Correction of Invalid XML Documents with Respect to Single Type Tree Grammars*

Martin Svoboda and Irena Mlýnková

Department of Software Engineering, Charles University in Prague
Malostranske namesti 25, 118 00 Prague 1, Czech Republic
Contact e-mail: {`svoboda,mlynkova`}`@ksi.mff.cuni.cz`

Abstract. XML documents and related technologies represent a widely accepted standard for managing semi-structured data. However, a surprisingly high number of XML documents is affected by well-formedness errors, structural invalidity or data inconsistencies. The aim of this paper is the proposal of a correction framework involving structural repairs of elements with respect to single type tree grammars. Via the inspection of the state space of a finite automaton recognising regular expressions, we are always able to find all minimal repairs against a defined cost function. These repairs are compactly represented by shortest paths in recursively nested multigraphs, which can be translated to particular sequences of edit operations altering XML trees. We have proposed an efficient algorithm and provided a prototype implementation.

Keywords: XML, correction, validity, grammar, tree.

1 Introduction

XML documents [10] and related standards represent without any doubt an integral part of the contemporary Word Wide Web technologies. They are used for data interchange, sharing knowledge or for storing semi-structured data. However, the XML usage explosion is accompanied with a surprisingly high number of documents involving various forms of errors [5].

These errors can cause that the given documents are not well-formed, they do not conform to the required structure or have inconsistencies in data values. Anyway, the presence of errors causes at least obstructions and may completely prevent successful processing. Generally we can modify existing algorithms to deal with errors, or we can attempt to modify invalid documents themselves.

We particularly focus on the problem of the structural invalidity of XML documents. In other words we assume the inspected documents are well-formed and constitute trees, however, these trees do not conform to a schema in DTD [10] or XML Schema [4], i.e. a regular tree grammar with the expressive power at the level of single type tree grammars [6]. Having a potentially invalid XML

* This work was partially supported by the Czech Science Foundation (GAČR), grants number 201/09/P364 and P202/10/0573.

document, we process it from its root node towards leaves and propose minimal corrections of elements in order to achieve a valid document close to the original one. In each node of a tree we attempt to statically investigate all suitable sequences of its child nodes with respect to a content model and once we detect a local invalidity, we propose modifications based on operations capable to insert new minimal subtrees, delete existing ones or recursively repair them.

Related Work. The proposed framework is based primarily on ideas from [1] and [9]. Authors of the former paper dynamically inspect the state space of a finite automaton for recognising regular expressions in order to find valid sequences of child nodes with minimal distance. However, this traversal is not effective, requires a threshold pruning to cope with potentially infinite trees, repeatedly computes the same repairs and acts efficiently only in the context of incremental validation. Although these disadvantages are partially handled in the latter paper, its authors focused on documents querying, but not repairing.

Next, we can mention an approximate validation and correction approach [11] based on testers and correctors from the theory of program verification. Repairs of data incosistencies like functional dependencies, keys and multivalued dependencies are the subject of [8, 12].

Contributions. Contrary to all existing approaches we consider single type tree grammars instead only local tree grammars. Thus we work both with DTD and XML Schema. Approaches in [1, 11] are not able to find repairs of more damaged documents, we are able to always find all minimal repairs and even without any threshold pruning to handle potentially infinite XML trees. Next, we have proposed much more efficient algorithm following only perspective ways of the correction and without any repeated repair computations. Finally, we have a prototype implementation available at [3] and performed experiments show a linear time complexity depending on a number of nodes in documents.

Outline. In Section 2 we define a formal model of XML documents and schemata as regular tree grammars. Section 3 introduces the entire proposed correction framework and Section 4 concludes this paper.

2 Preliminaries

In this section, we introduce preliminary definitions used in this paper.

2.1 XML Trees

Analogously to [1], we represent XML documents as data trees based on underlying trees with prefix numbering of nodes.

Definition 1 (Underlying Tree). *Let $\mathbb{N}_0^*$ be the set of all finite words over the set of non-negative integers $\mathbb{N}_0$, ϵ be an empty word and . a concatenation. A set $D \subset \mathbb{N}_0^*$ is an* underlying tree *or just* tree*, if the following conditions hold:*

- *D is closed under prefixes, i.e. having a binary* prefix relation *$\preceq$ (where $\forall u, v \in \mathbb{N}_0^*$ we define $u \preceq v$ if $u.w = v$ for some $w \in \mathbb{N}_0^*$) we require that $\forall u, v \in \mathbb{N}_0^*$, $u \preceq v$: $v \in D$ implies $u \in D$.*

– $\forall u \in \mathbb{N}_0^*$, $\forall j \in \mathbb{N}_0$: *if* $u.j \in D$ *then* $\forall i \in \mathbb{N}_0$, $0 \leq i \leq j$, $u.i \in D$.

We say that D is an empty tree*, if $D = \emptyset$. Elements of D are called* nodes*, node ϵ is a* root *node and $LeafNodes(D) = \{u \mid u \in D$ and $\neg \exists i \in \mathbb{N}_0$ such that $u.i \in D\}$ represents a set of* leaf *nodes.*

Given a node $u \in D$ we define $fanOut(u)$ as $n \in \mathbb{N}_0$ such that $u.(n-1) \in D$ and $\neg \exists n' \in \mathbb{N}$, $n' > n-1$ such that $u.n' \in D$. If $u.0 \notin D$, we put $n = 0$. Finally, we define $D_p = \{s \mid s \in \mathbb{N}_0^*, p.s \in D\}$ as a subtree *of D at position p.*

Since we are interested only in elements, we ignore attributes. Data values and element labels are modelled as partial functions on underlying nodes.

Definition 2 (Data Tree). *Let D be an underlying tree, $\mathbb{V}$ a domain for data values and $\mathbb{E}$ a domain of element labels (i.e. set of distinct element names). Tuple $\mathcal{T} = (D, lab, val)$ is a* data tree*, if the following conditions are satisfied:*

– lab *is a labelling function $D \to \mathbb{E} \cup \{\mathtt{data}\}$, where $\mathtt{data} \notin \mathbb{E}$:*
  – *$DataNodes(\mathcal{T}) = \{p \in D \mid lab(p) = \mathtt{data}\}$ is a set of* data nodes.
  – *If $p \in DataNodes(\mathcal{T})$, then necessarily $p \in LeafNodes(D)$.*
– val *is a function $DataNodes(\mathcal{T}) \to \mathbb{V} \cup \{\perp\}$ assigning values to data nodes, where $\perp \notin \mathbb{V}$ is a special symbol representing* undefined values.

Finally, we define $\mathcal{T}_p = (D', lab', val')$ as a data subtree *of $\mathcal{T}$ at position p, where $D' = D_p$ and for each function $\phi \in \{lab, val\}$: if $\phi(p.s)$ is defined, then $\phi'(s) = \phi(p.s)$, where $s \in \mathbb{N}_0^*$.*

Example 1. In Figure 1 we can find sample data tree $\mathcal{T}$ based on an underlying tree $D = \{\epsilon, 0, 0.0, 1, 1.0, 1.1\}$. Values of *lab* function are inside nodes, an implicit tree structure is depicted using edges. Ignoring *val* function this data tree corresponds to an XML fragment: <a><x><d/></x><d><d/><d/></d></a>.

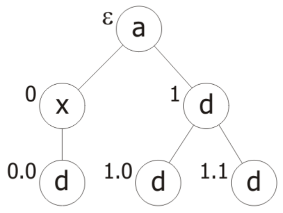


Fig. 1. Sample data tree

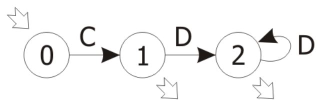


Fig. 2. Glushkov automaton for $C.D^*$

2.2 Regular Expressions

Schemata for XML documents especially restrict nesting of elements through allowed content models. These are based on regular expressions.

