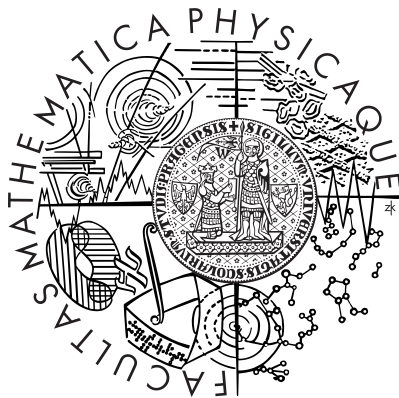
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Martin Svoboda

Processing of Incorrect XML Data

Department of Software Engineering

Supervisor: RNDr. Irena Mlýnková, Ph.D.
Study Program: Software Systems

2010

I would like to thank my supervisor RNDr. Irena Mlýnková, Ph.D. for provided advices, reviews of thesis working versions and text corrections.

Rád bych poděkoval mé vedoucí RNDr. Ireně Mlýnkové, Ph.D. za poskytnuté rady, připomínky k pracovním verzím práce a textové korektury.

I hereby declare that I have elaborated this master thesis on my own and listed all used references. I agree with making this thesis publicly available.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Dále souhlasím se zapůjčováním této práce a jejím zveřejňováním.

In Prague on July 1, 2010 Martin Svoboda
V Praze dne 1. srpna 2010

Contents

1 Introduction **6**

2 Preliminaries **10**
- 2.1 Extensible Markup Language 10
  - 2.1.1 Model of Documents 11
  - 2.1.2 Schema Languages 14
- 2.2 Regular Expressions 15
  - 2.2.1 Glushkov Automaton 16
- 2.3 Regular Tree Grammars 19
  - 2.3.1 Data Trees Validity 20
  - 2.3.2 Grammar Classes 21
- 2.4 Document Type Definition 22
  - 2.4.1 Schema Translation 22
- 2.5 XML Schema 24
  - 2.5.1 Schema Translation 25

3 Analysis **31**
- 3.1 General Aspects 31
- 3.2 Existing Implementations 35
- 3.3 Theoretical Research 36
  - 3.3.1 Incremental Validation and Correction 36
  - 3.3.2 Validity Sensitive Querying 41
  - 3.3.3 Correctors for XML Data 43
  - 3.3.4 Repairs and Consistent Answers 44
  - 3.3.5 Repairing Documents using Chase 45

4 Corrections **46**
- 4.1 Framework Concept 46
- 4.2 Edit Operations 50
  - 4.2.1 Edit Operations for Nodes 52
  - 4.2.2 Edit Operations for Attributes 56
  - 4.2.3 Costs and Sequences of Operations 57
- 4.3 Update Operations 59
  - 4.3.1 Elementary Update Operations 60
  - 4.3.2 Complex Update Operations 61
  - 4.3.3 Costs and Sequences of Operations 65

4.4 Correction Intents . . . 68
4.4.1 Grammar Context . . . 69
4.4.2 Repairing Instructions and Repairs . . . 71
4.4.3 Correction of Data Trees . . . 74
4.5 Correction Multigraphs . . . 83
4.5.1 Exploration Multigraph . . . 83
4.5.2 Correction Multigraph . . . 86
4.5.3 Repairing Multigraph . . . 87
4.5.4 Repairs Construction . . . 89
4.6 Repairs Presentation . . . 90
4.6.1 Repairs for Nodes and Attributes . . . 91
4.6.2 Repairs for Sequences and Intents . . . 94
4.7 Correction Algorithms . . . 98
4.7.1 Naive Correction Algorithm . . . 99
4.7.2 Dynamic Correction Algorithm . . . 103
4.7.3 Caching Correction Algorithm . . . 103
4.7.4 Incremental Correction Algorithm . . . 106

5 Conclusion **120**

A Content of CD **124**

Název práce: Zpracování nekorektních XML dat
Autor: Bc. Martin Svoboda
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: RNDr. Irena Mlýnková, Ph.D.
E-mail vedoucího: irena.mlynkova@ksi.mff.cuni.cz

Abstrakt:
XML dokumenty a technologie reprezentují široce akceptovaný standard pro správu a výměnu semistrukturovaných dat. Překvapivě vysoký počet XML dokumentů však obsahuje chyby dobré formovanosti, strukturální validity nebo nekonzistence dat. Cílem této práce je analýza existujících přístupů vedoucí k návrhu nového korekčního systému. Představený model zahrnuje opravy elementů a atributů vůči jednotypovým stromovým gramatikám. Průchodem stavového prostoru automatu na rozpoznávání regulárních výrazů jsme vždy schopni nalézt všechny minimální opravy. Tyto opravy jsou kompaktně reprezentovány rekurzivními multigrafy, které se dají přeložit do konkrétních sekvencí editačních operací modifikujících datové stromy. Navrženy byly čtyři konkrétní algoritmy doplněné o prototypovou implementaci a experimentální výsledky. Nejvíce efektivní algoritmus heuristicky sleduje pouze perspektivní směry oprav a brání jakýmkoli opakovaným výpočtům.

