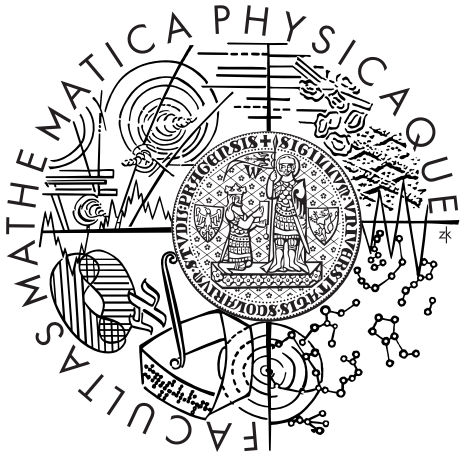
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Michal Švirec

Efficient Detection of XML Integrity Constraints

Department of Software Engineering

Supervisor of the master thesis: RNDr. Irena Mlýnková, Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague, 2011

I would like to thank my supervisor RNDr. Irena Mlýnková, Ph.D., for her helpful advices, corrections and suggestions.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... Signature

Název práce: Efektívna detekcia integritných obmedzení v XML

Autor: Michal Švirec

Katedra: Katedra softwarového inženýrství

Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.

Abstrakt: Znalosť integritných obmedzení v XML dátach je jeden z dôležitých aspektov ich spracovania. Avšak aj keď tieto integritné obmedzenia pre dané dáta poznáme, je častým javom, že dané dáta sú voči ním nekonzistentné. Z tohto dôvodu vznikla snaha detekovať tieto nekonzistentosti dát a následne ich opravovať. Táto práca rozširuje a zdokonaľuje doterajšie prístupy opráv XML dokumentov porušujúcich definované integritné obmedzenia, konkrétne takzvané funkčné závislosti. Práca prináša algoritmus začleňujúci váhový model a taktiež zapája užívateľa do procesu hľadania a následného aplikovania vhodnej opravy nekonzistentných XML dokumentov. Súčasťou práce sú experimentálne výsledky.

Klíčová slova: XML, funkčná závislosť, porušenie funkčných závislostí, oprava porušení

Title: Efficient Detection of XML Integrity Constraints

Author: Michal Švirec

Department: Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.

Abstract: Knowledge of integrity constraints covered in XML data is an important aspect of efficient data processing. However, although integrity constraints are defined for the given data, it is a common phenomenon that data violate the predefined set of constraints. Therefore detection of these inconsistencies and consecutive repair has emerged. This work extends and refines recent approaches to repairing XML documents violating defined set of integrity constraints, specifically so-called functional dependencies. The work proposes the repair algorithm incorporating the weight model and also involve a user into the process of detection and subsequent application of appropriate repair of inconsistent XML documents. Experimental results are part of the work.

Keywords: XML, functional dependency, functional dependencies violations, violations repair

Contents

Chapter 1

Introduction

Nowadays, one of the most largely used standards for information representation and data exchange on the Internet is the eXtensible Markup Language (XML) [1]. While XML as a markup language provides syntactic flexibility, the structure of an XML document is described with a so-called XML schema language. The two most popular XML schema languages proposed by W3C[1] are Document Type Definition (DTD) [1] and XML Schema Definition (XSD) [2, 3, 4]. They support some kind of semantic content (e.g., keys and foreign keys), but for improvement of semantic expressiveness integrity constraints for XML [5] have been defined.

However, similarly to the problem of schemas, also the problem of detection of integrity constrains has two distinct aspects. First, if integrity constraints are not explicitly expressed, they need to be detected from the given set of data. Second, if integrity constraints are expressed, XML document may not be consistent with respect to them and this needs to be detected (checking the satisfaction of integrity constraint in an XML document). Moreover, the detected XML data inconsistency needs to be repaired. In this thesis we choose the latter aspect of detection of integrity constraints along with the repair of the inconsistent XML document. Since several different classes of integrity constraints have been defined for XML, we choose so-called functional dependency (FD) [6] which is the most common semantic constraint used in relational databases.

The problem of checking the satisfaction of functional dependencies in an

[1]The World Wide Web Consorcium, `http://www.w3.org`

XML document is studied in [7]. Also algorithms computing a repair applied to an XML document such that functional dependencies will be satisfied again have been proposed [8, 9, 10].

1.1 Aim of the Thesis

The aim of this thesis is to propose an algorithm repairing XML functional dependency violations for a given XML document and a set of functional dependencies. It analyses recent approaches and discusses their advantages and disadvantages. The focus of the thesis is to incorporate a weight model, to involve the user the process of finding and applying the repair and to find out how this interaction helps. An important part of the thesis is an experimental implementation of the proposed algorithm and its experimental evaluation.

1.2 Structure of the Thesis

Chapter 2 introduces basic definitions which are necessary for further chapters. Chapter 3 presents recent approaches of repairing XML functional dependency violations. In Chapter 4 the main algorithm is proposed. Chapter 5 contains details of the experimental implementation. Experimental result are presented in Chapter 6. Finally, Chapter 7 contains a conclusion and several suggestions for future work.

Chapter 2

Basic Definitions

In this chapter basic definitions necessary in the following chapters are placed. The definitions are taken from [8, 9, 10].

2.1 XML Trees and DTDs

Definition 2.1 (Tree)**.** Being given an alphabet of nodes $\mathbb{N}$ and an alphabet of node labels Σ, a *tree* T over $\mathbb{N}$ and Σ is a tuple $(r_T, N_T, E_T, \lambda_T)$, where $N_T \subseteq \mathbb{N}$ is the set of nodes, $\lambda_T : N_T \to \Sigma$ is a node labelling function, $r_T \in N_T$ is the distinguished root of T, and $E_T \subseteq N_T \times N_T$ is a set of edges such that starting from any node $n_i \in N_T$ it is possible to reach any other node $n_j \in N_T$, walking through a sequence of edges $e_1, \ldots, e_k$ which are connected and acyclic. □

Let us also denote the set of leaf nodes as $Leaves(T)$ and the set of trees defined over an alphabet of node labels Σ as T_Σ.

Definition 2.2 (XML Tree)**.** *XML tree* is a pair $XT = \langle T, \delta \rangle$, where:

i) $T = (r, N, E, \lambda)$ is a tree from $T_{\tau \cup \alpha \cup \{S\}}$, where τ is a tag alphabet, α is an attribute name alphabet and S is a symbol not belonging to $\tau \cup \alpha$ (representing `#PCDATA` content of elements);

ii) given a node n of T, $\lambda(n) \in \alpha \cup \{S\} \Leftrightarrow n \in Leaves(T)$;

iii) $\delta : Leaves(T) \rightarrow Str$ where Str is a string alphabet is a function associating a (string) value to every leaf of T.

□

Example 1. Consider the following XML document representing a collection of books. Its graphical representation as an XML tree is in Fig. 2.1.

```
<bib>
  <book>
    <written_by>
      <author ano="A1">
        <name>John Writer</name>
      </author>
      <author ano="A1">
        <name>Eric Seller</name>
      </author>
    </written_by>
    <title>Some title</title>
  </book>
  <book>
    <written_by>
      <author ano="A2">
        <name>Adam Publisher</name>
      </author>
    </written_by>
    <title>Some title 2</title>
  </book>
</bib>
```

The nodes of an XML tree have a label denoting the tag name of the element and unique element identifier in brackets. Leaf nodes are either an attribute or the textual content of an element. The label of a textual node contains string contained inside of element and unique element identifier. The label of an attribute node contains in addition to textual node the name of the attribute. □

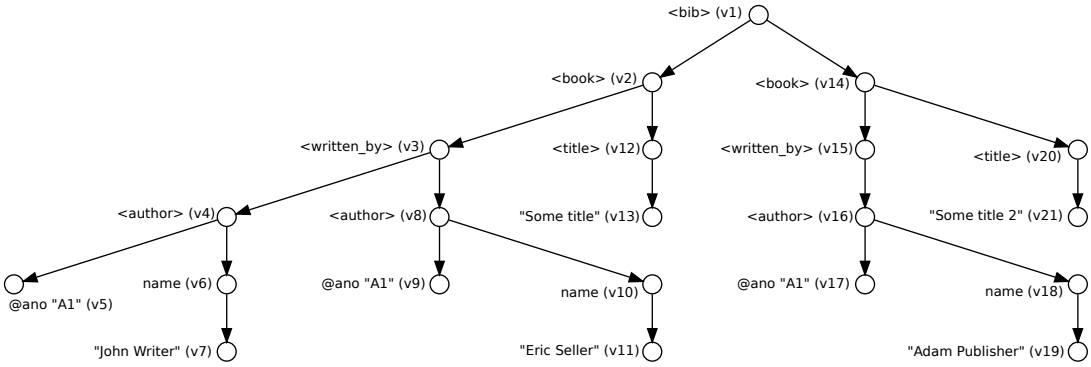


Figure 2.1: An XML Tree

Definition 2.3 (DTD)**.** *DTD* is a tuple $D = (\tau, \alpha, P, R, rt)$ where:

i) τ and α are of the same definition as in the *XML tree*

ii) P is the set of *element type definitions*

iii) R is the set of *attribute lists*

iv) $rt \in \tau$ is the tag of the document root element.

□

Definition 2.4 (Path Expression)**.** A *path expression* is an expression of the form $p = ('/'|'//')s_1 \ldots ('/'|'//')s_m$ where $s_1, \ldots, s_{m-1} \in \tau$, and $s_m \in \tau \cup \alpha \cup \{S\}$. □

Definition 2.5 (Path)**.** *Path* p on a DTD $D = (\tau, \alpha, P, R, rt)$ is a sequence $p = s_1, \ldots, s_m$ of symbols in $\tau \cup \alpha \cup \{S\}$ such that:

i) $s_1 = rt$;

ii) for each i in $2..m-1$, $s_i \in \tau$ and s_i appears in the element type definition of s_{i-1};

iii) $s_m \in \alpha \Rightarrow s_m$ appears in the attribute list of s_{m-1};

iv) $s_m \in \tau \cup \{S\} \Rightarrow s_m$ appears in the element type definition of s_{m-1}.

□

As we have defined a *path* on a DTD D, let us denote $paths(D)$ as the set of all paths which can be defined on a DTD D. Also, another important notion is $p(XT)$ (or $\{[\![p]\!]\}$), which is the set of nodes from XML tree XT conforming to DTD D, which can be reached by the path $p \in paths(D)$, starting from the root of XT. The set of nodes reachable from a node v following path p is denoted as $\{v[\![p]\!]\}$. When there is only one node in $\{v[\![p]\!]\}$, we use $v[\![p]\!]$ to denote this node. Moreover, let $XT.p$ denote the *answer* of the path p applied on XT that is:

- if $p \in EPath(D)$, where $EPath(D)$ denotes the set of the paths whose last symbol denotes an element, then $XT.p = p(XT)$
- if $p \in StrPath(D)$, where $StrPath(D)$ denotes the set of paths whose last symbol denotes either the textual content of an element or an attribute, then $XT.p = \{\delta_T(x) | x \in p(XT)\}$.