Definition 3 (Regular Expression). *Let Σ be a finite nonempty alphabet and $S = \{\varnothing, \epsilon, |, ., ^*, (,)\}$, such that $\Sigma \cap S = \emptyset$. We inductively define a* regular expression *r as a word in $\Sigma \cup S$ and $L(r)$ as an associated language:*

- *$r \equiv \varnothing$ and $L(\varnothing) = \emptyset$. $r \equiv \epsilon$ and $L(\epsilon) = \{\epsilon\}$. $\forall x \in \Sigma$: $r \equiv x$ and $L(x) = \{x\}$.*
- *$r \equiv (r_1|r_2)$ and $L(r_1|r_2) = L(r_1) \cup L(r_2)$,*
- *$r \equiv (r_1.r_2)$ and $L(r_1.r_2) = L(r_1).L(r_2)$,*
- *$r \equiv {r_1}^*$ and $L({r_1}^*) = (L(r_1))^*$,*

where r_1 and r_2 are already defined regular expressions. Having an expression $r = s_1 \dots s_n$, we define $symbols(r) = \{s_i \mid \exists\, i \in \mathbb{N}_0,\ 1 \leq i \leq n,\ s_i \in \Sigma\}$.

Languages of regular expressions can be recognised by finite automata. We use Glushkov automata [2], because they are deterministic for *1-unambiguous* regular expressions required by DTD and XML Schema and without ϵ-transitions.

Definition 4 (Glushkov Automaton). *The Glushkov automaton for a 1-unambiguous regular expression r over an alphabet Σ is a deterministic finite automaton $\mathcal{A}_r = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, where $Q = \Sigma' \cup \{q_0\}$ is a* set of states*, Σ is an* input alphabet*, δ is a partial* transition function *$Q \times \Sigma \to Q$, $q_0 \in Q$ is an* initial state *and $F \subseteq Q$ is a set of* accepting states*.*

Example 2. The Glushkov automaton $\mathcal{A}_r$ for regular expression $r = C.D^*$ over $N_R = \{C, D\}$ is depicted in Figure 2. This automaton has states $Q = \{0, 1, 2\}$, from which $q_0 = 0$ is the initial state and $F = \{1, 2\}$ are accepting states. The transition function δ is represented by directed edges between states.

2.3 Tree Grammars

Adopting and slightly modifying the formalism from [6], we represent schemata in DTD and XML Schema as *regular tree grammars.*

Definition 5 (Regular Tree Grammar). *A* regular tree grammar *is a tuple $\mathcal{G} = (N, T, S, P)$, where:*

- *N is a set of* nonterminal symbols *and T a set of* terminal symbols*,*
- *$S \subseteq N$ is a set of* starting symbols*,*
- *P is a set of* production rules *of the form $[a,\ r \to n]$, where $a \in T$, r is a 1-unambiguous regular expression over N and $n \in N$. Without loss of generality, for each $a \in T$ and $n \in N$ there exists at most one $[a,\ r \to n] \in P$.*

Definition 6 (Competing Nonterminals). *Let $\mathcal{G} = (N,\ T,\ S,\ P)$ be a regular tree grammar and $n_1,\ n_2 \in N$, $n_1 \neq n_2$ are two nonterminal symbols. We say that n_1 and n_2 are* competing *with each other, if there exist two production rules $[a,\ r_1 \to n_1]$, $[a,\ r_2 \to n_2] \in P$ sharing the same terminal symbol a.*

The presence of competing nonterminals makes the processing more complicated, thus we define two main subclasses with less expressive power.

Definition 7 (Tree Grammar Classes). *Let $\mathcal{G} = (N,\ T,\ S,\ P)$ be a regular tree grammar. We say that $\mathcal{G}$ is a* local tree grammar*, if it has no competing nonterminal symbols, and that $\mathcal{G}$ is a* single type tree grammar*, if for each production rule $[a,\ r \to n]$ all nonterminal symbols in r do not compete with each other and starting symbols in S do not compete with each other too.*

As a consequence, we do not need to distinguish between terminal and nonterminal symbols in local tree grammars. DTD schemata correspond to local tree grammars and XML Schema almost to single type tree grammars [6].

Example 3. Following the data tree from Example 1 we can introduce grammar $\mathcal{G}$, where $N = \{A, B, C, D\}$ are nonterminals, $T = \{a, b, c, d\}$ are terminals and $S = \{A, B\}$ are starting symbols. The set P contains these transition rules: $\mathcal{F}_1 = [a, C.D^* \to A]$, $\mathcal{F}_2 = [b, D^* \to B]$, $\mathcal{F}_3 = [c, \varnothing \to C]$ and $\mathcal{F}_4 = [d, D^* \to D]$. Since there are no competing nonterminals, this grammar is a local tree grammar.

2.4 Data Trees Validity

The validity of data trees can be defined via the existence of interpretation trees.

Definition 8 (Interpretation Tree). *Let $\mathcal{T} = (D,\ lab,\ val)$ be a data tree and $\mathcal{G} = (N,\ T,\ S,\ P)$ be a regular tree grammar. An* interpretation tree *of a data tree $\mathcal{T}$ against grammar $\mathcal{G}$ is a tuple $\mathcal{N} = (D,\ int)$, where D is the original underlying tree and* int *is a function $D \to N$ satisfying the following conditions:*

- *$\forall p \in D$ there exists a rule $[a,\ r \to n] \in P$ such that $int(p) = n$, $lab(p) = a$ and $int(p.0).int(p.1) \dots int(p.k) \in L(r)$, where $k = fanOut(p) - 1$.*
- *If $p = \epsilon$ is the root node, then we moreover require $int(p) \in S$.*

Definition 9 (Data Tree Validity). *We say that a data tree $\mathcal{T} = (D,\ lab,\ val)$ is* valid *against a regular tree grammar $\mathcal{G} = (N,\ T,\ S,\ P)$, if there exists at least one interpretation $\mathcal{N}$ of $\mathcal{T}$ against $\mathcal{G}$. Given a node $p \in D$, we say that p is* locally valid*, if $\mathcal{T}_p^{tree}$ is valid against grammar $\mathcal{G}' = (N,\ T,\ N,\ P)$.*

By $L(G)$ we denote a *local*, *single type* or *regular tree language*, i.e. a set of all trees valid against a regular, single type or local tree grammar $\mathcal{G}$ respectively.

Example 4. The data tree from Example 1 represented in Figure 1 is not valid against $\mathcal{G}$ from Example 3, especially because $lab(0) \notin T$ and thus there can not exist any production rule to be used for interpretation of node 0.