Klíčová slova: XML, validita, opravy.

Title: Processing of Incorrect XML Data
Author: Bc. Martin Svoboda
Department: Department of Software Engineering
Supervisor: RNDr. Irena Mlýnková, Ph.D.
Supervisor's e-mail address: irena.mlynkova@ksi.mff.cuni.cz

Abstract:
XML documents and related technologies represent widely accepted standard for managing and exchanging semi-structured data. However, surprisingly high number of XML documents is affected by well-formedness errors, structural invalidity or data inconsistencies. The aim of this thesis is the analysis of existing approaches resulting to the proposal of a new correction framework. The introduced model involves repairs of elements and attributes with respect to single type tree grammars. Via the inspection of the state space of an automaton recognising regular expressions, we are always able to find all minimal repairs. These repairs are compactly represented by recursively nested multigraphs, which can be translated to particular sequences of edit operations altering data trees. We have proposed four particular algorithms and provided the prototype implementation supplemented with experimental results. The most efficient algorithm heuristically follows only perspective repair directions and avoids repeated computations using the caching mechanism.

Keywords: XML, validity, corrections.

Chapter 1

Introduction

XML documents [18] and related technologies constitute without any doubt an integral part of the contemporary Word Wide Web. They are used for data interchange, sharing knowledge, other means of communication or for storing semi-structured data. The Web is already not only an extensive repository for unstructured pages, it is a web of data, web of services and its slowly moving towards semantic web or a web of linked data.

Technologies around the XML format are often treated as a standard for development of outlined directions of the web growth. As a consequence, we can be witnessing the XML usage explosion. However, this process is automatically attended with the presence of various forms of incorrect or damaged XML data.

We can detect several miscellaneous reasons causing these unwanted situations and according to [37], the number of documents with errors is surprisingly high. These errors can cause the given documents are not well formed, they do not conform to the required structure or have inconsistencies in data values.

Anyway these errors represent at least obstructions and may completely prevent successful processing. Generally we can modify existing algorithms to cope with these problems, or we can introduce a framework for proposing suitable corrections.

Problem Specification

This thesis focuses on the problem of the structural invalidity of XML documents. This means that we assume the inspected documents are well formed and constitute trees. However, these trees do not conform to the structure declared by a provided schema written in DTD [18] or XML Schema [53] languages.

Schema languages have generally different abilities and expressive power to restrict structure of data trees. Although there exist other languages like Relax NG [43] or Schematron [45], we will work only with schemata at the level of single type tree grammars [38].

These grammars are able to describe allowed XML trees by specifying permitted nesting of elements, their content via regular expressions or required presence of attributes.

The overall purpose of this work is the proposal of a correction framework capable to detect validity violations and propose suitable repairs close to the original trees.

Thesis Objectives

The proposed correction framework is based especially on two existing approaches presented in papers [14, 48].

The former one deals with the incremental correction problem, which is extended by finding local structural corrections. Suppose that we are provided a valid XML document which is later on updated by a sequence of edit operations. The resulting tree may not be valid, thus the proposed algorithm processes the tree in a bottom up manner and whenever it comes across a locally invalid node, it attempts to propose its corrections.

These repairs are gathered during the dynamic traversal of the state space of a finite automaton for recognising words of the language defined by a provided regular expression describing an allowed content model. However, this involves the inefficient generation of word by word, repeatedly invoked identical computations and the inability to find any repair under not rare circumstances.

The latter named paper works with the similar concept of local corrections, however, its primary purpose is to introduce a framework for querying potentially invalid XML databases. Contrary to the previous approach authors do not generate words successively, but statically construct a restoration graph describing all possible suitable repairs at once in a compact structure. Although we can efficiently represent repairs as shortest paths, the introduced set of operations is not sufficient and there are several other related problems.

Considering these two existing approaches we will harness their interesting ideas and deal with determined disadvantages.

We attempt to introduce a correction algorithm that will be always able to efficiently find repairs, will consider higher class from the hierarchy of regular tree grammars and, finally, that will have wider set of allowed trivial transforming operations.

Approach Concept

We consider the class of single type tree grammars. Therefore, we are able to process not only a DTD schema as do both presented existing approaches, but we can also encompass nearly all constructs from XML Schema language. As a consequence we have decided to process the provided invalid XML document from its root towards leaf nodes. This way brings the transparent working with contexts. Furthermore, we take attribute corrections into account too.