Example 2. Consider the XML tree XT from Example 1 conforming the DTD D defined below, which is representing a collection of books.

```
<!ELEMENT bib (book+)>
<!ELEMENT book (written_by, title)>
<!ELEMENT written_by (author+)>
<!ELEMENT author (name)>
<!ATTLIST author ano CDATA>
<!ELEMENT name PCDATA>
<!ELEMENT title PCDATA>
```

The set $paths(D)$ contains the following paths:

$$\begin{aligned} paths(D) = \{&/bib, /bib/book, /bib/book/written\_by, \\ &/bib/book/written\_by/author, \\ &/bib/book/written\_by/author/name, \\ &/bib/book/written\_by/author/name/S, \\ &/bib/book/written\_by/author/@ano, \\ &/bib/book/title, /bib/book/title/S\} \end{aligned}$$

□

2.2 Integrity Constraints

Before defining of the XML integrity constraint, let us introduce some notation used in the definition. A *path atom* is an expression of the form $[x_1]p[x_2]$, where p is a path expression, x_1 and x_2 are terms, and $x_1 \notin Str$.
A conjunction of path and built-in atoms $C = [X_1]p_1[Y_1] \cap \cdots \cap [X_n]p_n[Y_n] \cap U_1\theta_1V_1 \cap \cdots \cap U_k\theta_kV_k$ is said to be *safe* if all variables in C are *range restricted*, i.e. if

- for every $[X_i]p_i[Y_i]$, either X_i is a constant (node identifier of a string), or there is some $[X_j]p_j[Y_j]$ in C where X_j is range restricted;
- for every built-in term $U_i\theta_iV_i$ occurring in C, if θ_i is equal to " = " then at least one of the two terms is range restricted; otherwise both U_i and V_i must be range restricted.

A rootless tree formula is an expression of the form $p(\Phi_1 \wedge \cdots \wedge \Phi_k)$ where Φ_i is a rootless path formula (expression of the form $p[y]$ where p is a path expression and y is a term) or a rootless tree formula and p is a path expression. A tree atom is an expression of the form $[x]T$ where T is a rootless tree formula and x is a term.

Definition 2.6 (XML Integrity Constraint)**.** An *XML constraint* is a formula of the form:

$$(\forall X)[\Phi(X) \supset (\exists Y_1)\Psi_1(X, Y_1) \vee \cdots \vee (\exists Y_k)(X, Y_k)]$$

where $X, Y_1, \ldots, Y_k$ denote distinct sets of universally and existentially quantified variables, $\Phi(X)$ and $\Phi(X) \wedge \Psi_i(X, Y_i)(\forall i \in [1..k])$ are safe conjunctions of built-in and tree atoms. □

Example 3. Consider the XML tree XT from Example 1. Thereafter the integrity constraint that there must exist at least two books differents titles is expressed as

$$\forall(X)[[root]/bib[X] \supset \exists(Z_1, Z_2)([X]/book/title/S[Z_1] \wedge [X]/book/title/S[Z_2] \wedge Z_1 \neq Z_2)]$$

2.3 Functional Dependencies

Since different approaches uses slightly different representation of an XML document, also the definition of functional dependencies differs, too. This is the reason why we introduce two different definitions in this section.

2.3.1 Functional Dependency 1

In a relational database, a correspondence between values A and B in the tuple of D models the functional dependency denoted as $A \rightarrow B$. Since in XML there is no such standard tuple concept, the concept of XML tree tuples is introduced, corresponding to the concept of tuples in relational databases.

Definition 2.7 (Tree Tuple)**.** Being given an XML tree XT conforming to DTD D, a *tree tuple* t of XT is a maximal sub-tree of XT such that for every path $p \in paths(D)$, $t.p$ contains at most one element. □

Example 4. Consider the XML tree XT in Fig. 2.1. The subtrees of XT shown in Fig. 2.2 are tree tuples, and the subrees in Fig. 2.3 are not.

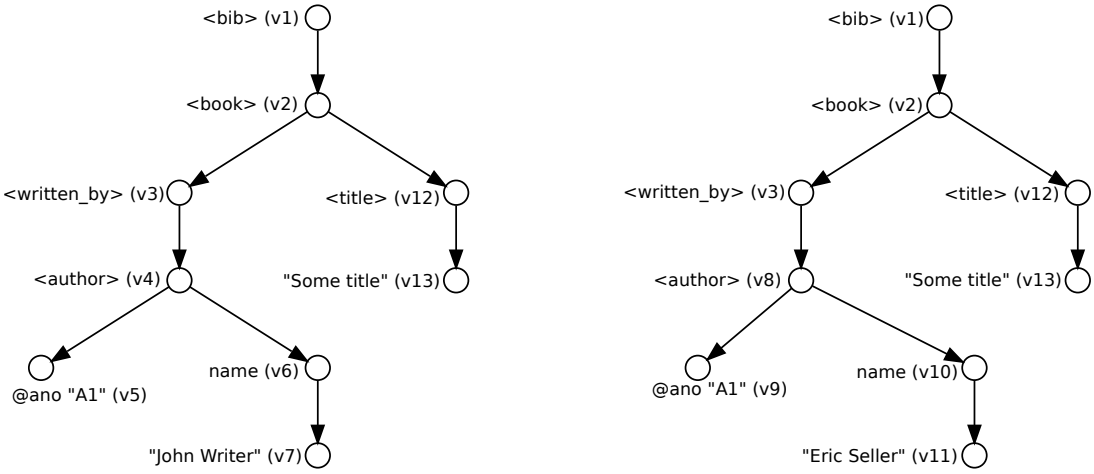


Figure 2.2: Two tree tuples of the XML tree in Fig. 2.1

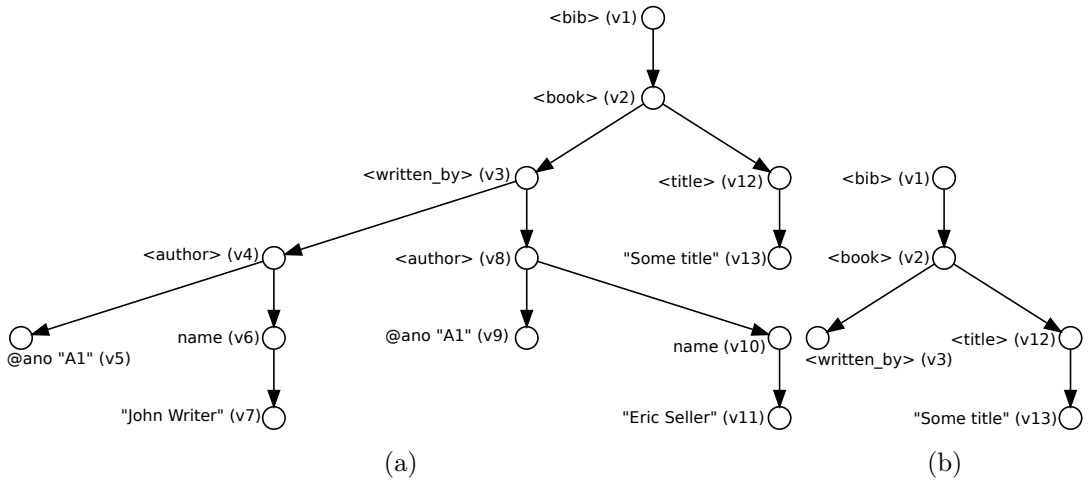


Figure 2.3: Two subtrees of the XML tree in Fig. 2.1 which are not tree tuples

In Fig. 2.3a, the subtree is not a tree tuple, because the answer of the path */bib/book/written_by/author* contains two distinct nodes (i.e. $v4$ and $v8$). The subtree in the Fig. 2.3b is not a tree tuple, because it is not a maximal subtree (it is a subtree of a tuple from Fig. 2.2). □

Definition 2.8 (Functional Dependency)**.** Being given a DTD D, a *functional dependency* on D is an expression of the form $S \rightarrow p$, where S is a finite non empty subset of $paths(D)$ and $p \in paths(D)$. □

Example 5. Consider the XML tree in Fig 2.1. The following functional dependency expresses the constraint that two distinct authors of the same book cannot have the same value of attribute `ano`:

$$\{/bib/book, /bib/book/written\_by/author/@ano\} \rightarrow /bib/book/written\_by/author$$

□

2.3.2 Functional Dependency 2

Definition 2.9 (Functional Dependency)**.** With a given DTD D, a *functional dependency* is of the form $\sigma = (P, P', (P_1, \ldots, P_n \rightarrow P_{n+1}))$. Here P is a root path (path where the first element is a root element of an XML document), or

$P = \epsilon$ (empty path). Each $P_i (i \in [1, n])$ is a singleton leaf path, and there is a no non-empty common prefix for $P_1, \ldots, P_{n+1}$. Being given an XML document T conforming to D, we say T satisfies σ:iff $\forall v \in \{[\![P]\!]\}, \forall v_1, v_2 \in \{v[\![P']\!]\}$, if $v_1[\![P_i]\!] \equiv v_2[\![P_i]\!]$ for all $i \in [1, n]$, then $v_1[\![P_{n+1}]\!] \equiv v_2[\![p_{n+1}]\!]$. □

The main difference between two definitions of the functional dependency is that the former definition can use any path from $paths(D)$, whereas the latter considers that each $P_i (i \in [1, n])$ is from $StrPaths(D)$. That means the constraint from Example 5 cannot be expressed by functional dependency defined in 2.9, because of $/bib/book/written\_by/author$ path. Nevertheless, modified constraint from Example 5 expressed by FD defined in 2.9 is shown in Example 6.

Example 6. Consider the constraint C saying that two distinct authors with different names of the same book cannot have the same value of attribute `ano`. The functional dependency expressing C is defined as follows:

$$(bib/book, written\_by/author, (@ano \rightarrow name))$$

□

Chapter 3

Analysis of Recent Approaches

This chapter introduces the description and categorization of recent approaches to repairing XML documents that violate functional dependencies defined in these documents.

All approaches dealing with the problem of finding optimal repair can be divided into categories according to the usage of elementary repair primitives. This repair primitives are: inserting node, deleting node, updating node and marking node as unreliable in XML document.

3.1 Repairs and Consistent Answers for XML Data with Functional Dependencies

A technique for computing repairs which solves the problem of XML data inconsistency with respect to a set of functional dependencies was proposed in [8]. In this approach, the authors are trying to find a minimal set of update operations which makes XML data consistent. These update operations can be divided into two categories i) replacing a value associated with an element or an attribute, and ii) marking a particular node information as unreliable.

3.1.1 XML Tree and Functional Dependency

To be able to resolve the problem of functional dependency violations in XML document, the authors try to introduce the concept of functional dependencies based on those defined for relational databases. The concept of a tree tuple and functional dependency was introduced in Definition 2.7 and 2.8.

Being given an XML tree XT conforming a DTD D and a functional dependency $F : S_1 \rightarrow S_2$, we say that XT satisfies $F (XT \models F)$ if for each pair of tree tuples t_1, t_2 of XT,

$$t_1.S_1 = t_2.S_1 \wedge t_1.S_1 = \emptyset \Rightarrow t_1.S_2 = t_2.S_2$$

Being given a set of functional dependencies $\mathcal{FD} = \{F_1, \ldots, F_n\}$ over D, we say that XT satisfies $\mathcal{FD}$ if it satisfies F_i for every $i \in 1..n$.

3.1.2 Repairing inconsistent XML data

The authors of this approach choose two kinds of actions to repair inconsistent XML data with regard to functional dependencies. The first action is updating the value of an attribute or the content of an element. As the second action the authors choose marking inconsistent element as “unreliable" rather than deleting it, because removing elements from an XML document leads to some undesired drawbacks: it does not always suffice to remove inconsistency and deleting a node can lead to a new document not valid against the original schema.

Depending on XML data and defined functional dependencies, each inconsistency could have many possible strategies to repair it. From all the possible repair strategies the authors prefer those for which smaller changes are made to the original document.

Example 7. Consider the XML tree XT conforming the DTD D from Example 2, which is representing a collection of books and the following functional dependency:
$\{bib.book, bib.book.written\_by.author.@ano\} \rightarrow bib.book.written\_by.author$ (@ denotes the attribute of an element).