Definition 10 (Grammar Context). *Let $\mathcal{G} = (N,\ T,\ S,\ P)$ be a single type tree grammar and $\mathcal{F} = [a,\ r \to n] \in P$. We define $\mathcal{C}_{\mathcal{F}} = (a,\ n,\ N_R,\ P_R,\ map,\ r)$ to be a* general context *of grammar $\mathcal{G}$ for rule $\mathcal{F}$, where:*

- *$N_R = \{x \mid x \in symbols(r)\}$ is a set of* allowed *nonterminal symbols.*
- *$P_R = \{\mathcal{F}' \mid \mathcal{F}' = [a',\ r' \to n'] \in P$ and $n' \in N_R\}$ is a set of* active rules.
- map *is a function $T \to N_R \cup \{\perp\}$ such that $\forall\, \mathcal{F}' = [a',\ r' \to n'] \in P_R$: $map(a') = n'$ and for all other $a' \in T$: $map(a') = \perp$ (where $\perp \notin N$).*

Next, we define a starting context *to be $\mathcal{C}_\bullet = (\perp,\ \perp,\ N_R,\ P_R,\ map,\ r_\bullet)$, where $N_R = S$, both P_R and map are defined standardly and $r_\bullet = (n_1 | \dots | n_s)$ is a* starting *regular expression meeting $s = |S|$, $\forall i \in \mathbb{N}$, $1 \le i \le s$, $n_i \in S$ and $\forall i,\ j \in \mathbb{N}$, $1 \le i < j \le s$, $n_i \ne n_j$. Finally, $\mathcal{C}_\varnothing = (\perp,\ \perp,\ \emptyset,\ \emptyset,\ map,\ r_\varnothing)$ is an* empty context*, where $r_\varnothing = \varnothing$ and map is defined standardly again.*

Example 5. Having production rule $\mathcal{F}_1$ of grammar $\mathcal{G}$ from Example 3, we can derive its context $\mathcal{C}_1 = (a,\ A,\ \{C, D\},\ \{\mathcal{F}_3, \mathcal{F}_4\},\ \{(a, \perp), (b, \perp), (c, C), (d, D)\},\ C.D^*)$. Since $S = \{A, B\}$ are starting symbols, the starting context is equal to $\mathcal{C}_\bullet = (\perp,\ \perp,\ \{A, B\},\ \{\mathcal{F}_1, \mathcal{F}_2\},\ \{(a, A), (b, B), (c, \perp), (d, \perp)\},\ A|B)$.

During the data trees correction, being in each particular node, the local correction possibilities are defined by a corresponding grammar context. However, the content model is represented as a regular expression over nonterminal symbols, thus we first need to transform the labels of existing nodes to nonterminals.

Definition 11 (Node Sequence Imprint). *Let $\mathcal{T} = (D,\ lab,\ val)$ be a data tree and $u = \langle u_1,\ \dots,\ u_k\rangle$ a sequence of nodes for some $k \in \mathbb{N}_0$, where $\forall i \in \mathbb{N}$, $1 \leq i \leq k$, $u_i \in D$. We define an* imprint *of u in context $\mathcal{C} = (a,\ n,\ N_R,\ P_R,\ map,\ r)$ to be sequence $imprint(u) = \langle map(lab(u_1)),\ \dots,\ map(lab(u_k))\rangle$.*

Example 6. Suppose that $u = \langle 0, 1\rangle$ is a sequence of child nodes of the root node in data tree $\mathcal{T}$ from Example 1. Labels of these nodes are $\langle x, d\rangle$. An imprint of u in $\mathcal{C}_1$ from Example 5 is a sequence $\langle \perp, D\rangle$.

3 Corrections

In order to propose a new correction framework, we especially need to introduce a model of allowed data trees transformations and efficient algorithms.

3.1 Edit Operations

First, we define several auxiliary sets of nodes, which become useful in a definition of such allowed transformations, called *edit operations.*

Definition 12 (Auxiliary Node Sets). *Given a tree D and a position $p \in D$, $p \neq \epsilon$, $p = u.i$, $u \in \mathbb{N}_0^*$, $i \in \mathbb{N}$ with $f = fanOut(u)$, we define node sets:*

- *$PosNodes(D) = \{u.i \mid i \in \mathbb{N}_0,\ u.i \notin D,\ u \in D$ and either $i = 0$ or $i > 0$ and $u.(i-1) \in D\}$. If D is an empty tree, then $PosNodes(D) = \{\epsilon\}$.*
- *$ExpNodes(D, p) = \{u.k.v \mid k \in \mathbb{N}_0,\ i \leq k < f,\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.*
- *$IncNodes(D, p) = \{u.(k+1).v \mid k \in \mathbb{N}_0,\ i \leq k < f,\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.*
- *$DecNodes(D, p) = \{u.(k-1).v \mid k \in \mathbb{N}_0,\ i+1 \leq k < f,\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.*

Set *PosNodes* together with the underlying tree represent positions ready for insertions, whereas nodes in *ExpNodes* are transferred to *IncNodes* or *DecNodes* after a performed insertion or deletion respectively.

Example 7. Having a data tree $\mathcal{T}$ from Example 1 and $p = 0$ we can derive $PosNodes(D) = \{0.0.0,\ 0.1,\ 1.0.0,\ 1.1.0,\ 1.2,\ 2\}$, $ExpNodes(D, 0) = \{0,\ 0.0,\ 1,\ 1.0,\ 1.1\}$ and $IncNodes(D, 0) = \{1,\ 1.0,\ 2,\ 2.0,\ 2.1\}$.

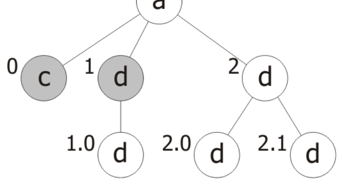
Although we have considered also an internal node insertion/deletion in [7], we focus only on a leaf node insertion/deletion and node renaming in this paper. For the purpose of the following definition of allowed edit operations, we use a symbol $\leftarrow$ as an assignment conditioned by the definability.

Definition 13 (Edit Operations). *An* edit operation e *is a partial function transforming a data tree* $\mathcal{T}_0 = (D_0, lab_0, val_0)$ *into* $\mathcal{T}_1 = (D_1, lab_1, val_1)$*, denoted by* $\mathcal{T}_0 \xrightarrow{e} \mathcal{T}_1$*. Assuming that* $\phi \in \{lab, val\}$*, we define these operations:*

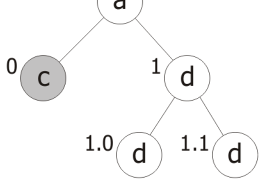
- $e = addLeaf(p, a)$ *for* $p \in D_0 \cup PosNodes(D_0)$*,* $p \neq \epsilon$*,* $p = u.i$*,* $u \in \mathbb{N}_0^*$*,* $i \in \mathbb{N}_0$*,* $u \notin DataNodes(D_0)$ *and* $a \in \mathbb{E}$*:*
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p)] \cup IncNodes(D_0, p) \cup \{p\}$*.*
  - $\forall w \in D_0 \setminus ExpNodes(D_0, p)$*:* $\phi_1(w) \leftarrow \phi_0(w)$*.*
  - $lab_1(p) = a$ *and if* $lab_1(p) = \texttt{data}$*, then* $val_1(p) = \perp$*.*
  - $\forall (u.(k+1).v) \in IncNodes(D_0, p)$*:* $\phi_1(u.(k+1).v) \leftarrow \phi_0(u.k.v)$*.*
- $e = addLeaf(p, a)$ *for* $p = \epsilon$*,* $D_0 = \emptyset$ *and* $a \in \mathbb{E}$*:*
  - $D_1 = \{p\}$*,* $lab_1(p) = a$ *and if* $a = \texttt{data}$*, then* $val_1(p) = \perp$*.*
- $e = removeLeaf(p)$ *for* $p \in LeafNodes(D_0)$*,* $p \neq \epsilon$*,* $p = u.i$*,* $u \in \mathbb{N}_0^*$*,* $i \in \mathbb{N}_0$*:*
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p)] \cup DecNodes(D_0, p)$*.*
  - $\forall w \in D_0 \setminus ExpNodes(D_0, p)$*:* $\phi_1(w) \leftarrow \phi_0(w)$*.*
  - $\forall (u.(k-1).v) \in DecNodes(D_0, p)$*:* $\phi_1(u.(k-1).v) \leftarrow \phi_0(u.k.v)$*.*
- $e = removeLeaf(p)$ *for* $p = \epsilon$*,* $D_0 = \{\epsilon\}$*:*
  - $D_1 = \emptyset$*,* lab_1 *and* val_1 *are not defined anywhere.*
- $e = renameLabel(p, a)$ *for* $p \in D_0$*,* $a \in \mathbb{E}$ *and* $a \neq lab_0(p)$*:*
  - $D_1 = D_0$*.*
  - $\forall w \in [D_0 \setminus \{p\}]$*:* $\phi_1(w) \leftarrow \phi_0(w)$*.*
  - $lab_1(p) = a$ *and if* $a = \texttt{data}$*, then* $val_1(p) = \perp$*.*

Combining edit operations into *edit sequences*, we obtain complex operations capable to insert new subtrees, delete existing ones or recursively repair them.