Tree modifications are realised via elementary edit operations, through which we can add a new leaf or internal node into a tree, remove an existing one or change a label of a node. Composing these edit operations into sequences we can define more complex update operations for insertions of new subtrees, deletions of existing ones, pushing sibling nodes one level lower or pulling them

one level higher. Each of these transformations is assigned a non-negative cost and using it we can measure distances between trees and grammars.

For each node in a tree we construct a repairing multigraph, which contains all found repairs with a minimal cost. These repairs are compactly represented in a form of shortest paths. However, there are several ways how we can construct these multigraphs. Thus we have introduced four correction algorithms, starting with the naive one and ending with the most efficient one.

As a consequence, we actually do not need to consider all possible correction intents in each situation, but we directly follow only those directions, which seem to be perspective in each moment of the algorithm execution. This means that we are not forced to fully evaluate all possible ways of correction and even more. Using the introduced mechanism of caching we are never forced to compute any repair repeatedly. The algorithm behaves lazily even to the depth, meaning that we are able to incrementally give precision to estimated costs of repairs and follow only the perspective ways starting by the correction of the root node and moving towards leaf nodes.

Finally, the presented approach is able under any circumstances compute all minimal repairs, in which we differ to approach in [14].

Related Work

As a key aspect of any correction framework we can probably determine the selection of operations, through which we are able to transform invalid trees into valid. This problem has a rather close connection to measuring similarities between XML documents or between a document and a schema. The former case is presented in [42, 23], for the latter case there exist papers [41, 49, 54].

We have already listed the most common schema languages for restricting content of XML documents. However, a schema may not always be consistent, for example it may enforce an infinite recursion. This problem is discussed in [4, 35]. Having a consistent schema we can represent it and detect validity using several theoretical models. The most common are probably tree automata [40, 38], or we can choose from automata for extended context free grammars [33] or visibly pushdown automata [2].

The problem of detecting validity is studied for example in [47, 20, 34, 46]. All these papers especially focus on the problem of validating streaming XML data. The special category of validation is the incremental validation, dealing with originally valid documents that are partially updated by the user [5, 12, 15]. Similarly we can put attention to attributes validation as authors in [11] or to validate keys and foreign keys [13].

Authors in [19, 16] extended the incremental validation by proposing repairs for locally invalid nodes and thus introduced the framework for structural repairs. Another work on correcting validity is presented in [44, 9]. Corrections of integrity constraints such as functional dependencies, keys and foreign keys are discussed in [27, 28, 51, 52].

The purpose of correction frameworks, however, needs not be only the generation of suitable repairs. Authors in [48] attempted to extend the standard

model of query evaluation in order to retrieve more information from invalid documents. Finally, we can mention work [1], where the authors introduced a model for working with partially incomplete or missing XML data.

Thesis Organization

Chapter 2 introduces the essential formal definitions for modelling XML trees, working with regular expressions and characterisation of regular tree grammars with introduction of their classes. Finally, we will provide mechanism for translation of DTD or XML Schema constructs into tree grammars.

In Chapter 3 we first discuss several general aspects and questions that are crucial for proposing frameworks working with incorrect XML data. Next, we will shortly introduce existing software projects for correction of HTML documents. The last section of this chapter concerns on the detailed discussion of existing theoretical approaches for correcting structural validity or integrity constraints in XML documents.

The main part of this thesis is represented by Chapter 4, where we completely introduce the formal framework and correction algorithms proposed in this work. The attention will be first put on sets of permitted elementary edit operations and then complex update operations, which are used for XML trees transformations. Next sections will be dedicated to the detailed description of correction strategies represented by correction intents and ways of finding, gathering and composing best suitable repairs using multigraphs model and shortest paths in them. Finally, we will in detail present four correction algorithms with different level of reached efficiency.

Chapter 5 concludes this thesis giving a brief overview of advantages and disadvantages of the proposed framework including potential directions of the future work.

Chapter 2

Preliminaries

The primary goal of this work is to propose a correction routine for XML documents capable of suggesting repairs for such documents, that do not conform to the structural restrictions given by a provided schema written in DTD or XML Schema languages.

Therefore our main interest during the inspection of a provided XML document lies in studying the structure of a tree, the given document represents. We are also interested in the presence of compulsory or optional attributes, but our goal is not to manipulate textual values of elements or attributes. Our general view of XML documents is significantly simplified and we ignore every allowed construct, which is not directly connected with our approach to the correction.

The first section of this chapter presents this adjusted notion of XML documents. In order to provide a formal framework for our proposal, we then follow with two sections providing required knowledge from the domain of regular expressions and regular tree grammars. The chapter is concluded by two another sections, which describe the most important steps of DTD or XML Schema translation into our concept of single type tree grammars.

2.1 Extensible Markup Language

XML documents [17, 18] play without any doubt an important role in the data interchange especially over the Internet. However, we can find several causes, which can lead to various types of damages.