The functional dependency defined above requires that for each book there is only one element author having a given @*ano* value. Therefore XT does not satisfy the given functional dependency, because two author elements of the same book have the same value of attribute @*ano*. To resolve this data inconsistency, we can use two different repair strategies: 1) changing one of the value of attribute @*ano*; 2) marking one of the elements author as unreliable. Since the first strategy changes only attribute @*ano*, it is preferred to the second strategy, which changes a larger portion of document, since it marks a whole author element as unreliable. □

3.1.3 Repair Algorithm

Before introducing the algorithm to repair an inconsistent XML document, let us define the notion of reliability of elements in an XML tree:

Definition 3.1 (R-XML Tree)**.** A R-XML *tree* is a triplet $RXT = \langle T, \delta, \varrho \rangle$, where $\langle T, \delta \rangle$ is an XML tree and ϱ is a reliability function from N_T to `{true, false}`, such that for each pair of nodes $n_1, n_2 \in N_T$ with n_2 descendant of n_1, it holds that $\varrho(n_1) = false \Rightarrow \varrho(n_2) = false$. □

To be able to create a repair, R-XML Tree must not satisfy FD according to definition of weak satisfiability:

Definition 3.2 (Weak satisfiability)**.** Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming a DTD D, and $f : S \rightarrow p$ be a functional dependency. We say that RXT *weakly satisfies* f ($RXT \models_w f$) if one of the following conditions holds:

1. $\langle T, \delta \rangle \models f$;
2. for each pair of tuples t_1, t_2 of RXT one of the following holds:
   (a) there exists a path $p_i \in S$ such that:
   $(\varrho(p_i(t_1)) = false) \vee (\varrho(p_i(t_2)) = false)$;
   (b) $(\varrho(p(t_1)) = false) \vee (\varrho(p(t_2)) = false)$. □

The repair of an R-XML tree which does not satisfy $\mathcal{FD}$ set of functional dependencies is a pair of functions δ' and ϱ' such that RXT' tree composed of the original tree and the repair ($RXT' = \langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle$) weakly satisfies FD ($RXT' \models_w \mathcal{FD}$). The composition of the R-XML tree and the repair (i.e. composition of their δ and ϱ functions) is defined as follows:

Definition 3.3 (Composition of δ functions)**.** The *composition* of two functions δ_1 and δ_2 asssociating values to leaf nodes is

$$\delta_1 \cdot \delta_2(n) = \begin{cases} \delta_1(n) & \text{if } \delta_1(n) \text{ is defined over } n, \\ \delta_2(n) & \text{otherwise (i.e. } \delta_1(n) \text{ is not defined over } n). \end{cases}$$

□

Definition 3.4 (Composition of ϱ functions)**.** The *composition* of two reliability functions ϱ_1 and ϱ_2 asssociating a boolean value to nodes is

$$\varrho_1 \cdot \varrho_2(n) = \varrho_1(n) \ AND \ \varrho_2(n).$$

□

With a repair $\langle \delta, \varrho \rangle$ of R-XML tree and a set of labelled nodes N of this tree, we denote $Updated_\delta(N)$ the set of nodes modified by δ. Analogously, we denote $True_\varrho(N) = \{n \in N | \varrho(n) = true\}$ and $False_\varrho(N) = \{n \in N | \varrho(n) = false\}$.

Definition 3.5 (Minimal Repair)**.** Let $RXT = \langle T, \delta, \varrho \rangle$ be an R-XML tree conforming DTD D, $\mathcal{FD}$ a set of functional dependencies and $R_1 = \langle \delta_1, \varrho_1 \rangle$, $R_2 = \langle \delta_2, \varrho_2 \rangle$ two repairs for RXT. We say that R_1 is smaller than R_2 ($R_1 \preceq R_2$) if $Updated_{\delta_1}(N_T) \cup False_{\delta_1}(N_T) \subseteq Updated_{\delta_2}(N_T) \cup False_{\delta_2}(N_T)$ and $False_{\delta_1}(N_T) \subseteq False_{\delta_2}(N_T)$. Repair R is *minimal* if there is no repair $R' \neq R$ such that $R' \preceq R$. □

An R-XML tree is used as an input for the main algorithm computing repaired R-XML tree described in Algorithm 3.1. First, the algorithm computes all the possible repairs of tuples which do not satisfy a functional dependency using the function `computeRepairs()` (lines 2-6). Next, all non-minimal repairs are

removed from all possible repairs (line 7). In the last step, all the repairs are merged and a unique repaired R-XML tree is returned.

Algorithm 3.1 XML Repair

Input:
$RXT = \langle T, \delta, \varrho \rangle$: R-XML tree conforming a DTD D
$\mathcal{FD} = F_1, \ldots, F_m$: Set of functional dependencies
Output: a unique repaired R-XML tree
1: $S = \emptyset$ // Set of repairs
2: **for each** $(F : S \rightarrow p) \in \mathcal{FD}$ s.t. $RXT \not\models_w F$ **do**
3: **for each** t_1, t_2 tuples of RXT s.t. t_1, t_2 do not weakly satisfy F **do**
4: $S = S \cup computeRepairs(F, t_1, t_2, RXT)$
5: **end for**
6: **end for**
7: $S = removeNonMinimal(S, RXT)$
8: $\langle \delta', \varrho' \rangle = mergeRepairs(S)$
9: **return** $\langle T, \delta' \cdot \delta, \varrho' \cdot \varrho \rangle$

Function `computeRepairs()` (in Algorithm 3.2) gets an R-XML tree, a functional dependency F and tuples t_1, t_2 of the R-XML tree as input and computes the repair as follows:

- If path p denotes a textual element, one of the two terminal values of $t_1.p$ or $t_2.p$ is changed, so that they become equal (line 3).
- Otherwise p denotes a node, so either the node $t_1.p$ or $t_2.p$ is marked as unreliable (line 5).
- For each path p_i on the left side of a functional dependency F
  - If path p_i denotes a textual element, then one of the two terminal values $t_1.p_i$ or $t_2.p_i$ is changed to the newly generated value ($\perp$) (line 9).
  - Otherwise p_i denotes a node, therefore one of the nodes $t_1.p_i$ or $t_2.p_i$ is marked as unreliable (line 11).

Function 3.2 $computeRepairs(F, t_1, t_2, RXT)$

Input:
$RXT = \langle T, \delta\varrho\rangle$: R-XML tree conforming to DTD D
$F : X \to p$ functional dependency
t_1, t_2 tuples of RXT
Output: S: Set of repairs

1: $S = \emptyset$
2: **if** $p \in StrPaths(D)$ **then**
3: $\quad S = S \cup \{\langle\{\delta(p(t_1)) = t_2.p\}, \varrho\rangle\} \cup \{\langle\{\delta(p(t_2)) = t_1.p\}, \varrho\rangle\}$
4: **else**
5: $\quad S = S \cup \{\langle\emptyset, \varrho_{\{t_1.p\}} \cdot \varrho\rangle\} \cup \{\langle\emptyset, \varrho_{\{t_2.p\}} \cdot \varrho\rangle\}$
6: **end if**
7: **for each** $p_i \in X$ **do**
8: $\quad$ **if** $p_i \in StrPaths(D)$ **then**
9: $\quad\quad S = S \cup \{\langle\{\delta(p_i(t_1)) = \perp_1\}, \varrho\rangle\} \cup \{\langle\{\delta(p_i(t_2)) = \perp_2\}, \varrho\rangle\}$
10: $\quad$ **else**
11: $\quad\quad S = S \cup \{\langle\emptyset, \varrho_{\{t_1.p_i\}} \cdot \varrho\rangle\} \cup \{\langle\emptyset, \varrho_{\{t_2.p_i\}} \cdot \varrho\rangle\}$
12: $\quad$ **end if**
13: **end for**
14: **return** S

3.1.4 Conclusion

The authors proposed a technique for repairing XML documents violating functional dependencies based on approaches proposed for relational database repairing. The algorithm introduces two possible repair primitives, which create many possible results from which those with minimal impact on the document are chosen. However the authors do not consider creation of new violations after repairing the initial violations as it is in [10]. Another disadvantage of this approach is that an unnecessary repair of some particular violation could be applied to an XML document because another repair could repair that violation before.

3.2 Querying and Repairing Inconsistent XML Data

Studying the problem of repairing inconsistent XML documents with respect to a set of functional dependencies and investigating the existence of repairs has been

introduced in [9]. The authors introduce two kinds of repair primitives. The first one is deleting “unreliable" nodes of document, the second one is inserting new nodes. Similarly to other approaches, authors prefer minimal set of repair primitives applied to the XML document to form a repair.

The introduced repair primitives the authors use in three different repair strategies consisting of:

1. *(general) repairs*, where both delete and insert operations are used,
2. *cleaning repairs*, where for documents interpreted as "dirty" only delete operations are used to repair inconsistencies,
3. *completing repairs*, where for documents interpreted as incomplete, insert operations are used.

3.2.1 General Repair

With the insert and delete operations as repair primitives, the structure of the XML document which conforms DTD D (defined in Definition 2.3) and violated functional dependency is updated. The insert operation is denoted as $\langle +[x]a[y], z\rangle$, where:

i) x is a node identifier

ii) a is a label

iii) y is either a node identifier or a value (y is a value, if $a \in \alpha \cup \{S\}$; otherwise it is a node identifier)

iv) z denotes the child of x which must immediately precede y ($\perp$ if y is inserted as the first or a single child of x).

The deletion is represented as $-[x]a[y]$, where x is a node identifier, $a \in \alpha \cup \tau \cup \{S\}$, and y is either a node id or a string value, denoting the node to be deleted.

The set R of update operations can be divided into two subsets: R^+ is the subset of all the insertion operations in R; R^- is the subset of all the deletion operations. A set R is said to be consistent if the following conditions hold:

1. the deletion of node implies the deletion of all descendant nodes;

2. insertions cannot refer to deleted nodes.

We say that two sets of update operations R_1 and R_2 are equivalent ($R_1 \equiv R_2$) if R_1 is equal to R_2 up to an injective renaming of node identifiers. Moreover, we say that $R_1 \preceq R_2$ if $R_1^- \subseteq R_2^- \vee R_1^- = R_2^-$ and $R_1^+ \subseteq R_2^+$. We say that $R_1 \sqsubseteq R_2$ if there exists a $R_2' \equiv R_2$ such that $R_1 \preceq R_2'$. At last $R_1 \sqsubset R_2$ if $R_1 \sqsubseteq R_2$ and $R_1 \not\equiv R_2$.

With basic repair operations for general repair of inconsistent XML document defined let us define a repair as follows:

Definition 3.6 (Repair)**.** Given an XML tree T, a DTD D and a set of integrity constraints IC (Definition 2.6), a set of update operations R is said to be a *repair* of T (with respect to D and IC) if $R(T) \models IC$, where $R(T)$ is application of consistent set of updates R to T, $R(T)$ conforms D and $\not\exists R' \sqsubset R$ such that $R'(T) \models IC$ and $R'(T)$ conforms D. □

With further investigation, the authors discover that the problem of deciding whether there exists a repair for a XML document in the presence of DTD with functional dependencies is undecidable. Therefore they consider restricted forms of repairs, more specifically cleaning and completing repairs.

3.2.2 Conclusion

In this approach the authors introduce different types of repairs (general, cleaning and completing) and focus on checking whether there exists a repair for many classes of integrity constraints (general integrity constraints, inclusion dependencies, functional dependencies, etc.). For the functional dependencies the authors found out that the problem of checking whether there exists a general repair for an XML document is $\mathcal{NP}$-complete.

3.3 Improving XML Data Quality with Functional Dependencies

The algorithm for repairing XML functional dependency violations which uses a modification of node value as the repair primitive has been proposed in [10]. To find the optimal repair of an XML document, the authors introduce a cost model, which assigns a weight to each leaf node in an XML document. The optimal repair is the one with the lowest repair cost which is measured by the total weight of the modified nodes.

The authors of this approach found out that repairing one functional dependency violation can violate another. Therefore they divide the main algorithm into two phases. In the first phase, the conflict hypergraph capturing the initial functional dependency violations is constructed and all the violations are fixed by modifying the values of all the nodes on a vertex cover of the conflict hypergraph. In the second phase, remaining violations are resolved by modifying the violating nodes and their core determinants to prevent of introducing new conflicts.