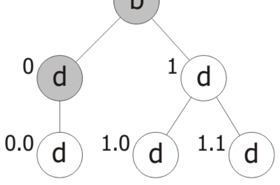
Example 8. Assume that we have edit sequences $\mathcal{X}_1 = \langle addLeaf(0, c), renameLabel(1, d)\rangle$, $\mathcal{X}_2 = \langle renameLabel(0, c), removeLeaf(0.0)\rangle$ and $\mathcal{X}_3 = \langle renameLabel(\epsilon, b), renameLabel(0, d)\rangle$. Applying these sequences separately to data tree $\mathcal{T}$ from Example 1, we obtain data trees depicted in Figures 3(a), 3(b) and 3(c) respectively. Auxiliary node sets for $addLeaf(0, c)$ are derived in Example 7.



(a) Insert-Rename



(b) Rename-Delete



(c) Rename-Rename

Fig. 3. Transforming sample data tree using edit operations

Definition 14 (Cost Function). *Given an edit operation e, we define $cost(e)$ to be a function assigning to e its non-negative* cost. *Having a sequence of edit operations $E = \langle e_1, \ldots, e_k \rangle$ for some $k \in \mathbb{N}_0$, we define $cost(E) = \sum_{i=1}^{k} cost(e_i)$.*

Definition 15 (Data Tree Distance). *Assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ are two data trees and $\mathcal{S}$ is a set of all sequences of update operations capable to transform $\mathcal{T}_1$ to $\mathcal{T}_2$. We define* distance *of $\mathcal{T}_1$ and $\mathcal{T}_2$ to be $dist(\mathcal{T}_1, \mathcal{T}_2) = \min_{E \in \mathcal{S}} cost(E)$.*

Given a regular tree grammar $\mathcal{G}$ and the corresponding regular tree language $L(\mathcal{G})$, we define the distance between a tree $\mathcal{T}_1$ and language $L(\mathcal{G})$ as $dist(\mathcal{T}_1, L(\mathcal{G})) = \min_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}_1, \mathcal{T}_2)$.

The goal of the correction algorithm is to find all minimal repairs, i.e. edit sequences of minimal cost. Although the definition of distances talks about all sequences, the algorithm can clearly inspect only the perspective ones.

Example 9. Assigning unit costs to all edit operations, we can find out that $dist(\mathcal{T}, L(\mathcal{G})) = 2$ for $\mathcal{T}$ from Example 1 and grammar $\mathcal{G}$ from Example 3.

Our algorithm is always able to find all such sequences and because we would like to represent found repairs compactly, we need to abstract away positions from edit operations. Thus we introduce repairing instructions, which need to be translated later on to edit operations over particular nodes.

Definition 16 (Repairing Instructions). *For edit operations $addLeaf(p, a)$, $removeLeaf(p)$ and $renameLabel(p, a)$ with $p \in \mathbb{N}_0^*$ and $a \in \mathbb{E}$ we define associated* repairing instructions ($\texttt{addLeaf}, a$), ($\texttt{removeLeaf}$) *and* ($\texttt{renameLabel}, a$) *respectively. Each repairing instruction is assigned with the corresponding* cost.

3.2 Correction Intents

Assume that we are processing a sequence of sibling nodes in order to correct them. For this purpose we statically investigate the state space of the corresponding Glushkov automaton to find edit sequences transforming the original sequence and all nested subtrees with the minimal cost. Being on a given position, we have already considered the given sequence prefix. The notion of a correction intent involves the assignment for this recursive subtree processing.

Definition 17 (Correction Intent). *Given $\Omega = \{$`correct`, `insert`, `delete`, `repair`, `rename`$\}$ we define a* correction intent *to be a tuple $\mathcal{I} = (t, p, e, v_I, v_E, u, \mathcal{C}, Q_T, Y)$ satisfying the following general constraints:*

- *$t \in \Omega$ is an intent* type, *p is a* base node *and e is a repairing instruction.*
- *$v_I = (s_I, q_I)$: $s_I \in \mathbb{N}_0$ is an* initial stratum *and q_I is an* initial state.
- *$v_E = (s_E, q_E)$: $s_E \in \mathbb{N}_0$ is an* ending stratum *and q_E is an* ending state.
- *$u = \langle u_1, \ldots, u_k \rangle$ is a sequence of nodes to be processed for some $k \in \mathbb{N}_0$.*
- *$\mathcal{C}$ is a grammar context and Q_T is a set of* target states.
- *$Y \subseteq \Omega$ is a set of* allowed types *for nested correction intents.*

Definition 18 (Starting Intent). *Having a data tree $\mathcal{T} = (D, lab, val)$ and a single type tree grammar $\mathcal{G} = (N, T, S, P)$, we define $\mathcal{I}_\bullet = (\texttt{correct}, \perp, \perp, \perp, \perp, u, \mathcal{C}, Q_T, Y)$ to be a* starting correction intent, *where:*

- *If D is not empty, then $u = \langle \epsilon \rangle$, else $u = \langle \rangle$.*
- *$\mathcal{C} = \mathcal{C}_\bullet = (\perp, \perp, N_R, P_R, map, r_\bullet)$ is the starting context.*
- *$Q_T = F$ from the Glushkov automaton $\mathcal{A}_r = (Q, N_R, \delta, q_0, F)$ for $r_\bullet$.*
- *$Y = \Omega \setminus \{\texttt{correct}\}$.*

The data tree correction starts at its root by the starting intent and recursively continues towards leaves by the invocation of nested recursive intents. Authors in [1], contrary to our framework, process data trees from leaves towards a root and attempt to correct only subtrees of locally invalid nodes.