Our main intent is to inspect provided XML documents as trees that should be valid with respect to a given set of structural restrictions. These constraints are in our case defined by tree grammars. Since we are not interested in a plenty of technical details of the XML specification, we will simplify the model of these trees in order to provide more suitable framework adjusted to our concern.

2.1.1 Model of Documents

The main idea of the construction of our XML trees is based on the prefix numbering schema over the set of non-negative integers. Trees themselves are in fact not trees, but sets of nodes. These nodes are numbered by the mentioned schema and thus we can sense the tree itself implicitly due to the numbering. Before we can formally introduce this notion, we first need to define several required string operations and relations.

Processing of Strings

The following definition introduces the concatenation and Kleene star operations over strings.

Definition 2.1 (String Operations)**.** *Let* $u = u_1.u_2 \ldots u_m$ *and* $v = v_1.v_2 \ldots v_n$ *be two words over an alphabet* X*, i.e.* $u_i \in X$ *for* $\forall i \in \mathbb{N}$, $1 \leq i \leq m$ *and* $v_j \in X$ *for* $\forall j \in \mathbb{N}$, $1 \leq j \leq n$ *and some* $m, n \in \mathbb{N}_0$*. We suppose that* ϵ *is considered to be a special symbol for the empty word.*

We define string concatenation *by* $u.v = u_1 \ldots u_m.v_1 \ldots v_n$*.*

Assume that $S \subseteq X$ *is a set of symbols from an alphabet* X*. Given* $S_0 = \{\epsilon\}$ *and inductively* $S_{i+1} = \{v.s \mid v \in S_i \text{ and } s \in S\}$ *for* $i \in \mathbb{N}_0$*, we define* S^* $= \bigcup_{i \in \mathbb{N}_0} S_i$ *as a set of all words over symbols from* S*.*

Let L *be a set of words over an alphabet* X*. Given* $L_0 = \{\epsilon\}$ *and inductively* $L_{i+1} = \{v.s \mid v \in L_i \text{ and } s \in L\}$ *for* $i \in \mathbb{N}_0$*, we analogously define* L^* $= \bigcup_{i \in \mathbb{N}_0} L_i$ *as the smallest set which contains* ϵ *and which is closed under the string concatenation operation.*

In order to easily work with our model of XML trees, we need to define two essential relations. The first one will be used to implicitly construct the structure of XML trees and the second one orders nodes of such trees built over the prefix numbering schema.

Definition 2.2 (Prefix Relation)**.** *Let* L *be a language over an alphabet* X*, i.e.* L *is a set of words over* X*. We define* prefix relation $\preceq$ *as a binary relation in* L*. For* $u, v \in L$ *we say that* $u \preceq v$ *if* $u.w = v$ *for some* $w \in L$*.*

A set $D \subseteq L$ *is* closed under prefixes*, if for* $\forall u, v \in L$ *such that* $u \preceq v$*,* $v \in D$ *implies* $u \in D$*.*

The relation $\preceq$ is reflexive, antisymmetric and transitive.

Definition 2.3 (Partial order relation)**.** *Assume that* $u = u_1.u_2 \ldots u_m$*,* $v = v_1.v_2 \ldots v_n \in \mathbb{N}_0^*$ *for some* m*,* $n \in \mathbb{N}_0$*. We define the partial order relation* $\leq$ *for words over an alphabet* $\mathbb{N}_0$ *this way:* $u \leq v$ *if and only if* $\exists c \in \mathbb{N}_0$*,* $0 \leq c \leq min(m, n)$*,* $\forall k \in \mathbb{N}$*,* $1 \leq k \leq c$*:* $u_k = v_k$ *and* $u_{c+1} \leq v_{c+1}$ *in case of* $c < min(m, n)$ *or* $m \leq n$ *if* $c = min(m, n)$*.*

Underlying Trees

The underlying tree, simply called a tree, is a structure, which formally corresponds to a set of words over an alphabet of non-negative integers, but using the prefix relation we can implicitly view such set as an ordinary tree with one designated node called root.

Definition 2.4 (Tree). *Let $\mathbb{N}_0^*$ be the set of all finite words over the set of non-negative integers $\mathbb{N}_0$. A set $D \subset \mathbb{N}_0^*$ is a* tree*, if both following conditions hold:*

- *D is closed under prefixes.*
- *For $\forall u \in \mathbb{N}_0^*$ and $\forall j \in \mathbb{N}_0$, if $u.j \in D$ then for $\forall i \in \mathbb{N}_0$, $0 \leq i \leq j$, also $u.i \in D$.*

Elements of D are called nodes *and references also as* positions *or* addresses*. We say that D is an* empty tree*, if $D = \emptyset$. Otherwise the given tree must contain at least the* root *node, i.e. node ϵ.*