3.3.1 Cost Model and Repairing Primitive

Each leaf node v of XML tree is associated with a weight from range $[0, 1]$, which is denoted $W(v)$. Let us assume that the larger the weight of the leaf, the more reliable it is. The weight may be automatically generated by statistical methods or it can be assigned by the user.

As was mentioned earlier, a repairing primitive is a node value modification, where for repairing algorithm a combination of two rules to resolve violation is used. Let us have an FD $\sigma = (P, P', (P_1, \ldots, P_n \rightarrow P_{n+1}))$ (from Definition 2.9) and consider two nodes v_1 and v_2 matching path P' in a subtree rooted at a node in $\{[\![p]\!]\}$. If the child nodes of v_1 and v_2 qualified by paths P_i have equal values for all $i \in [1, n]$ and their child nodes qualified by P_{n+1} have different values, then v_1 and v_2 violates σ. The first rule used to repair this violation is to change the value of the node qualified by P_{n+1} from v_1 to the value of v_2's child node that matches P_{n+1} (or reversely). The second rule is to choose an arbitrary $P_i (i \in [1, n])$ and introduce a new value to the node qualified by P_i from v_1 (or v_2).

Definition 3.7 (Optimal repair)**.** Being given an inconsistent XML document T violating a set Σ of FDs, the repair T_R of T is called *optimal repair*, if T_R has the minimum cost among all repairs of T. The cost $cost(T_R)$ is defined as:

$$cost(T_R) = \sum_{v \in T} w(v) \times dist(v, v_R),$$

where $dist(v, v_R) = 1$ if $val(v) \neq val(v_R)$, otherwise $dist(v, v_R) = 0$. □

3.3.2 Initial Conflicts Hypergraph

A weighted hypergraph is used in the first part of the repair algorithm as a tool modeling initial functional dependency violations in an XML document. Hypergraph g of XML document T can be defined as a pair $g = (V, E)$, where V stands for a set of elements (called nodes), and E is a set of non-empty subset of V called hyperedges, more accurately each hyperedge indicates a set of value nodes violating FDs. Since hypergraph is weighted and a cost model is used in this approach, each node $v \in V$ of the hypergraph is assigned with a weight $w(v)$, which is the same as the weight of v in T.

To actually resolve the problem of repairing FD violations in an XML document, the authors convert this problem into well-known problem of weighted vertex cover for hypergraph [11]. Let us have hypergraph $g = (V, E)$, where each hyperedge $e \in E$ is a set of value nodes which violate some FD. In a repair of an inconsistent XML document, for each hyperedge at least one value node is modified, therefore it is essential to find a vertex cover (VC) for g, which is a set $S \subseteq V$, such that for all edges $e \in E$, $S \cap e \neq \emptyset$. Since the hypergraph is weighted, we can define weight of VC as the total weight of all vertices in S.

The algorithm fixing initial FD violations is shown in Algorithm 3.3. The algorithm uses an approximation algorithm to find VC for the minimum weighted vertex cover proposed in [11].

Algorithm 3.3 Fix-Initial-Conflicts

Input: An XML document T, a set Σ of FDs.
Output: A modified document T.

1: // Create the initial conflict hypergraph g of T w.r.t Σ
2: // Use a known algorithm to find an approximation VC for the minimum weighted vertex cover of g
3: $remaining$:= VC
4: **while** there are two target nodes $v_1, v_2 \in T$ violating a FD $\sigma \in \Sigma$, and $v_1[\![P_{n+1}]\!]$ or $v_2[\![P_{n+1}]\!]$ is the only node in VC from the set of nodes $\{v_1[\![P_i]\!](i \in [1, n+1])\} \cup \{v_2[\![P_i]\!](i \in [1, n+1])\}$. (W.l.o.g assume the violation is as follows: $\sigma = (P, P', (P_1, \ldots, P_n \rightarrow P_{n+1}))$, $v \in \{[\![P]\!]\}$, $v_1, v_2 \in \{v[\![P']\!]\}$, $v_1[\![P_i]\!] \equiv v_2[\![P_i]\!]$ for all $i \in [1, n]$, and $v_1[\![P_{n+1}]\!] \not\equiv v_2[\![P_{n+1}]\!]$.) **do**
5: $val(v_1[\![P_{n+1}]\!]) := val(v_2[\![P_{n+1}]\!])$ // W.l.o.g, we assume $v_1[\![P_{n+1}]\!]$ is in VC
6: **end while**
7: **for each** node $u \in remaining$ **do**
8: $val(u) := gen\_new\_value()$
9: // Introduce new values to all the remaining nodes in VC
10: **end for**

3.3.3 Resolving Violations Thoroughly

After repairing initial FD violations, there is a chance that new violations may be introduced, therefore the authors provided a method to do modifications on value nodes without incurring new conflicts (Algorithm 3.4). This method uses core determinant C_u of value node u defined as follows:

Definition 3.8 (Core Determinant)**.** Being given an XML document T, a set Σ of FDs and a node u in T, we say that a set of nodes $\{u_1, u_2, \ldots, u_n\}$ is a $\sigma-determinant$ of u, if there exists a nontrivial FD $\sigma = (P, P', (P_1, \ldots, P_n \rightarrow P_{n+1}))$ logicaly implied by Σ, such that $\exists v \in \{[\![P]\!]\}, \exists v_1 \in \{[\![P']\!]\}, v_1[\![P]\!] = u_i$ for $i \in [1, n]$, and $v_1[\![P_{n+1}]\!] = u$.
We say that a set C_u of nodes is a *core determinant* of u, if (a) for every nontrivial FD σ implied by Σ and every set W that is $\sigma-determinant$ of u, $C_u \cap W \neq \emptyset$; and (b) for any proper subset C'_u of C_u, there exists a nontrivial FD σ implied by Σ, and a set W that is $\sigma-determinant$ of u, $C'_u \cap W = \emptyset$. □

Algorithm 3.4 Resolve-Remaining-Violations

Input: An XML document T, a set Σ of FDs.
Output: A modified document T, with all the violations fixed.

```
1: while there are FD violations in T w.r.t. Σ do
2:    pick a violating value node u from T w.r.t. Σ
3:    let C_u be a core determinant u
4:    for each node w ∈ (C_u ∪ {u}) do
5:       val(w) := gen_new_value()
6:       // it guarantees that no new violations will be introduced
7:    end for
8: end while
```

3.3.4 Conclusion

The authors introduced an effective two-step heuristic method to solve a problem of finding optimal repair of XML violations against functional dependencies, which is $\mathcal{NP}$-complete. Moreover, they experimentally verified the effectivity and scalability of their approach using real-life and synthetic data.

Chapter 4

Proposed Algorithm

The main goal of this thesis is to propose an algorithm to repair an XML document violating functional dependencies defined for this document. We use an XML tree and tree tuples to represent XML data. We would like our algorithm to have the following features::

- Incorporation of a weight model into the XML data representation to provide the selection of repair candidates with the lowest modification cost.
- Involvement of a user in the process of finding and applying repair candidates.
- Application of a single repair candidate at the time and then recalculate repair candidates again to prevent using unnecessary repairs to repair one violation and also to prevent of introducing new violations, which will not be repaired.
- The paths which form a functional dependency are described with XPath language [12], where only paths with basic construction are allowed (only path constructed with "/", "//" or "@", no wildcards, "[]" or another constructs are allowed).
- We introduce a new concept of the so-called repair group, which clusters repair candidates (repairs for repairing one FD violation) repairing the same violation, or modifying the same part of the XML tree.

4.1 Repairing Algorithm

The proposed algorithm is based on the algorithm described in 3.1 presented in [8]. This algorithm was chosen because of simple representation of the XML data using a concept which corresponds with concept used in relational databases. Another reason, besides modification of node value as an update operation, is using also marking particular node information as unreliable, which can reveal forgotten inconsistencies in the data.

Algorithm 4.1 Repair RW-XML tree

Input:
$RXT = \langle RT, \omega \rangle$: RW-XML tree conforming to DTD D
$\mathcal{FD} = F_1, \ldots, F_m$: Set of functional dependencies
Output: a unique repaired RW-XML tree

```
1: resultRXT = RXT
2: while resultRXT ⊭_w FD do
3:    S = ∅ // Set of repair groups
4:    for each (F : S → p) ∈ FD s.t. RXT ⊭_w F do
5:       for each t1, t2 tuples of RXT s.t. t1, t2 do not weakly satisfy F do
6:          S = S ∪ computeRepairGroup(F, t1, t2, RXT, S)
7:       end for
8:    end for
9:    R = getRepair(S, RXT)
10:   resultRXT = applyRepair(R, resultRXT)
11: end while
```

The algorithm is split into three steps, where the second and the third step are repeated until all violations are repaired. In the first step, XML document and FDs are loaded, and XML tree with corresponding tree tuples are created. Next step of the algorithm computes repair groups containing repair candidates for FD violations. In the third step the chosen repair candidate is applied to an XML tree. In Algorithm 4.1 we can see the simplified process of repairing XML FD violations, which covers the second and the third step of the whole algorithm.

4.1.1 Initial data model

To represent an XML document we use an extended R-XML tree (Definition 3.1), called RW-XML tree, which has weights assigned to each node of the tree. The weight of a node is from interval $[0, 1]$ and indicates correctness of the data the particular node holds, meaning the higher the weight is, the more correct that particular node is. The weights are used to measure the cost of repair candidates, where candidate with the lowest cost is picked to be applied to the XML tree. This is also the first place where some kind of user interaction can be implemented. The user could assign the weights to nodes manually or could use some sort of statistical methods to generate them automatically. If the user does not assign weights nor does he use statistical methods, a default weight is assigned to all nodes.

Example 8. Consider the XML document and corresponding XML tree from Example 2.1. Let us assign weights to nodes defined by XPath constructs shown in Table 4.1. The XML tree with assigned custom and also default weights is shown in Fig. 4.1.

XPath	weight
/bib/book/title/text()	0.8
/bib/book/written_by/author	0.6
/bib/book/written_by/author/@ano	0.1

Table 4.1: Table of custom weights assigned to the XML tree.

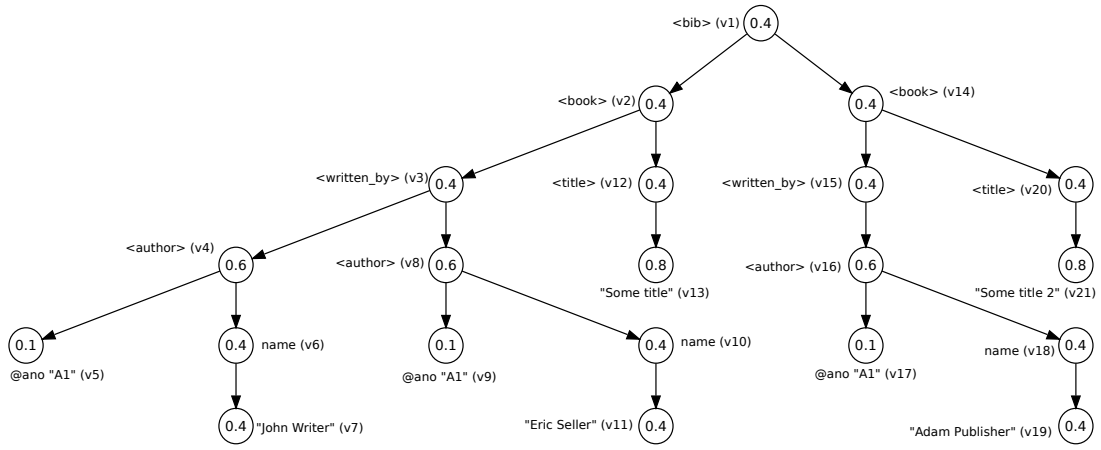


Figure 4.1: The XML tree with assigned weights to the nodes.