Definition 19 (Recursive Intents). *Let $\mathcal{T} = (D, lab, val)$ be a data tree and $\mathcal{G} = (N, T, S, P)$ a single type tree grammar. Next, assume that $\mathcal{I} = (t, p, e, v_I, v_E, u, \mathcal{C}, Q_T, Y)$ is an already defined correction intent, where $u = \langle u_1, \ldots, u_k \rangle$, $k \in \mathbb{N}_0$, $\mathcal{C} = (a, n, N_R, P_R, map, r)$ and $\mathcal{A}_r = (Q, N_R, \delta, q_0, F)$ is the Glushkov automaton for r. Finally, let $imprint(u) = \langle m_1, \ldots, m_k \rangle$.*

Given a position $v'_I = (s'_I, q'_I)$, where $s'_I \in \mathbb{N}_0$, $s'_I \leq k$, $q'_I \in Q$, we define the following recursive correction intents *$\mathcal{I}' = (t', p', e', v'_I, v'_E, u', \mathcal{C}', Q'_T, Y')$:*

- *If $\texttt{insert} \in Y$: $\forall x \in N_R$: if $\delta(q'_I, x)$ is defined, then we define $\mathcal{I}'$, where:*
  - *Let $\mathcal{F} = [a', r' \to n'] \in P_R$ such that $n' = x$ and $map(a') = x$.*
  - *$t' = \texttt{insert}$, $p' = \perp$, $e' = (\texttt{addLeaf}, a')$, $v'_E = (s'_I, \delta(q'_I, x))$, $u' = \langle \rangle$.*
  - *$\mathcal{C}' = \mathcal{C}_\mathcal{F} = (a', n', N'_R, P'_R, map', r')$ with $\mathcal{A}_{r'} = (Q', N'_R, \delta', q'_0, F')$.*
  - *If $r' \neq \varnothing$, then $Q'_T = F'$, else $Q'_T = \{q'_0\}$. $Y' = \{\texttt{insert}\}$.*

  Suppose that $\langle \mathcal{I}^1, \ldots, \mathcal{I}^j \rangle$ is the longest sequence of correction intents for some $j \in \mathbb{N}_0$, such that $\forall i \in \mathbb{N}$, $1 \leq i < j$, $t^i = \texttt{insert}$, $\mathcal{I}^i$ invokes $\mathcal{I}^{i+1}$ and $\mathcal{I}^j = \mathcal{I}$, $t^j = \texttt{insert}$. We do not allow the previously described intent $\mathcal{I}'$, if $\exists\, i$, $1 \leq i \leq j$: $a^i = a'$ and $n^i = x$ with symbols a^i and n^i from $\mathcal{C}^i$. Finally, we put $ContextChain(\mathcal{I}) = \langle (a^1, n^1), \ldots, (a^j, n^j) \rangle$.
- *If $\texttt{delete} \in Y$ and $s'_I < k$, then we define $\mathcal{I}'$, where:*
  - *$t' = \texttt{delete}$, $p' = u_{s'_I+1}$ and $e' = (\texttt{removeLeaf})$.*
  - *$v'_E = (s'_I + 1, q'_I)$ and $u' = \langle u_{s'_I+1}.0, \ldots, u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1) \rangle$.*
  - *$\mathcal{C}' = \mathcal{C}_\varnothing = (\perp, \perp, \emptyset, \emptyset, map, r_\varnothing)$ with $\mathcal{A}_\varnothing = (Q', N'_R, \delta', q'_0, F')$.*
  - *$Q'_T = \{q'_0\}$ and $Y' = \{\texttt{delete}\}$.*
- *If $\texttt{repair} \in Y$, $s'_I < k$, $m_{k+1} \neq \perp$ and $\delta(q'_I, m_{s'_I+1})$ is defined, then:*
  - *Let $\mathcal{F} = [a', r' \to n'] \in P_R$ such that $n' = m_{s'_I+1}$ and $a' = lab(u_{s'_I+1})$.*
  - *$t' = \texttt{repair}$, $p' = u_{s'_I+1}$, $e' = \perp$ and $v'_E = (s'_I + 1, \delta(q'_I, m_{s'_I+1}))$.*
  - *$u' = \langle u_{s'_I+1}.0, \ldots, u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1) \rangle$.*
  - *$\mathcal{C}' = \mathcal{C}_\mathcal{F} = (a', n', N'_R, P'_R, map', r')$ with $\mathcal{A}_{r'} = (Q', N'_R, \delta', q'_0, F')$.*
  - *If $r' \neq \varnothing$, then $Q'_T = F'$, else $Q'_T = \{q'_0\}$.*
  - *If $r' \neq \varnothing$, then $Y' = \Omega \setminus \{\texttt{correct}\}$, else $Y' = \{\texttt{delete}\}$.*
- *If $\texttt{rename} \in Y$, $s'_I < k$ and $[m_{s'_I+1} = \perp$ or $\delta(q'_I, m_{s'_I+1})$ is not defined$]$, then $\forall x \in N_R$: if $\delta(q'_I, x)$ is defined, then we define $\mathcal{I}'$, where:*
  - *Let $\mathcal{F} = [a', r' \to n'] \in P_R$ such that $n' = x$ and $map(a') = x$.*

- $t' = \texttt{rename}$, $p' = u_{s'_I+1}$ *and* $e' = (\texttt{renameLabel}, a')$.
- $v'_E = (s'_I + 1,\ \delta(q'_I, x))$, $u' = \langle u_{s'_I+1}.0,\ \ldots,\ u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1)\rangle$.
- $\mathcal{C}' = \mathcal{C}_{\mathcal{F}} = (a',\ n',\ N'_R,\ P'_R,\ map',\ r')$ *with* $\mathcal{A}_{r'} = (Q',\ N'_R,\ \delta',\ q'_0,\ F')$.
- *If* $r' \neq \varnothing$, *then* $Q'_T = F'$, *else* $Q'_T = \{q'_0\}$.
- *If* $r' \neq \varnothing$, *then* $Y' = \Omega \setminus \{\texttt{correct}\}$, *else* $Y' = \{\texttt{delete}\}$.

Finally, we define $NestedIntents(\mathcal{I})$ *as a set of all* nested *correction intents* invoked *by* $\mathcal{I}$, *i.e. all* $\mathcal{I}'$ *introduced in this definition and derived from* $\mathcal{I}$.

Example 10. Suppose that within the starting intent $\mathcal{I}_\bullet$ for data tree $\mathcal{T}$ from Example 1 and grammar $\mathcal{G}$ from Example 3 we have invoked a nested `repair` intent $\mathcal{I}$ on base node ϵ. Thus we need to process sequence $u = \langle 0, 1\rangle$ of nodes with labels $\langle x, d\rangle$ in context $\mathcal{C}_1$ from Example 5. Being at a position $(0,0)$, i.e. at stratum 0 (before the first node from u) and in the initial state $q_0 = 0$ of $\mathcal{A}_r$ for $r = C.D^*$ in Example 2, we can derive these nested intents:
$\mathcal{I}_1 = (\texttt{insert},\ \bot,\ (\texttt{addLeaf}, c),\ (0,0),\ (0,1),\ \langle\rangle,\ \mathcal{C}_3,\ Q_3,\ \{\texttt{insert}\})$,
$\mathcal{I}_2 = (\texttt{rename}, 0, (\texttt{renameLabel}, c), (0,0), (1,1), \langle 0.0\rangle, \mathcal{C}_3, Q_3, \Omega\setminus\{\texttt{correct}\})$,
$\mathcal{I}_3 = (\texttt{delete},\ 0,\ (\texttt{removeLeaf}),\ (0,0),\ (1,0),\ \langle 0.0\rangle, \mathcal{C}_\varnothing,\ Q_\varnothing,\ \{\texttt{delete}\})$,
where Q_3 is a set of accepting states for $\mathcal{C}_3$ based on $\mathcal{F}_3$ and $Q_\varnothing$ contains only the initial state of $\mathcal{A}_\varnothing$ for $r = \varnothing$.

The recursive nesting terminates, if the node sequence to be processed is empty and the context allows only an empty model.

3.3 Correction Multigraphs

Correction intents can be viewed as multigraphs with edges corresponding to nested intents and vertices to pairs of sequence positions and automaton states. The idea of these multigraphs is adopted and extended from [9].