The set of leaves *of D is defined as $LeafNodes(D) = \{u \mid u \in D$ and $\neg \exists i \in \mathbb{N}_0$ such that $u.i \in D\}$.*

Given a node $u \in D$ we define a fan-out *of node u ($fanOut(u)$) 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 depth *of a node to be a function $D \to \mathbb{N}$ such that $depth(\epsilon) = 1$ and for $\forall u \in D$, $u \neq \epsilon$, $u = v.i$ for some $v \in \mathbb{N}_0^*$ and $i \in \mathbb{N}_0$: $depth(u) = depth(v) + 1$.*

For technical reasons we also need to introduce the notion of a subtree, since it is not true, that using the previous definition we can follow the widespread idea automatically considering a subtree as a tree. This problem can be easily rectified by trimming addresses of nodes.

Definition 2.5 (Subtree). *Let D be a tree and $p \in D$ its node. We denote by D_p the* subtree *whose root is at position p, i.e. $D_p = \{r \mid r \in D$ and $\exists s \in \mathbb{N}_0^* : r = p.s\}$. Node p stands for a* root *of the given subtree.*

Since D_p is not a tree, we define D_p^{tree} as a tree resulting from corresponding subtree D_p. Formally $D_p^{tree} = \{s \mid s \in \mathbb{N}_0^*$ and $p.s \in D_p\}$.

Data Trees for Modeling XML

We consider XML documents as trees that are ordered and unranked. These trees are constituted from nodes representing elements and nodes representing data values located in elements. Constructs like processing instructions, namespaces, entities or notations are completely ignored in our framework. Attributes are modelled using a function over element nodes, thus attributes do not directly influence the structure of a tree.

We assume that names for elements and attributes, as well as data values of attributes and textual values of elements, do not play an important role in our correction approach, thus we will always suppose, that we have three special domains $\mathbb{E}$, $\mathbb{A}$ and $\mathbb{V}$. The former one is a set of all names for elements, the latter one stands for a set of all attribute names and, finally, the last domain is a set of all data values for both attributes and data nodes.

Definition 2.6 (Data Tree)**.** *Let D be a tree, $\mathbb{V}$ a domain for data values, $\mathbb{E}$ a domain of element labels (i.e. set of distinct element names) and analogously $\mathbb{A}$ a domain of attribute names.*

Tuple $\mathcal{T} = (D, lab, val, att)$ is a data tree*, if all following conditions are satisfied:*

- lab *is a labelling function $D \to \mathbb{E} \cup \{\texttt{data}\}$, where $\texttt{data} \notin \mathbb{E}$ is a special symbol for labelling data nodes.*
  - Data *nodes are those nodes $p \in D$, for which $lab(p) = \texttt{data}$.*
  - *If p is a data node, then $p \in LeafNodes(D)$.*
  - *The set of all data nodes in D is denoted by $DataNodes(D)$.*
- val *is a partial function $D \to \mathbb{V} \cup \{\perp\}$ assigning values to data nodes, where $\perp \notin \mathbb{V}$ is a special symbol for marking an undefined value. This function is not defined on nodes that are not data nodes.*
- att *is a partial function assigning a set of pairs of the form $(name, value)$ to $p \in D$, where $name \in \mathbb{A}$ and $value \in \mathbb{V} \cup \{\perp\}$.*
  - *This function is not defined on data nodes. In other cases the function is defined, but the set of pairs may be empty.*
  - *Whenever (n_1, v_1), $(n_2, v_2) \in att(p)$ for n_1, $n_2 \in \mathbb{A}$, v_1, $v_2 \in \mathbb{V}$ and any $p \in D$, then either $n_1 \neq n_2$ or concurrently $n_1 = n_2$ and $v_1 = v_2$.*

Finally, we define functions labelsDomain *and* attsDomain *returning active domains for node labels and attribute names used in a given data tree. Formally $labelsDomain(D) = \{e \mid e \in \mathbb{E}$ and $\exists p \in D,\ lab(p) = e\}$ and $attsDomain(D) = \{a \mid a \in \mathbb{A}$ and $\exists p \in D,\ \exists v \in \mathbb{V},\ (a, v) \in att(p)\}$.*

XML documents are modelled using data trees. Each data tree is in fact an ordinary underlying tree with the prefix numbering, which is enriched by three functions representing labels of element nodes, values of data nodes and sets of existing attributes assigned to particular element nodes.

Definition 2.7 (Data Subtree)**.** *Given a data tree $\mathcal{T} = (D, lab, val, att)$ and a node $p \in D$, we define $\mathcal{T}_p = (D_p, lab_p, val_p, att_p)$ to be a data subtree of the tree $\mathcal{T}$, where:*

- *D_p is a subtree of D rooted at position p,*

- *for each function* $f \in \{lab, val, att\}$*: if* $f(p.u)$ *is defined, then* $f_p(p.u) = f(p.u)$*, where* $u \in \mathbb{N}_0^*$.