□

Definition 4.1 (RW-XML tree)**.** A *RW-XML tree* is a pair $RWXT = \langle RXT, \omega \rangle$, where RXT is an R-XML tree and ω is a weight function from N_T to $[0, 1]$. □

After creating the RW-XML tree from input XML data, a set of tree tuples (defined in Definition 2.7) is constructed. Since the set $paths(D)$, containing all possible paths defined in DTD D, can be infinite (DTD can define recursive structure of elements), the actual content of $paths(D)$ is modified to reflect the current structure of the RW-XML tree. Since definition of a tree tuple says that answer to the path p contains at most one element, our modification of $paths(D)$ has no effect on constructing tree tuples if a DTD defines some optional path which is not defined for the RW-XML tree $RWXT$ (the set $p(RWXT)$ is empty).

Functional dependencies as defined in Definition 2.8 consist of paths described with XPath. As was described before, these paths can only have basic structure shown in Example 9.

Example 9. Example of paths that can be used in functional dependencies:

```
//bib/book/author
/bib/book/author/@ano
/bib/book/author/name/text()
```

4.1.2 Computing Repair Groups

With RW-XML tree $RWXT$ and corresponding tree tuples from input data created, we can now proceed to compute repair groups. Repair group is a set of repair candidates, which repairs the same FD or modifies the same part of $RWXT$. Each repair candidate is a pair of functions δ and ϱ ($\langle\delta, \varrho\rangle$), which either modifies value of an RW-XML tree node (δ is defined) or marks a node as unreliable (ϱ is defined). The δ function of a repair candidate is defined the same way as in the XML tree (Definition 2.2) and defines a new value of the RW-XML tree node. Similarly, the ϱ function defines the node which is marked as unreliable.

To be able to compute repair groups, we need to decide which FDs violate $RWXT$. This can be achieved by finding all tree tuple pairs that do not weakly satisfy particular FD (Definition 3.2). A repair group is then computed for each tuple pair (line 6 in Algorithm 4.1).

The function `computeRepairGroup()` responsible for creating the repair group is shown in Algorithm 4.2. The function gets an RW-XML tree, a functional dependency F, tuples t_1, t_2 and a set of repair groups RGS and computes the repair group containing repair candidates as follows:

1. First the repair candidates are created using function `computeRepairs()` from Algorithm 3.1 (line 1). Since an R-XML tree is a special case of RW-XML tree where all the weights are equal to zero, we can pass RW-XML tree as a parameter to this function.

2. Next a check is performed whether the repair candidates intersect other candidates from existing repair groups (line 2)

   - If the candidates intersect with some group, they are added to this group (line 3)
   - Otherwise a new repair group containing repair candidates is created (line 5)

Example 10 demostrates repair candidates of one repair group modifying node values and marking nodes as unreliable.

Function 4.2 *computeRepairGroup*(F, t_1, t_2, RXT, RGS)

Input:
$RXT = \langle T, \omega \rangle$: RW-XML tree conforming to DTD D
$F : X \rightarrow p$ functional dependency
t_1, t_2 tuples of RXT
RGS set of repair groups
Output: RG: repair group
1: $S = computeRepairs(F, t_1, t_2, RXT)$ // set of repair candidates
2: **if** $candidatesIntersectRepairGroups(S, RGS)$ **then**
3: $RG = getIntersectingRepairGroup(S, RGS)$
4: **else**
5: $RG = createNewRepairGroup(S)$
6: **end if**
7: **return** RG

Example 10. Consider XML data and functional dependency from Example 7, where XML data is graphically represented as XML tree XT in Fig 2.1. The XT violates FD

$$\{/bib/book, /bib/book/written\_by/author/@ano\} \rightarrow /bib/book/written\_by/author$$

because two authors of the same book have the same value of attribute @*ano*. One repair group with these repair candidates is created:

- $R_1 = \langle \{\delta(v5) = \perp\}, \varrho_{\{\}}(v) \rangle$
- $R_2 = \langle \{\delta(v9) = \perp\}, \varrho_{\{\}}(v) \rangle$
- $R_3 = \langle \{\}, \varrho_{\{v4,v5,v6,v7\}}(v) \rangle$
- $R_4 = \langle \{\}, \varrho_{\{v8,v9,v10,v11\}}(v) \rangle$
- $R_5 = \langle \{\}, \varrho_{\{v2,v3,\ldots,v13\}}(v) \rangle$

First two repair candidates modify the value of an attribute @*ano* by assigning newly generated value ($\perp$). Next two repair candidates mark each *author* node with its child nodes that have the same @*ano* attribute respectively. The last candidate marks as unreliable a whole subtree with the *book* node holding *authors* with the same attribute as the root node. □

Weight model of a Repair Group

In this section, we introduce a weight model for a repair group, which will be used in the next step of our repair algorithm. Before we define a weight of a repair group, let us introduce some important notations.

Being given a repair candidate R, $Modified_{\delta}(R)$ denotes the set of nodes modified by δ function from R. Analogously, we denote $Modified_{\varrho}(R)$ the set of nodes modified by ϱ function. The count of nodes modified by repair candidate R is denoted with $\lambda(R)$. Last, we denote $PS(R)$ a set of paths defining all nodes modified by R (example of the set $PS(R)$ is shown in Example 11).

Example 11. Consider the XML tree from Fig. 2.1 and a repair candidate, which marks nodes $v12$ and $v13$ as unreliable. Then the set $PS(R)$ contains the following paths:

```
/bib/book/title
/bib/book/title/text()
```

□

Since each repair candidate consists of a set of nodes which are modified (marked as unreliable or having a modified value), we can compute the cost of each candidate. It is important to say that unlike in original approach, we do not prefer repair candidates that modify the value of the nodes to those marking nodes as unreliable. Therefore we added a coefficient k to the calculation of the repair candidate cost, so that the priority of repair candidate marking node as unreliable can be achieved. The definition of repair candidate cost is as follows:

Definition 4.2 (Repair Candidate Cost)**.** Being given an RW-XML tree RXT and a repair candidate R, we define the *cost* of R as:

$$cost(R) = \sum_{u \in Modified_{\delta}(R)} \omega(u) + \sum_{v \in Modified_{\varrho}(R)} \omega(v) \cdot k,$$

where k is the priority of repairing the candidate by modifying the node value in constrast to marking it unreliable. □

By default, k is set to such value that the cost of repair candidate marking as an unreliable node will be higher than the one that modifies node value. However, this is another place where user can intervene and change the priority of repair candidates. We show in Example 12 how k can change the costs of repair candidates.

Example 12. Consider XML tree XT and repair candidates from Example 10. The default weight of all nodes v of XT is $\omega(v) = 0.5$. If the coefficient $k = 1$, cost of repair candidates would be as follows: $cost(R_1) = 0.5$, $cost(R_2) = 0.5$, $cost(R_3) = 2$, $cost(R_4) = 2$ and $cost(R_5) = 6$. Let us assume that a repair candidate with the lowest repair cost will be applied to XT. If we order costs of repair candidates from the lowest to the highest, we can see that candidates which have marked unreliable nodes are in the end (and will not be applied to XT).

However, if we set the coefficient $k = 0.1$, costs of candidates would be sorted as follows: $cost(R_3) = 0.2$, $cost(R_4) = 0.2$, $cost(R_1) = 0.5$, $cost(R_2) = 0.5$ and $cost(R_5) = 0.6$. We can see that order of costs has significantly changed and the candidate which will be applied is R_3 (marking nodes as unreliable). □

And, finally, since a repair group is a set of repair candidates, its weight can be defined as follows:

Definition 4.3 (Repair Group weight)**.** Being given an RW-XML tree RXT and a repair group RG, we define the *weight* of RG as the sum of costs of all repair candidates in the RG.

4.1.3 Repair Candidate Selection and Application

The last step of our algorithm can be divided into two parts, namely selection of the repair candidate and the subsequent application of the candidate to RW-XML tree.

Selection of the Repair Candidate

The previous step of the algorithm computes repair groups containing repair candidates, which can repair violations of provided FDs. From these candidates,

we must choose the one that is applied to the RW-XML tree. Since we introduced a weight model to the repair group, we can use the weight of repair groups and the cost of repair candidates to choose a suitable candidate. In our approach, we introduce two distinct algorithms: first that does not involve the user in the process of selection and the second one that uses the user interaction.

The former algorithm simply selects the first repair group with the lowest weight, and from it the repair candidate with the lowest cost is selected.

The latter algorithm allows the user to choose the repair candidate which is the most suitable for his needs. Very important aspect of all user interactions in this kind of algorithm is that the user will not be willing to select more than, e.g., ten repairs. We could simply allow user to switch to the first selection algorithm, but that does not take in account previous user selections of repair candidates. Therefore, we introduce in this algorithm a functionality able to guess his next selection from selection done by the user before.

The function `selectRepairByUser()` responsible for selection of repair candidate involving user interactivity is shown in Algorithm 4.3.

First it decides, whether the algorithm is in a state where the user is selecting repair candidates (the user selection mode), or the user leaves decision making on the algorithm using his previous selections (the guess mode) (line 2). If we are still in user selection mode, the user chooses from repair groups sorted by the weight the most convenient one, and from this group the user chooses the repair candidate that will be applied on RW-XML tree (line 3). Repair candidates in each repair group are also sorted by the cost, working as a hint for the user which repair would be chosen by the previous automatic method of selection. The last step of user selection mode is saving the information from selected repair candidate (line 4). This information consists of the FD which this candidate repairs, nodes the repair changes (their paths) and also whether the change was a value modification or marking a node as unreliable.

If the user has decided not to select repair candidates by himself anymore, the algorithm goes into the guess mode (starting at line 5). In this mode the algorithm checks all repair groups whether one of them contains a repair candidate that is sufficiently similar to some previously selected repair candidate (line 6-12). This similarity is checked by function `canBeUsedUserSelection()` shown

in Algorithm 4.4. If neither of the candidates is sufficiently similar, the algorithm chooses the repair candidate with the lowest cost from the repair group with the lowest weight (line 13-14).

Function 4.3 *selectRepairByUser*(RGS, SR, t)

Input:
RGS set of repair groups
SR: the set of repair candidates previously selected by the user
t: modified nodes count threshold of previously selected repair candidates
Output: R: repair candidate

```
1:  R = ∅
2:  if isUserSelection() then // the user selection mode
3:     R = getRepairFromUser() // repair candidate selected by user
4:     saveSelectedRepair(SR, R)
5:  else // the guess mode
6:     for each RG in RGS do
7:        for each RC in RG do // for all repair candidates in the repair group
8:           if canBeUsedUserSelection(SR, t, RC) then
9:              return RC
10:          end if
11:       end for
12:    end for
13:    RG = getFirstRG(RGS) // get the repair group with smallest weight
14:    R = getFirstRepairCandidate(RG) // get the repair candidate with the
       lowest cost
15: end if
16: return R
```

The function `canBeUsedUserSelection()` gets the current repair candidate RC, the set of all previously selected repair candidates SR, and the modified nodes count threshold t from interval $(0, 1]$ of previously selected repair candidate and it determines whether the RC is sufficiently similar to some repair candidate from SR. To be similar with some previously selected repair candidate S, RC must repair the same FD as S, it must use the same update operation as S and $\lambda(RC) \geq \lceil \lambda(S) \cdot t \rceil$ (line 2). Furthermore, if $\lambda(RC) = \lceil \lambda(S) \cdot t \rceil$, the set of paths $PS(RC)$ must be a subset of $PS(S)$ (line 3). Otherwise, if $\lambda(RC) > \lceil \lambda(S) \cdot t \rceil$, there must exist a subset of $PS(S)$ with $\lceil \lambda(S) \cdot t \rceil$ elements that is a subset of

$PS(RC)$. In other words, without taking into account the threshold t, if a repair candidate R_1 modifies some nodes, the sufficiently similar repair candidate R_2 is the one that modifies at least the same nodes as R_1 (when we say the same nodes we mean same paths representing those nodes). The threshold t can reduce the number of modified nodes of some previously selected repair candidate R_1, which means that sufficiently similar repair candidate needs to modify fewer nodes that are similar with nodes modified by R_1. In Example 13 the usage of the threshold t is shown.