Definition 20 (Exploration Multigraph). *Assume that* $\mathcal{T}$ *is a data tree,* $\mathcal{G}$ *a single type tree grammar and* $\mathcal{I} = (t,\ p,\ e,\ v_I,\ v_E,\ u,\ \mathcal{C},\ Q_T,\ Y)$ *a correction intent with* $u = \langle u_1,\ \ldots,\ u_k\rangle$, $k \in \mathbb{N}_0$. *We define an* exploration multigraph *for* $\mathcal{I}$ *to be a directed multigraph* $E(\mathcal{I}) = (V,\ E)$, *where:*

- $V = \{(s, q) \mid s \in \mathbb{N}_0,\ 0 \leq s \leq k,\ q \in Q\}$ *is a set of* exploration vertices.
- $E = \{(v_1,\ v_2,\ \mathcal{I}') \mid \exists \mathcal{I}' \in NestedIntents(\mathcal{I}),\ \mathcal{I}' = (t',\ p',\ e',\ v'_I,\ v'_E,\ u',\ \mathcal{C}',\ Q'_T,\ Y')$ *and* $v_1 = v'_I,\ v_2 = v'_E\}$ *is a set of* exploration edges.

Extending the exploration multigraph and especially its edges with already evaluated *intent repairs* of nested intents, we obtain a correction multigraph.

Definition 21 (Correction Multigraph). *Given an exploration multigraph* $E(\mathcal{I}) = (V,\ E)$ *for correction intent* $\mathcal{I} = (t,\ p,\ e,\ v_I,\ v_E,\ u,\ \mathcal{C},\ Q_T,\ Y)$ *with* u *of size* $k \in \mathbb{N}_0$ *and finite automaton* $\mathcal{A}_r = (Q,\ N_R,\ \delta,\ q_0,\ F)$ *for* r *from context* $\mathcal{C}$, *we define a* correction multigraph *to be a tuple* $C(\mathcal{I}) = (V',\ E',\ v_S,\ V_T)$, *where:*

- $V' = V$ *is a set of* correction vertices.

– $E' = \{(v_1,\, v_2,\, \mathcal{I}',\, \mathcal{R}_{\mathcal{I}'},\, c) \mid (v_1,\, v_2,\, \mathcal{I}') \in E\}$ *is a set of* correction edges, *where* $\mathcal{R}_{\mathcal{I}'}$ *is an* intent repair *for* $\mathcal{I}'$ *and* $c = \mathit{cost}(\mathcal{R}_{\mathcal{I}'})$ *is a* cost *of* $\mathcal{R}_{\mathcal{I}'}$.
– $v_S = (0, q_0)$ *is a* source vertex.
– $V_T = \{v_T \mid v_T = (k, q_T),\, q_T \in Q_T\}$ *is a set of* target vertices.

Example 11. Continuing with Example 10, we can represent all nested intents derived from $\mathcal{I}$ by a correction multigraph $C(\mathcal{I})$ with $v_S = (0, 0)$ and $V_T = \{(2, 1), (2, 2)\}$ in Figure 4. For simplicity, edges are described only by abbreviated intent types (I for `insert`, D for `delete`, R for `repair` and N for `rename`), supplemented by a repairing instruction parameter if relevant and, finally, the complete *cost* of assigned intent repair. Names of vertices are concatenations of a stratum number and an automaton state.

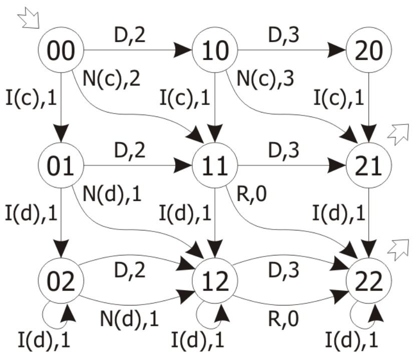


Fig. 4. Sample correction multigraph

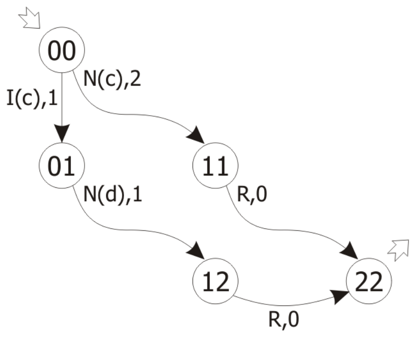


Fig. 5. Sample repairing multigraph

The problem of finding minimal repairs, i.e. the evaluation of correction intents, can now be easily converted to the problem of finding all shortest paths from the source vertex to any target vertex in correction multigraphs.

Definition 22 (Correction Paths). *Let* $C(\mathcal{I}) = (V,\, E,\, v_S,\, V_T)$ *be a correction multigraph. Given* $x,\, y \in V$, *we define a* correction path *from* x *to* y *to be a sequence* $p_{x,y} = \langle e_1,\, \ldots,\, e_n \rangle$ *of correction edges, where* $n \in \mathbb{N}_0$ *is a* length *and:*

– *Let* $\forall k \in \mathbb{N},\, 1 \le k \le n,\, e_k = (v_1^k,\, v_2^k,\, \mathcal{I}^k,\, \mathcal{R}^k_{\mathcal{I}^k},\, c^k)$.
– *If* $n > 0$, *then* $v_1^1 = x$ *and* $v_2^n = y$. *Next,* $\forall k \in \mathbb{N},\, 1 \le k < n$: $v_2^k = v_1^{k+1}$.
– $\neg\, \exists\, j,\, k \in \mathbb{N},\, 1 \le j < k \le n$: $v_1^j = v_1^k$ *or* $v_2^j = v_2^k$ *or* $v_1^j = v_2^j$.

If $x = y$, *then* $p_{x,y} = \langle\rangle$. *Next,* $P_{x,y}$ *is a set of all correction paths from* x *to* y. Path cost *for* $p_{x,y}$ *is defined as* $\mathit{cost}(p_{x,y}) = \sum_{k=1}^{n} c^k$. *We say that* $p_{x,y}$ *is the* shortest path *from* x *to* y, *if and only if* $\neg\, \exists\, p'_{x,y}$ *such that* $\mathit{cost}(p'_{x,y}) < \mathit{cost}(p_{x,y})$. *By* $P^{min}_{x,y}$ *we denote a set of all shortest paths from* x *to* y. *Given nonempty* $Z \subseteq V$, *let* $m = \min_{z \in Z} \mathit{cost}(p_{x,z})$. *Then* $P^{min}_{x,Z} = \{p \mid \exists\, z \in Z,\, p \in P^{min}_{x,z}$ *and* $\mathit{cost}(p) = m\}$ *is a set of all shortest paths from* x *to any* $z \in Z$.

Finally, given a vertex $v \in V$, *we say that* $v \in p_{x,y}$, *if* $\exists\, k \in \mathbb{N},\, 1 \le k \le n$ *such that* $v = v_1^k$ *or* $v = v_2^k$. *Analogously, given an edge* $e \in E$, *we say that* $e \in p_{x,y}$, *if* $\exists\, k \in \mathbb{N},\, 1 \le k \le n$ *such that* $e = e_k$.

Once we have found all required shortest paths, we can forget not involved parts of the correction multigraph. And moreover, these shortest paths themselves constitute the compact structure of the intent repair.

Definition 23 (Repairing Multigraph). *Given a correction intent $\mathcal{I}$ and its correction multigraph $C(\mathcal{I}) = (V, E, v_S, V_T)$, we define a* repairing multigraph *for $\mathcal{I}$ to be a tuple $R(\mathcal{I}) = (V', E', v_S, V_T, c)$ as a subgraph of $C(\mathcal{I})$, where:*

- *$V' = \{v \mid \exists p \in P^{min}_{v_S,V_T}, v \in p\}$ and $E' = \{e \mid \exists p \in P^{min}_{v_S,V_T}, e \in p\}$.*
- *$c = cost(p_{min})$ for some (any) $p_{min} \in P^{min}_{v_S,V_T}$.*

Example 12. A repairing multigraph $R(\mathcal{I})$ for correction intent $\mathcal{I}$ from Example 10 is derived from correction multigraph $C(\mathcal{I})$ and is depicted in Figure 5.