Analogously to the underlying subtrees, we define $\mathcal{T}_p^{tree} = (D_p^{tree},\ lab_p^{tree},\ val_p^{tree},\ att_p^{tree})$ *to be a data tree resulting from the data subtree* $\mathcal{T}_p$*, where:*

- D_p^{tree} *is a tree resulting from the subtree* D_p^{tree}*,*
- *for each function* $f \in \{lab, val, att\}$*: if* $f_p(p.u)$ *is defined, then* $f_p^{tree}(u) = f_p(p.u)$*, where* $u \in \mathbb{N}_0^*$.

The definition of the data subtree directly follows the idea of the subtree of its underlying tree.

Definition 2.8 (Data Trees Equivalence)**.** *Assume that* $\mathcal{T}_1 = (D_1,\ lab_1,\ val_1,\ att_1)$ *and* $\mathcal{T}_2 = (D_2,\ lab_2,\ val_2,\ att_2)$ *are two data trees. We define a relation* equivalence $\equiv$ *on data trees:* $\mathcal{T}_1 \equiv \mathcal{T}_2$ *if and only if* $D_1 = D_2$*,* $lab_1 = lab_2$*,* $val_1 = val_2$ *and* $att_1 = att_2$*. In such case we say, that* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are* equivalent.

Two data trees are equivalent, if all their components are equivalent. It is not surprising, that the relation $\equiv$ on data trees is reflexive, symmetric and transitive.

Data Trees Construction

In our correction model we suppose, that XML documents to be corrected are well formed. This generally means, that we can assume the well-formedness as it is defined in [18], but we can relax it only on constructs, that are accounted in our model. Thus we only consider restrictions on elements, attributes and generally on the structure of an XML tree.

Having such XML document, we can easily construct its data tree using for example a SAX parser [36] or parsers based on [25]. Processing the tree from top to down, we create for each element a new node in a constructed data tree. Derived functions over nodes are easy to define and all other content of the given XML document is ignored.

2.1.2 Schema Languages

We start with a short overview of a several languages for creating specifications of schemata the given XML documents should conform to.

- **Document Type Definition** [18]. The specification of this language is an integral part of the XML specification itself. It provides basic ways how to restrain the structure of documents, describing especially the allowed nesting of elements or sets of compulsory or optional attributes.
- **XML Schema** [53]. This schema language can be considered as an extension of DTD. Ignoring constructs that we do not handle in this thesis, we have for example better possibilities to define content model of

elements. The expressive power is approximately at the level of single type tree grammars. The own schema is written as an ordinary XML document, which can be concurrently treated as an advantage and disadvantage.

- **Relax NG** [43]. Another example of a schema language for XML, which is based on patterns. It introduces an XML syntax and a compact syntax, has an extended support for restriction of element content and can be used in conjunction with data types from XML Schema.

- **Schematron** [45]. This framework allows creating schemata, that are not built on tree grammars. Its concept is based on finding patterns in parsed documents and using two main constructs assert and report, we can specify restrictions the given documents should respect.

In this thesis only DTD an XML Schema languages will be considered and even only on the level of single type tree grammars, as it will be explained later in this chapter.

2.2 Regular Expressions

The basic way how we can restrain structure of an XML tree is to enforce only allowed sequences of child nodes for nodes in a tree. This can be done using regular expressions.

Definition 2.9 (Regular Expression)**.** *Let Σ be a finite nonempty alphabet and $S = \{\varnothing, \epsilon, |, ., ^*, (,)\}$ are special symbols such that $\Sigma \cap S = \emptyset$. We inductively define a* regular expression *over Σ as a word in alphabet $\Sigma \cup S$:*

- *Each of the following is a* primitive *regular expression:*
  - $\varnothing$
  - ϵ
  - x *for* $\forall x \in \Sigma$

- *Assume that r_1 and r_2 are regular expressions that have already been defined. Then each of the following is a* compound *regular expression:*
  - $(r_1|r_2)$
  - $(r_1.r_2)$
  - ${r_1}^*$

The set of all regular expressions over Σ is denoted by $RE(\Sigma)$.

Having an expression $r = r_1, \ldots, r_n$ as a word over $\Sigma \cup S$ for some $n \in \mathbb{N}_0$, we define symbols to be a function $RE(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ such that $symbols(r) = \{r_i \mid r_i \in \Sigma \text{ for } i \in \mathbb{N}_0,\ 1 \leq i \leq n\}$.

In order to simplify the notation of regular expressions, we usually adopt several conventions. We can for example omit outer parentheses or parentheses around . and | due to the associativity. Sometimes we omit . and define priorities of operators.