Function 4.4 $canBeUsedUserSelection(SR, t, RC)$

Input:
SR: the set of previously selected repair candidates by the user
t: a suitability threshold $(0, 1]$ of previously selected repair candidates
RC: the current repair candidate

Output: **true** if RC is similar to some previous repair candidate.

```
1: for each S in SR do
2:   if S repairs the same FD as RC and S use the same update operation as
     RC and ⌈λ(SC) × t⌉ ≤ λ(RC) then
3:     if ⌈λ(SC) × t⌉ = λ(RC) and PS(RC) ⊆ PS(SC) then
4:       return true
5:     end if
6:     if ⌈λ(SC) × t⌉ < λ(RC) and there exists a subset s of PS(SC) with
       ⌈λ(SC) × t⌉ elements, that s ⊆ PS(RC) then
7:       return true
8:     end if
9:   end if
10: end for
11: return false
```

Example 13. Consider repair candidate R_1 selected by user, that marks as unreliable two nodes and $PS(R_1) = \{/bib/book/title, /bib/book/title/text()\}$. Next, consider repair candidate R_2 with $PS(R_2) = \{/bib/book/title/text()\}$, that the modified node is marked as unreliable and both R_1 and R_2 repairs the same FD. If $t = 1$, R_2 is not considered as sufficiently similar to R_1, because $\lceil \lambda(R_1) \times t \rceil > \lambda(R_2)$. However, if $t = 0.5$, $\lceil \lambda(R_1) \times t \rceil = \lambda(R_2)$ and $PS(R_2) \subseteq PS(R_1)$, then R_2 is sufficiently similar to R_1. □

Application of the Repair Candidate

In this part, the selected repair candidate is finally applied to the RW-XML tree. If after this part the RW-XML tree does not violate any FDs, the whole repair algorithm ends at this point, otherwise repair groups are regenerated and the selection of the repair candidate part takes place again.

To apply the selected repair candidate R to the RW-XML tree RXT means to compose δ and ϱ functions of R with the corresponding functions of R-XML tree contained in RXT. These compositions are defined in Definition 3.3 and 3.4.

After application of the repair candidate, some of RW-XML tree nodes could become unreliable, which can lead to the situation that some of tree tuples are not anymore considered a tuple (it is no longer a maximal subtree). Therefore, before regeneration of repair groups we need to check all tree tuples to see whether they satisfy definition of tree tuple.

Chapter 5

Implementation

A part of this work is an experimental implementation of our proposed algorithm. Besides our approach, we have also implementened the algorithm proposed in [8], to be able to compare it with our algorithm. Our algorithm has been implemented in the Java language, version 6.0 as a part of the schema inference framework for XML called jInfer [13].

5.1 jInfer Framework

jInfer framework was developed as a Software Project at the Faculty of Mathematics and Physics, Charles University at Prague. The framework is based on the NetBeans platform and is mainly used for XML schema inference. Although our proposed algorithm is not dealing with schema inference, jInfer is designed to be extensible and is suitable for incorporation of our algorithm.

The whole process of repairing inconsistent XML data is divided three steps (illustrated in Fig. 5.1):

1. Import of input data (XML, functional dependencies, weights) into an internal representation - *Initial Model*.
2. Repairing of *Initial Model* with optional help of user interaction.
3. Export back the repaired data into XML format.

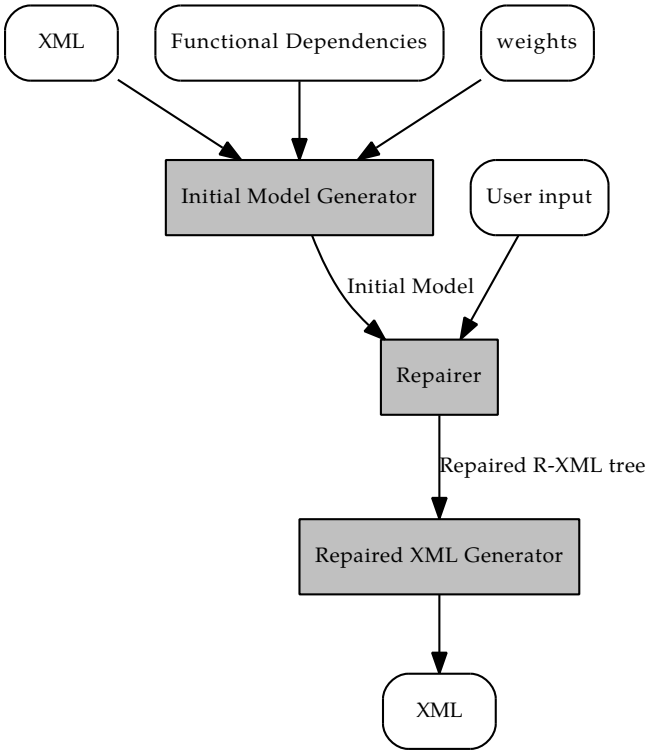


Figure 5.1: High-level view of the repair process

5.2 Architecture

The main part of our algorithm is implemented as the `FDRepairer` module of the jInfer framework. The core package of the module is `cz.cuni.mff.ksi.jinfer.functionalDependencies` and contains classes representing base object important for repairing (Tuple, RW-XML tree, Path etc.). This package is further divided into the following subpackages:

fd contains data structure of functional dependencies used as an input to the algorithm.

interfaces contains Java interfaces necessery for integration in the jInfer framework.

modelGenerator contains classes responsible for creating data structures from loaded input files.

newRepairer contains class `NewRepairerImpl`, the class that implements our proposed algorithm. Also it contains other data structures introduced in our approach

properties contains classes reponsible for showing properties of this extension in the jInfer.

repairer contains classes implementing the original algorithm from [8].

weights contains data structure for node weights used as an input to the algorithm.

5.2.1 Input Files Processing

Files that can be added as an input to our algorithm are of three types. The XML data, file containing functional dependencies and file containing weights. First, the XML is processed with build-in Java DOM parster. Files with FDs and weights both contain XML data with specific structure defined as XML Schema in Fig. B.2 and B.1 and are both processed with JAXB [14].

5.3 Restriction of Implementation

The `FDRepairer` module of the jInfer framework is an implementation of our proposed algorithm presented in Chapter 4. However, one part of the algorithm have not been implemented yet, namely the intersection of repair candidates with existing repair group.

5.4 Building and Executing

For building jInfer with `FDRepairer` from sources, please refer to the tutorial at `http://jinfer.sourceforge.net/building_jinfer.html`. To execute jInfer, install plugins from enclosed CD or built from sources into NetBeans.

Appendix C contains a simple user guide with screenshots about how to use `FDRepairer`.

Chapter 6

Experimental Results

In this chapter, experimental results of the `FDRepairer` module of the jInfer framework are presented. Set of documents and functional dependencies have been provided as an input and results from the former algorithm have been compared with results from `FDRepairer`. Documents in provided datasets are split into real-world and synthetic XML documents.

6.1 Datasets

All provided documents with functional dependencies and also repaired documents are placed on the enclosed CD in the directory `/datasets/`.

6.1.1 Real-World Data

The first real-world dataset originates from the XML data repository (`http://www.cs.washington.edu/research/xmldatasets/`), specifically the Course data derived from university websites. The course data consist, along with other information, of day of the week, time and place (building and room), where each course is situated. DTD of one of the dataset file is shown in Fig. 6.1. The consistency of data is evaluated against FD defined as: *Two courses starting at the same date and time are each situated in a different place.*

```
<!ELEMENT root (course*)>
<!ELEMENT course (reg_num,subj,crse,sect,title,units,
                  instructor,days,time,place)>
<!ELEMENT reg_num (#PCDATA)>
<!ELEMENT subj (#PCDATA)>
<!ELEMENT crse (#PCDATA)>
<!ELEMENT sect (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT units (#PCDATA)>
<!ELEMENT instructor (#PCDATA)>
<!ELEMENT days (#PCDATA)>
<!ELEMENT time (start_time,end_time)>
<!ELEMENT start_time (#PCDATA)>
<!ELEMENT end_time (#PCDATA)>
<!ELEMENT place (building,room)>
<!ELEMENT building (#PCDATA)>
<!ELEMENT room (#PCDATA)>
```

Figure 6.1: DTD of reed.xml from course dataset

The second dataset is a set of actors of IMDB database obtained from the Niagara data source (`http://www.cs.wisc.edu/niagara/data.html`). The dataset consists of a set of authors, where for each author a set of movies he played in is listed. DTD of this dataset is shown in Fig. 6.2. For this dataset, we defined FD as follows: *For each two authors, which played in the movie with the same name, the year of release of this movie must be the same.*

```
<!ELEMENT W4F_DOC (Actor)>
<!ELEMENT Actor (Name,Filmography)>
<!ELEMENT Name (FirstName, LastName)>
<!ELEMENT FirstName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
<!ELEMENT Filmography (Movie)*>
<!ELEMENT Movie (Title,Year)>
<!ELEMENT Title (#PCDATA)>
<!ELEMENT Year (#PCDATA)>
```

