Annals of Mathematics and Artificial Intelligence (2013), doi: 10.1007/s10472-013-9352-z
14. Vincent, M.W., Liu, J., Liu, C.: Strong functional dependencies and their application to normal forms in XML. ACM Transactions on Database Systems 29(3), 445–462 (2004)
15. Wang, J., Topor, R.W.: Removing XML Data Redundancies Using Functional and Equality-Generating Dependencies. In: Proc. ADC, pp. 65–74 (2005)
16. Watson, B.W.: A taxonomy of finite automata construction algorithms. Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands (1994)