Sometimes it is useful to extend the basic set of operators with ? or $^+$. The former one stands for $r? = (r|\epsilon)$, the latter one $r^+ = (r.r^*)$. In our approach we will not integrate these operators into the regular expression definition and if needed, we will translate these operators into the basic set using provided formulae.

Definition 2.10 (Regular Expression Evaluation)**.** *Let r be a regular expression over an alphabet Σ. We define the value of r as a regular language $L(r)$ inductively:*

- *Evaluation of primitive regular expressions:*
  - *If $r \equiv \varnothing$, then $L(r) = \emptyset$*
  - *If $r \equiv \epsilon$, then $L(r) = \{\epsilon\}$*
  - *If $r \equiv x$ for $x \in \Sigma$, then $L(r) = \{x\}$*
- *Evaluation of compound regular expressions:*
  - *If $r \equiv (r_1|r_2)$, then $L(r) = L(r_1) \cup L(r_2)$*
  - *If $r \equiv (r_1.r_2)$, then $L(r) = L(r_1).L(r_2)$*
  - *If $r \equiv {r_1}^*$, then $L(r) = (L(r_1))^*$*

Each regular expression thus represents a regular language over a given alphabet. Both DTD and XML Schema use regular expressions to limit allowed sequences of child nodes in XML trees. If the sequence of child nodes belongs to the language defined by a provided regular expression called *content model*, it is treated as an allowed sequence. The only problem is that we cannot generally build these words and expressions over an alphabet of element names. This topic will be in detail discussed in Section 2.3.

2.2.1 Glushkov Automaton

We have already defined regular expressions and their value. Now we will focus the problem of recognising words of a language generated by a given regular expression. Assume that a schema for a given XML document defines a content model for a specified node, i.e. we are provided a regular expression and we have to decide, whether the word created from the sequence of child nodes belongs to the language of such regular expression.

For this purpose we can utilise finite automata. The nondeterministic variant is used for example in [48], the deterministic variant in [5, 20]. In our work we have chosen the Glushkov automaton. States of this automaton correspond to positions of a given regular expression and transitions connect those positions that can be consecutive in words generated by that expression.

Before we can introduce the notion of the Glushkov automaton, we need to define positions in a regular expression. This auxiliary notion allows us to define 1-unambiguous regular expressions and we will also need it in order to construct the required automaton.

Definition 2.11 (Marked Regular Expression). *Let r be a regular expression over an alphabet Σ. We define* marked *regular expression r' as a regular expression over an alphabet Σ', where:*

- $\Sigma' = \{x_i \mid \forall x \in \Sigma$ *and* $\forall i \in \mathbb{N}, 1 \leq i \leq count(r, x), x_i$ *is a symbol x with a subscript i*$\}$, *where $count(r, x)$ is the overall number of symbol x occurrences in the expression r.*
- *Items of Σ' are called* marked *symbols. We define $sym(x')$ as a function $\Sigma' \to \Sigma$ by the formula $sym(x_i) = x$.*
- *We define r' to be the adjusted expression r, where all occurrences of symbol x are replaced by x_i for $\forall x \in \Sigma$ and some $x_i \in \Sigma'$ only ensuring, that whenever x_j, $x_k \in \Sigma'$ are two different occurrences in the expression r', necessarily $j \neq k$.*

If we have a regular expression, a corresponding marked regular expression is derived from the original one by adding subscripts to all occurrences of symbols from the original alphabet in order to distinguish occurrences of the same original symbols. This is in other words the last condition in the previous definition.

Although the particular algorithm of indexing symbols is not presented, we only have to ensure uniqueness of all symbols. One way how to add such subscripts to symbols is to create a continuous numbering sequence from natural numbers for each original symbol separately, marking that symbols from the left to the right.

It is easy to see, that a marked regular expression can be seen as a regular expression over a marked alphabet.

Definition 2.12 (1-unambiguous Regular Expressions). *Assume that r is a regular expression over an alphabet Σ and r' is a derived marked regular expression over an alphabet Σ'. We say that r is* 1-unambiguous *if and only if for all words $u, v, w \in (\Sigma')^*$ and all marked symbols $x, y \in \Sigma'$ such that $x \neq y$ whenever $u.x.v$, $u.y.w \in L(r')$, then $sym(x) \neq sym(y)$.*

If a regular expression is 1-unambiguous, the processed input string, which is to be recognised, can be uniquely matched only with a look ahead of one character.