Figure 6.2: DTD of actors.xml

6.1.2 Synthetic Data

Two synthetic datasets have been constructed, the former represents data introduced in Example 2.1 with FD presented in Example 5. The latter is created according to Example 1 introduced in [10]. DTD of the former dataset is identical with DTD defined in Example 2. The latter datasets DTD is shown in Fig. 6.3.

```
<!ELEMENT customers (country)>
<!ELEMENT country (name,c_list)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT c_list (customer)*>
<!ELEMENT customer (no,phone,zip,city)>
<!ELEMENT no (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT zip (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

Figure 6.3: DTD of customers.xml

6.2 Algorithms Comparison

For each dataset we perform the following. First, we run the original algorithm and collect from a repaired document informations about how many nodes have been modified with particular update operation. Moreover, we split the value modification operation into two types. The first is changing value to new one and the second one is copying the value from one node to another.

After that, we use our proposed algorithm with minimal repair candidate selection mode, and along with the data we collect for original algorithm, we also gather information on how many repair groups have been created before the first application of repair candidate and the total number of picked repair groups. We run our algorithm multiple times with different value of the coefficient k and assignment of node weights w.

The first real-world data (course data) is represented by `reed.xml` and `wsu.xml`. For each document, we execute our algorithm with three different settings. First two use the default value of k ($k = 1.5$), for the last one, we set $k = 0.005$. For the last two executions, we set for `reed.xml` (`wsu.xml`) the weight w of nodes represented by XPath `/root/course/time/start_time/text()` (resp. `/root/course/time/start/text()`) to 0.1.

The document `actors.xml` represents the second real-world dataset. Our algorithm is executed two times, where for both executions $k = 1.5$ is set. Moreover, the second has weights of nodes represented by XPath `/W4F_DOC/Actor/Filmography/Movie/Year/text()` set to $w = 0.2$.

Generated datasets represented by `bib.xml` and `customers.xml` documents have been both used as a input for our algorithm two times, where first execution was done with the default value of k. The second execution on `bib.xml` uses $k = 0.1$ and on `customers.xml` uses $k = 1.5$ and $w = 0.2$ is set to all nodes represented by `/customers/country/c_list/customer/city/text()`.

dataset	repairer	k	w	RG	RG total	U	NV	ChV
reed.xml	old repairer	-	-	-	-	0	780	0
	`FDRepairer`	1.5	-	218	195	0	195	0
	`FDRepairer`	1.5	0.1	218	195	0	195	0
	`FDRepairer`	0.005	0.1	218	195	5265	0	0
wsu.xml	old repairer	-	-	-	-	0	2612	0
	`FDRepairer`	1.5	-	3520	653	0	653	0
	`FDRepairer`	1.5	0.1	3520	653	0	653	0
	`FDRepairer`	0.005	0.1	3520	653	20833	0	0
actors.xml	old repairer	-	-	-	-	0	62	62
	`FDRepairer`	1.5	-	69	62	0	32	30
	`FDRepairer`	1.5	0.2	69	79	0	0	79

Table 6.1: Real-World datasets

6.3 Results

All information gathered from real-world and synthetic datasets is shown in Tables 6.1 and 6.2. The columns of both tables are defined as follows: *RG* is the count of repair groups created before application of repair candidate, *RG total* is the total count of picked repair groups, *U* specifies the number of nodes marked as unreliable, *NV* defines the number of nodes changing value to a new one and finally *ChV* specifies the number of nodes with value copied from another node.

From the information presented in these tables is apparent that our approach found for each dataset a repair that modifies less nodes than the original repair algorithm. One exception where our algorithm modifies more nodes is the case when we set value of k near 0, which marks nodes as unreliable. With this setting we want to demonstrate that with our approach it is possible to mark nodes as unreliable instead of modifying their values. This is not possible with the original repairer, since it prefers node value modification to marking node as unreliable.

Setting the weight to particular nodes causes two different behaviour of the algorithm. First, setting the weights effects the choise of strategies used to repair the violations. In datasets `actors.xml` and `customers.xml` setting the custom weight to nodes causes the node values are changed to values of other nodes. Second causes modification of nodes defined by the path specifying the weights.

dataset	repairer	k	w	RG	RG total	U	NV	ChV
bib.xml	old repairer	-	-	-	-	0	913	0
	`FDRepairer`	1.5	-	1444	913	0	913	0
	`FDRepairer`	0.1	-	1444	913	5478	0	0
customers.xml	old repairer	-	-	-	-	0	99	99
	`FDRepairer`	1.5	-	232	136	0	72	64
	`FDRepairer`	1.5	0.2	232	361	0	0	361

Table 6.2: Synthetic datasets

In datasets `reed.xml` and `wsu.xml` setting the weights still changes the value of nodes to a newly generated one, however for `reed.xml` only values of nodes defined by `/root/course/time/start_time/text()` are modified and for `wsu.xml` are modified values defined by `root/course/time/start/text()`.

Another interesting observation is that for almost all datasets, our algorithm picks less repair groups (from which picks repair candidate to apply to document) than were created before first pick. It means that one of the repair candidate repairs more than one FD violation and therefore less value modifications are needed. In the case where the number of picked repair groups outreach the number of firstly created ones, it means that some repair candidate introduces new violation which was consequently repaired in the next iteration of our algorithm.

Chapter 7

Conclusion

The aim of this thesis was to propose and implement an algorithm repairing XML functional dependencies violations. At first, existing solutions were analyzed and described. The result of the analyses was that none of the recent approaches uses, even in only limited form, user interaction in the process of finding suitable repair for the FD violation. Consequently, we proposed algorithm based on [8] that uses user interaction in various ways.

First, we incorporated a weight model into the XML data representation, which allows us to measure the modification cost of each possible repair candidate. This is also the first place for the user to interact with the repair process. Next, we have introduced a new concept of repair candidates clusters called repair groups. These clusters group repair candidates according to the violation which they are repairing or part of the XML data they are modifying.

The main user interaction part of our algorithm is the selection of the repair group and consequently the repair candidate which is applied to the XML document. Besides the repair candidate selection we also introduced a mechanism allowing to guess the next selection based on the previous candidates selected by the user, if the user does not want to select the repair candidate anymore.

The experimental implementation of the original as well as proposed algorithms based on the jInfer framework. Both algorithms have been compared against each other on synthetic as well as real-world datasets. From the data

gathered from the comparison it is clear that our approach has found repairs with less modifications applied on the XML data. We have also shown that with the user interaction it is possible to change the usage of update operations used in repair, which may create more reasonable result for the user.

7.1 Future Work

Although our approach has introduced user in the process of repairing FD violations in XML data, there is still work to be done. The first main task in the further work is to extend paths defining functional dependencies. With usage of more constructs that XPath provides one can define more sophisticated FDs.

Another part of the algorithm to be improved is clustering repair candidates into repair groups. The user can provide some additional criteria, that can change the resulting repair.

Last but not least, guessing part of the user selection algorithm can be improved by using more sophisticated heuristic algorithm to select proper repair candidate with regard to previous candidates selected by the user.

Bibliography

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

[2] P. Walmsley and D. C. Fallside, "XML schema part 0: Primer second edition," W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/.

[3] H. S. Thompson, M. Maloney, D. Beech, and N. Mendelsohn, "XML schema part 1: Structures second edition," W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

[4] A. Malhotra and P. V. Biron, "XML schema part 2: Datatypes second edition," W3C recommendation, W3C, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/.

[5] P. Buneman, W. Fan, J. Siméon, and S. Weinstein, "Constraints for semistructured data and xml," *SIGMOD Rec.*, vol. 30, pp. 47–54, March 2001.

[6] M. Vincent, , M. W. Vincent, and J. Liu, "Functional dependencies for xml," in *In Fifth Asian Pacific Web Conference*, pp. 22–34, Springer, 2003.

[7] M. Vincent and J. Liu, "Checking functional dependency satisfaction in xml," in *Database and XML Technologies* (S. Bressan, S. Ceri, E. Hunt, Z. Ives, Z. Bellahsène, M. Rys, and R. Unland, eds.), vol. 3671 of *Lecture Notes in Computer Science*, pp. 578–579, Springer Berlin / Heidelberg, 2005.

[8] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano, "Repairs and consistent answers for xml data with functional dependencies," in *Database and XML Technologies* (Z. Bellahsène, A. Chaudhri, E. Rahm, M. Rys, and R. Unland, eds.), vol. 2824 of *Lecture Notes in Computer Science*, pp. 238–253, Springer Berlin / Heidelberg, 2003.

[9] S. Flesca, F. Furfaro, S. Greco, and E. Zumpano, "Querying and repairing inconsistent xml data," in *Web Information Systems Engineering – WISE 2005* (A. Ngu, M. Kitsuregawa, E. Neuhold, J.-Y. Chung, and Q. Sheng, eds.), vol. 3806 of *Lecture Notes in Computer Science*, pp. 175–188, Springer Berlin / Heidelberg, 2005.

[10] Z. Tan and L. Zhang, "Improving xml data quality with functional dependencies," in *Database Systems for Advanced Applications* (J. Yu, M. Kim, and R. Unland, eds.), vol. 6587 of *Lecture Notes in Computer Science*, pp. 450–465, Springer Berlin / Heidelberg, 2011.

[11] V. V. Vazirani, *Approximation algorithms*. Springer, 2001.

[12] J. Clark and S. DeRose, "XML path language (XPath) version 1.0," W3C recommendation, W3C, Nov. 1999. http://www.w3.org/TR/1999/REC-xpath-19991116.

[13] M. Klempa, M. Mikula, R. Smetana, M. Švirec, and M. Vitásek, "jinfer xml schema inference framework." `http://jinfer.sourceforge.net/modules/paper.pdf`. Website of the project: `http://jinfer.sourceforge.net`.

[14] "Java architecture for xml binding." `http://jaxb.java.net/`.

List of Tables

List of Figures

Appendix A

Content of CD

The CD is a part of this thesis. It contains text of this thesis, source code of jInfer framework with `FDRepairer` module together with documentation and built jInfer as `nbm` plugins applicable into NetBeans platform. It aslo contains experimental data and results. CD has the following structure:

- *content.txt* - A file with this text
- *text* - A directory with the PDF file containing text of this thesis
- *src* - Source codes of jInfer including `FDRepairer` extension
- *javadoc* - A generated javadoc documentation
- *bin* - A directory with `nbm` plugins of jInfer and FDRepairer
- *datasets* - Experimental data and results

Appendix B

Attachments

Figure B.1: XML Schema for weights

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="Tweight">
  <xs:sequence>
    <xs:element name="path" type="xs:string"/>
    <xs:element name="value" type="xs:decimal"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tweights">
  <xs:sequence>
    <xs:element name="weight" type="Tweight"
        minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="weights" type="Tweights"/>
</xs:schema>
```

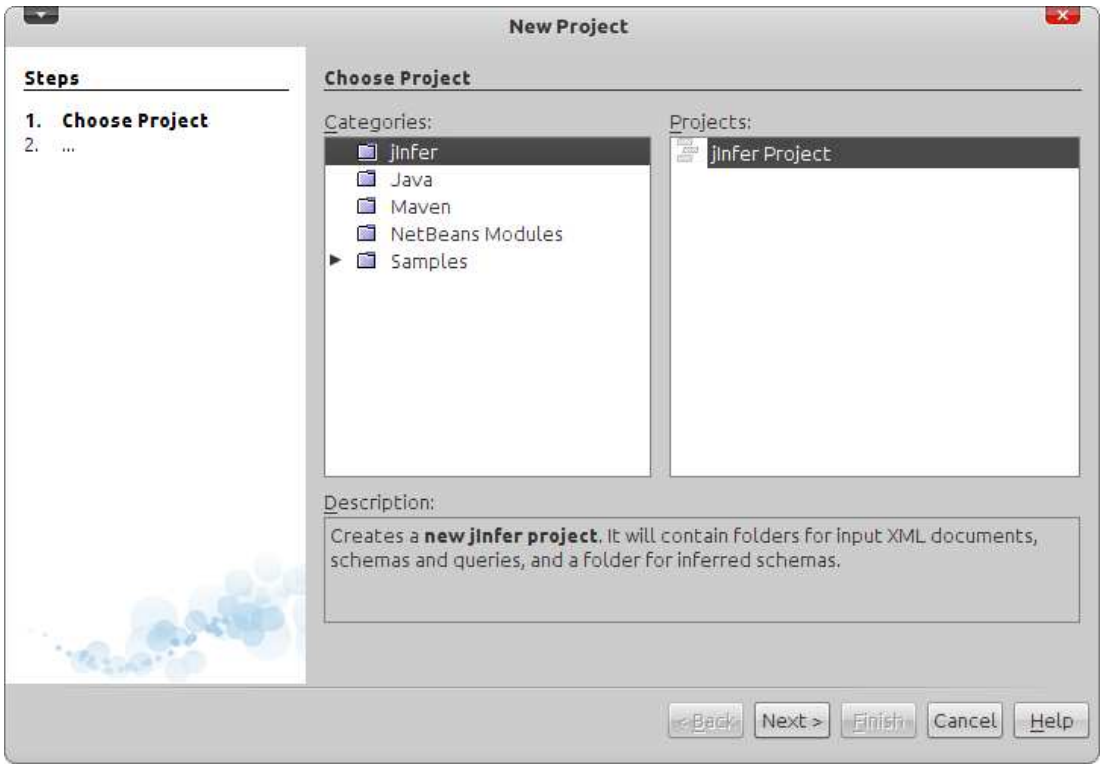
Figure B.2: XML Schema for functional dependencies