Definition 24 (Intent Repair). *Assume that $R(\mathcal{I}) = (V, E, v_S, V_T, c)$ is a repairing multigraph for $\mathcal{I} = (t, p, e, v_I, v_E, u, \mathcal{C}, Q_T, Y)$. We define an* intent repair *for $\mathcal{I}$ to be a tuple $\mathcal{R}_{\mathcal{I}} = (R_N, R_S, cost)$, where $R_N = e$ is a repairing instruction, $R_S = R(\mathcal{I})$ a repairing multigraph and $cost = cost(e) + c$.*

At the bottom of the recursive intents nesting, the intent repair contains only one shortest path – a path on one vertex, without edges and with zero cost.

3.4 Repairs Translation

Assume that we have processed the entire data tree and thus we have computed all required nested intent repairs. The intent repair for the starting intent stands for all minimal corrections of the given XML document. Our goal is to prompt the user to choose the best suitable edit sequence, however, we first need to gain all these sequences from nested shortest paths, which involves also the translation of repairing instructions to edit operations along these paths.

Definition 25 (Repairing Instructions Translation). *Given a repairing instruction e, we define a* translation *of e to the associated edit operation as $fix(e)$:*

- *If $e = (\texttt{addLeaf}, a)$, then $fix(e) = addLeaf(0, a)$.*
- *If $e = (\texttt{removeLeaf})$, then $fix(e) = removeLeaf(0)$.*
- *If $e = (\texttt{renameLabel}, a)$, then $fix(e) = renameLabel(0, a)$.*

For the purpose of sequences translation, we need three auxiliary functions.

Definition 26 (Auxiliary Translation Functions). *Given a node $u \in \mathbb{N}_0^*$ and a constant $c \in \mathbb{N}_0$, we define $modPre(u, c) = c.u$. If $u \neq \epsilon$, $u = i.v$, $i \in \mathbb{N}_0$, $v \in \mathbb{N}_0^*$, then we define $modAlt(i.v, c) = (i + c).v$ and $modCut(i.v) = v$.*

Once we have defined these functions on nodes, we can extend them on edit operations, edit sequences and, finally, sets of edit sequences. We just straightforwardly transform the node parameter of each particular edit operation, e.g. $modPre(addLeaf(u, a), c) = addLeaf(modPre(u, c), a)$.

Definition 27 (Repairing Multigraph Translation). *Let $R(\mathcal{I}) = (V, E, v_S, V_T, c)$ be a repairing multigraph for $\mathcal{I}$. For each path $p \in P^{min}_{v_S,V_T}$, $p = \langle e_1, \dots, e_m \rangle$, $m \in \mathbb{N}_0$, we define $S_p = \{s^1_p.s^2_p \dots s^m_p \mid \forall i \in \mathbb{N}, 1 \leq i \leq m, s^i_p \in S^i_p\}$, where all particular S^i_p are derived via the successive processing of edges from e_1 to e_m. Thus let $\forall i \in \mathbb{N}, 1 \leq i \leq m, e_i = (v^i_1, v^i_2, \mathcal{I}^i, \mathcal{R}_{\mathcal{I}^i}, c^i)$. Starting with $a_0 = 0$ and $i = 1$, we put $S^i_p = modAlt(fix(\mathcal{R}_{\mathcal{I}^i}), a_{i-1})$ and $a_i = a_{i-1} + x_i$, where: $x_i = 1$ for $t^i \in \{$*`insert`*, *`repair`*, *`rename`*$\}$ and $x_i = 0$ for $t^i =$ *`delete`*. *Finally, we define a* repairing multigraph translation $fix(R(\mathcal{I})) = \bigcup_{p \in P^{min}_{v_S,V_T}} S_p$.

The intent repair translation idea is based on the traversal of all shortest paths stored in the repairing multigraph and the successive processing of their edges leading to the combination of already generated sequences from nested intents and the proper numbering of position parameters in edit operations.

Example 13. Suppose we have paths p_1 and p_2 from $(0, 0)$ to $(2, 2)$ via $(0, 1)$ and $(1, 1)$ respectively in $R(\mathcal{I})$ from Example 12. For p_1 we successively derive $a_0 = 0$, $S^1_p = \{\langle addLeaf(0, c)\rangle\}$, $a_1 = 1$, $S^2_p = \{\langle renameLabel(1, d)\rangle\}$, $a_2 = 2$, $S^3_p = \{\langle\rangle\}$ and $a_3 = 3$. Analogously for p_2: $a_0 = 0$, $S^1_p = \{\langle renameLabel(0, c), removeLeaf(0.0)\rangle\}$, $a_1 = 1$, $S^2_p = \{\langle\rangle\}$ and $a_2 = 2$. Then $S_{p_1} = \{\mathcal{X}_1\}$ and $S_{p_2} = \{\mathcal{X}_2\}$ for $\mathcal{X}_1$ and $\mathcal{X}_2$ from Example 8. Finally, $fix(R(\mathcal{I})) = \{\mathcal{X}_1, \mathcal{X}_2\}$.

Definition 28 (Intent Repair Translation). *We define $fix(\mathcal{R}_\mathcal{I})$ to be an* intent repair translation *for $\mathcal{R}_\mathcal{I} = (R_N, R_S, cost)$ of intent with type t, where:*

- *If $t =$ `correct`, then $fix(\mathcal{R}_\mathcal{I}) = \{modCut(r_S) \mid r_S \in fix(R_S)\}$.*
- *If $t =$ `insert`, then $fix(\mathcal{R}_\mathcal{I}) = \{\langle fix(R_N)\rangle.modPre(r_S, 0) \mid r_S \in fix(R_S)\}$.*
- *If $t =$ `delete`, then $fix(\mathcal{R}_\mathcal{I}) = \{modPre(r_S, 0).\langle fix(R_N)\rangle \mid r_S \in fix(R_S)\}$.*
- *If $t =$ `repair`, then $fix(\mathcal{R}_\mathcal{I}) = \{modPre(r_S, 0) \mid r_S \in fix(R_S)\}$.*
- *If $t =$ `rename`, then $fix(\mathcal{R}_\mathcal{I}) = \{\langle fix(R_N)\rangle.modPre(r_S, 0) \mid r_S \in fix(R_S)\}$.*

Example 14. The correction of a data tree in Figure 1 against a local tree grammar from Example 3 leads to $fix(\mathcal{R}_{\mathcal{I}_\bullet}) = \{\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3\}$, and thus all data trees in Figure 3 represent corrections with cost 2 using $\mathcal{X}_1$, $\mathcal{X}_2$ and $\mathcal{X}_3$ respectively.

3.5 Correction Algorithms

Finally, we need to propose an algorithm for recursive intent repairs computation. A naive algorithm would first initiate the starting intent with the root node and at each level of nesting, it would construct the entire exploration multigraph, then evaluate its edges to acquire nested intent repairs to find shortest paths over them. However, such algorithm would be extremely inefficient.