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="TleftSidePaths">
  <xs:sequence>
    <xs:element name="path" type="xs:string"
        minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="TrightSidePaths">
  <xs:sequence>
    <xs:element name="path" type="xs:string"
        minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tdependency">
  <xs:sequence>
    <xs:element name="leftSidePaths" type="TleftSidePaths"/>
    <xs:element name="rightSidePaths" type="TrightSidePaths"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="Tdependencies">
  <xs:sequence>
    <xs:element name="dependency" type="Tdependency"
        minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="dependencies" type="Tdependencies"/>
</xs:schema>
```

Appendix C

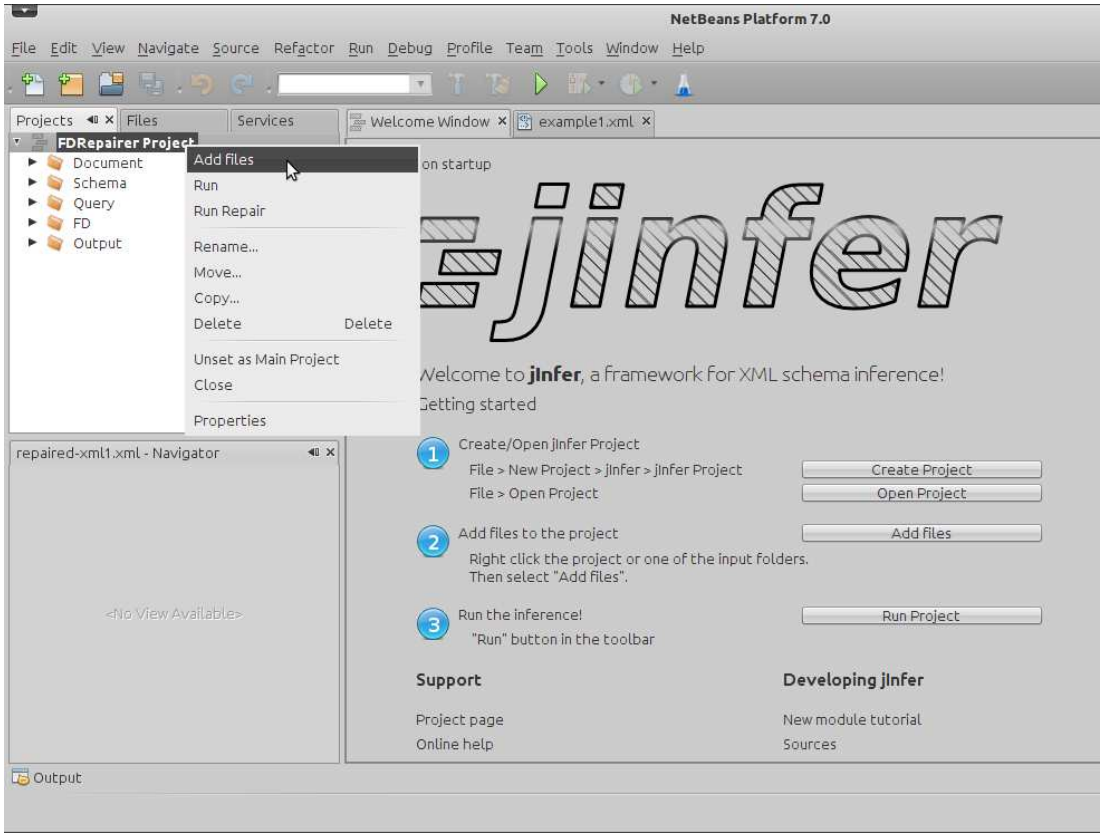
FDRepairer Guide

First, before we start repairing XML data violating functional dependencies, new jInfer project needs to be created:

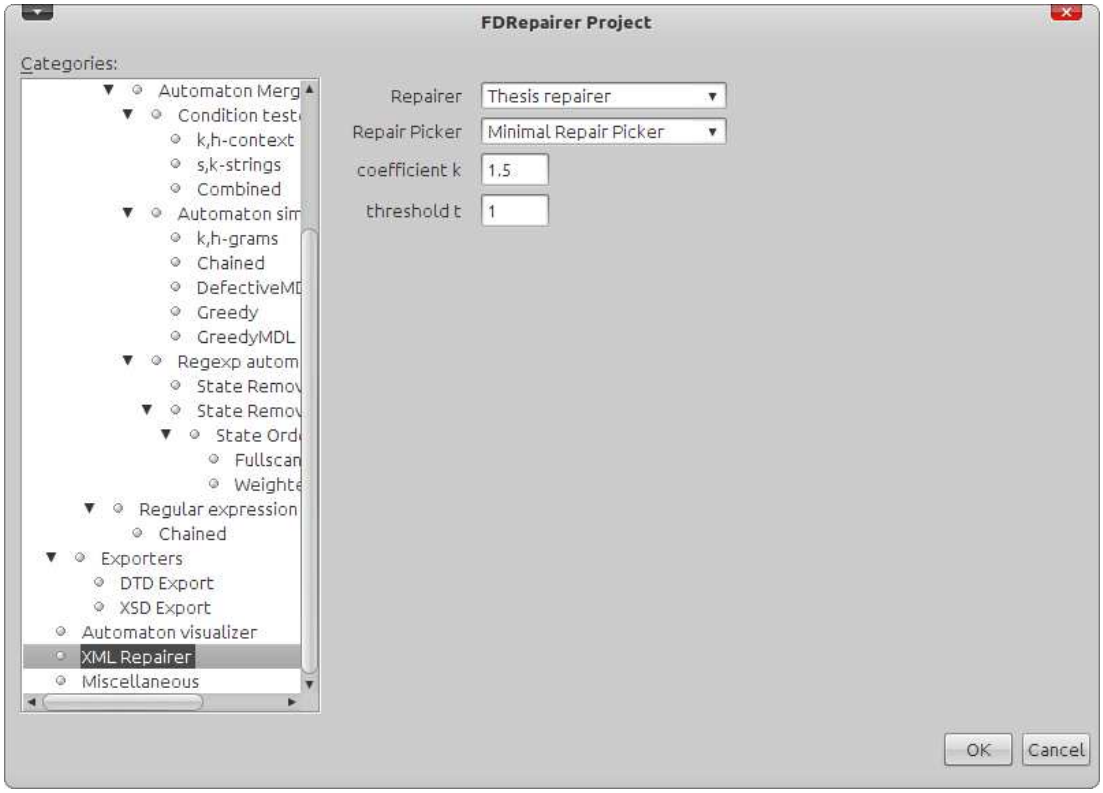


Files with XML data, defined functional dependencies and weights assigned to the nodes needs to be added to the newly created project. Files with functional

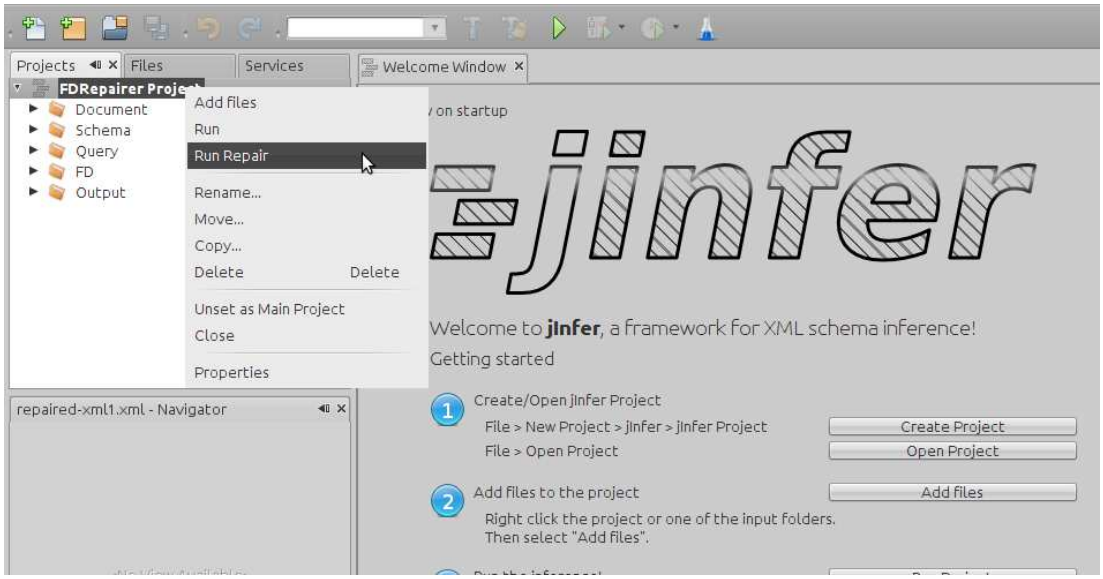
dependencies and weights are xml files with XML Schema shown in Fig. B.2 and B.1. These files appears after adding in the *FD* folder of the jInfer project.



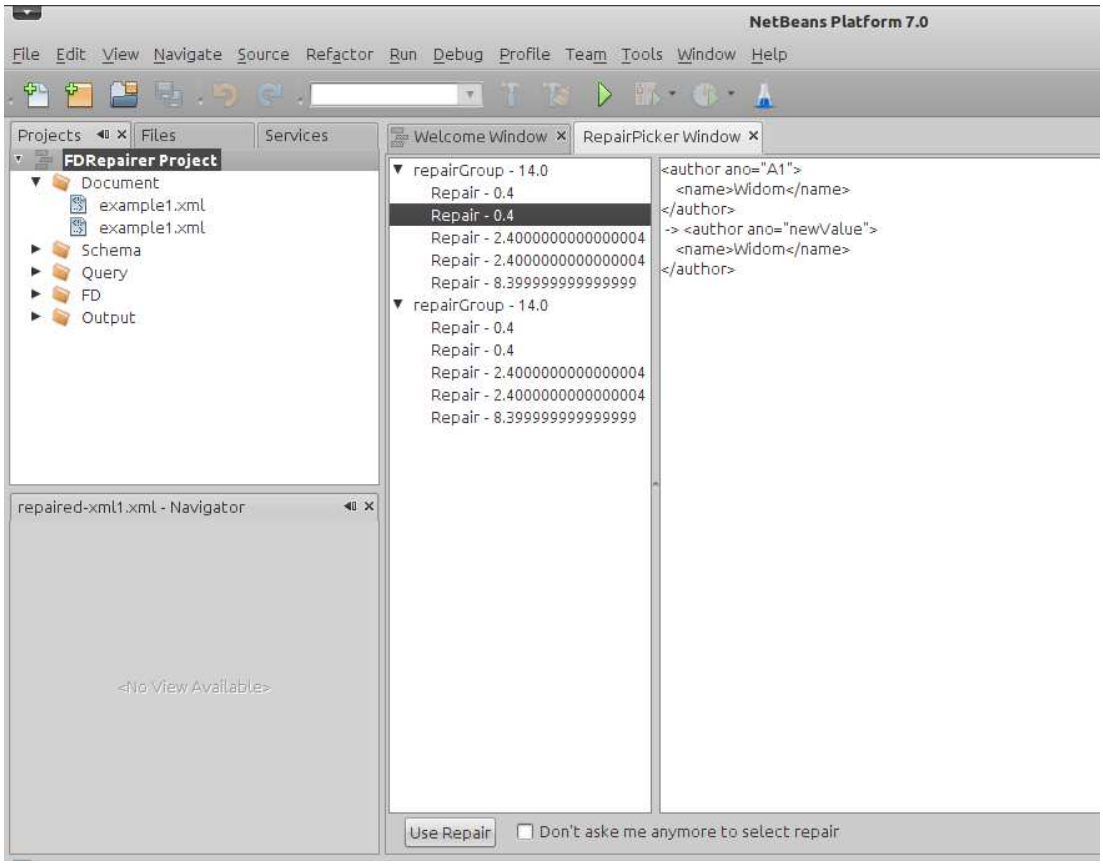
The settings of the `FDRepairer` extension is available in the properties window of the jInfer project accessible through project context menu. In these settings, one could choose between original repairer and repairer proposed in this thesis. For the proposed algorithm is then available to choose between *user interactive* repair picker and *Minimal Repair Picker*. In this place is also possible to set coefficient k and threshold t.



The repair is executed by the *Run Repair* item from the jInfer project context menu:



If the *user interactive* repair picker has been selected, *RepairPicker Window* is shown to the user. In this window the user picks repair candidate, which will be applied to the repaired XML.



After repairing all the violations, repaired XML document is created in the *Output* subfolder of the jInfer project.