Definition 29 (Intent Signature). *Assume that $\mathcal{T} = (D, lab, val)$ is a data tree and $\mathcal{I} = (t, p, e, v_I, v_E, u, \mathcal{C}, Q_T, Y)$ is a correction intent with grammar context $\mathcal{C} = (a, n, N_R, P_R, map, r)$. We define a* signature $\mathcal{S}(\mathcal{I})$ *to be a tuple:*

- *If $t =$ `correct`, then $\mathcal{S}(\mathcal{I}) =$ (`correct`).*

– *If* t = `insert`, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{insert}, n, a, ContextChain(\mathcal{I}))$.
– *If* t = `delete`, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{delete}, p)$.
– *If* t = `repair`, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{repair}, p, n)$.
– *If* t = `rename`, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{rename}, p, n, a)$.

First, we in fact do not need to construct and evaluate the entire correction multigraph, we can use the idea of Dijsktra algorithm and directly find shortest paths in a continuously constructed multigraph. Next, using the concept of intent signatures, we can avoid repeated computations of the same repairs. Although two different intents are always different, the resulting intent repair may be the same, e.g. the deletion depends only on a subtree, but not on a particular context.

Algorithm 1: cachingCorrectionRoutine

```
Input  : Data tree $\mathcal{T}$, grammar $\mathcal{G}$, intent $\mathcal{I} = (t, p, e, v_I, v_E, u, \mathcal{C}, Q_T, Y)$.
Output: Intent repair $\mathcal{R}_\mathcal{I}$ for $\mathcal{I}$.
1  $\mathcal{R}_\mathcal{I} \leftarrow$ getCachedRepair($\mathcal{S}(\mathcal{I})$); if $\mathcal{R}_\mathcal{I} \neq \perp$ then return $\mathcal{R}_\mathcal{I}$;
2  Let $u \leftarrow \langle u_1, \ldots, u_k \rangle$, $\mathcal{C} \leftarrow (a, n, N_R, P_R, map, r)$ and $\mathcal{A}_r \leftarrow (Q, N_R, \delta, q_0, F)$;
3  $C(\mathcal{I}) \leftarrow (V \leftarrow \{(0, q_0)\}, E \leftarrow \emptyset, v_S \leftarrow (0, q_0), V_T \leftarrow \{(k, q_T) \mid q_T \in Q_T\})$;
4  $pCost(v_S) \leftarrow 0$; $pPrev(v_S) \leftarrow \emptyset$; $reachedVertices \leftarrow \{v_S\}$; $finalCost \leftarrow \perp$;
5  while $reachedVertices \neq \emptyset$ do
6      $v \leftarrow$ fetchMinimalVertex($reachedVertices$);
7      if $v \in V_T$ and $finalCost = \perp$ then $finalCost \leftarrow pCost(v)$;
8      if $finalCost \neq \perp$ and $finalCost < pCost(v)$ then break;
9      foreach $\mathcal{I}' = (t', p', e', v'_I = v, v'_E, u', \mathcal{C}', Q'_T, Y') \in NestedIntents(\mathcal{I})$ do
10         $\mathcal{R}'_\mathcal{I} = (R_N, R_S, cost) \leftarrow$ cachingCorrectionRoutine($\mathcal{T}, \mathcal{G}, \mathcal{I}'$);
11         if $v'_E \notin V$ then Add correction vertex $v'_E$ into $V$ and $reachedVertices$;
12         Add correction edge $(v, v'_E, \mathcal{I}', \mathcal{R}'_\mathcal{I}, cost)$ into $E$;
13         $c \leftarrow pCost(v) + cost$;
14         if $pCost(v'_E) \neq \perp$ and $pCost(v'_E) = c$ then $p(v'_E) \leftarrow pPrev(v'_E) \cup \{v\}$;
15         else if $pCost(v'_E) > c$ then $pCost(v'_E) \leftarrow c$; $pPrev(v'_E) \leftarrow \{v\}$;
16 $R(\mathcal{I}) \leftarrow$ createRepairingGraph($C(\mathcal{I})$, $finalCost$, $pPrev$);
17 $\mathcal{R}_\mathcal{I} \leftarrow$ createIntentRepair($\mathcal{I}$, $R(\mathcal{I})$); setCachedRepair($\mathcal{S}(\mathcal{I})$, $\mathcal{R}_\mathcal{I}$);
18 return $\mathcal{R}_\mathcal{I}$;
```

The algorithm first detects, whether we have already computed the repair with the same signature (line 1). If not, we initialise the correction multigraph (lines 2-3) and start (line 4) the traversal for finding all shortest paths to any of target vertices (lines 5-15). Finally, we compose the repair structure and store it in the cache under its signature (lines 16-18).

4 Conclusion

We have proposed and formally described a correction framework based on existing approaches and dealing with invalid nesting of elements in XML documents using the top-down recursive processing of potentially invalid data trees and the state space traversal of automata for recognising regular expression languages with the connection to regular tree grammars model of XML schemata.

Contrary to all existing approaches we have considered the class of single type tree grammars instead only local tree grammars. Under any circumstances we are able to find all minimal repairs using the efficient caching algorithm, which follows only the perspective ways of the correction and prevents repeated computations of the same correction intents. This efficiency is supported by performed experiments using the prototype implementation. A direct experimental comparison to other approaches cannot be presented, since these approaches result to different correction qualities and have different presumptions.

However, we do not support neither local transpositions, nor global moves of entire subtrees. In [7] we have considered wider set of edit operations and also corrections of attributes. The framework can also be extended to find not only minimal repairs and the algorithm can be improved to the parallel one.

References

1. B. Bouchou, A. Cheriat, M. H. Ferrari Alves, A. Savary: Integrating Correction into Incremental Validation. In: BDA (2006)
2. C. Allauzen, M. Mohri: A Unified Construction of the Glushkov, Follow, and Antimirov Automata. In: MFCS 2006. pp. 110–121. Springer (2006)
3. Corrector Prototype Implementation, `http://www.ksi.mff.cuni.cz/~svoboda/`
4. H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn: XML Schema Part 1: Structures (Second Edition) (2004), `http://www.w3.org/TR/xmlschema-1/`
5. I. Mlynkova, K. Toman, J. Pokorny: Statistical Analysis of Real XML Data Collections. In: Proceedings of the 13th International Conference on Management of Data (2006)
6. M. Murata, D. Lee, M. Mani, K. Kawaguchi: Taxonomy of XML Schema Languages using Formal Language Theory. ACM Trans. Internet Technol. 5(4), 660–704 (2005)
7. M. Svoboda: Processing of Incorrect XML Data. Master's thesis, Department of Software Engineering, Charles University in Prague, Czech Republic, Malostranske namesti 25, 118 00 Praha 1, Czech Republic (July 2010)
8. S. Flesca, F. Furfaro, S. Greco, E. Zumpano: Querying and Repairing Inconsistent XML Data. In: WISE '05: Proceedings of the 6th International Conference on Web Information Systems Engineering. LNCS, vol. 3806/2005, pp. 175–188. Springer (2005)
9. S. Staworko, J. Chomicky: Validity-Sensitive Querying of XML Databases. In: Current Trends in Database Technology – EDBT 2006, DataX'06. Lecture Notes in Computer Science, vol. 4254/2006, pp. 164–177. Springer (2006)
10. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, J. Cowan: Extensible Markup Language (XML) 1.1 (Second Edition) (2006), `http://www.w3.org/XML/`
11. U. Boobna, M. de Rougemont: Correctors for XML Data. In: Database and XML Technologies. LNCS, vol. 3186/2004, pp. 69–96. Springer (2004)
12. Z. Tan, Z. Zhang, W. Wang, B. Shi: Computing Repairs for Inconsistent XML Document Using Chase. In: Advances in Data and Web Management. LNCS, vol. 4505/2007, pp. 293–304. Springer (2007)