Definition 2.13 (Glushkov Automaton). *The Glushkov automaton for a regular expression r over an alphabet Σ is a nondeterministic finite automaton $\mathcal{A}_r = (Q, \Sigma, \delta, q_0, F)$, where*

- $Q = \Sigma' \cup \{q_0\}$ *is a set of states,*

- *Σ is the original alphabet standing for input symbols,*
- *δ is a partial* transition *function $Q \times \Sigma \rightarrow \mathcal{P}(Q)$ such that:*
  - $\delta(q_0, a) = \{x \mid x \in first(r) \text{ and } sym(x) = a\}$ *for* $\forall a \in \Sigma$,
  - $\delta(q, a) = \{x \mid x \in follow(r, q) \text{ and } sym(x) = a\}$ *for* $\forall q \in \Sigma'$ *and* $\forall a \in \Sigma$,
- *$q_0 \in Q$ is the initial state,*
- *$F \subseteq Q$ is a set of output states such that:*
  - *if $\epsilon \in L(r)$ then $F = last(r) \cup \{q_0\}$,*
  - *otherwise $F = last(r)$.*

Having the transition function δ we can define an extended transition function *$\delta^*\colon Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$, where $\forall w \in \Sigma^*$ we define $\delta^*(q, w)$ to be:*

- *If $w = \epsilon$, then $\delta^*(q, w) = \{q\}$.*
- *If $w = a.v$ for $a \in \Sigma$ and $v \in \Sigma^*$, then $\delta^*(q, w) = \bigcup_{q' \in \delta(q,a)} \delta^*(q', v)$.*

For each $q \in Q$ we define $reachable(q) = \{q' \mid \exists w \in \Sigma^*,\ q' \in \delta^*(q, w)\}$ to be a set of all states reachable *from q.*

Auxiliary functions first, follow *and* last *are defined as follows:*

- first *is a function $RE(\Sigma) \rightarrow \mathcal{P}(\Sigma')$ assigning a set of all marked symbols, that can appear as the first symbol in some word in language $L(r')$ for a given regular expression r,*
- follow *is a function $RE(\Sigma) \times \Sigma' \rightarrow \mathcal{P}(\Sigma')$ assigning a set of all marked symbols, that can appear immediately after a given symbol in some word in language $L(r')$,*
- last *is a function $RE(\Sigma) \rightarrow \mathcal{P}(\Sigma')$ assigning a set of all marked symbols, that can appear as the last symbol in some word in language $L(r')$.*

It can be shown, that a language recognised by the Glushkov automaton constructed for a given regular expression corresponds to the language of such regular expression. Moreover the Glushkov automaton is deterministic if and only if the given regular expression is 1-unambiguous. Therefore, if we have 1-unambiguous regular expression, the transition function of the corresponding automaton assigns to each state and symbol pair at most one state.

2.3 Regular Tree Grammars

Although XML documents are naturally represented as ordinary strings, we should rather comprehend this form of representation only as a serialisation of an XML tree. This is because XML data in fact stand for semi structured data, not a monolithic string.

From the same point of view we should use tree grammars to describe allowed XML trees instead of using context free grammars. However, this string approach is frequently used for example by [47, 5, 33]. Similarly another string approach is based on visibly pushdown languages and automata [2, 34, 46] as well as we can find papers using regular hedge grammars [54].

The purpose of this section is to introduce a notion of a regular tree grammar and its derived classes. Even though we can base our definitions on [38], the provided formal system is modified in order to support also description of attributes.

Definition 2.14 (Regular Tree Grammar)**.** *A* regular tree grammar *is a tuple* $\mathcal{G} = (N, T, A, S, P)$*, where:*

- N *is a finite set of* nonterminal symbols*,*
- T *is a finite set of* terminal symbols*,*
- S *is a set of* starting symbols*, where* $S \subseteq N$
- T *is a finite set of* production rules *of the form* $[a,\ C,\ O,\ r \to n]$*, where*
  - $a \in T$ *is a terminal symbol,*
  - $C \subseteq A$ *is a set of* required *attributes,*
  - $O \subseteq A$ *is a set of* optional *attributes,*
  - r *is a 1-unambiguous regular expression over an alphabet* N*,*
  - $n \in N$ *is nonterminal symbol.*

If $[a_1,\ C_1,\ O_1,\ r_1 \to n]$ *and* $[a_2,\ C_2,\ O_2,\ r_2 \to n]$ *are two production rules, then necessarily* $a_1 \neq a_2$*.*

Without loss of generality, we assume that no two production rules from a given grammar have concurrently the same terminal symbol a and the same nonterminal symbol n on the right-hand side of the rule. This restriction is expressed by the last condition in the definition. If there should be such two consistent rules, they can be merged together into a single rule.

Informally let us have a data tree and a regular tree grammar to which the given tree should conform. In order to verify this conformity, we can process the tree in a top-down manner, i.e. a selected node is processed after its parent is already processed. The regular expression restricts the suitable sequences of children whereas sets of attributes enforce the presence of attributes. If all conditions are satisfied and a label of a given node corresponds to a terminal

symbol, the given node can be assigned with a nonterminal symbol from used production rule.

Probably a more frequent formalism for XML schemata are variations of tree automata. This approach is used for example by [12, 15, 16], the basic overview of tree automata can be found in [40]. Approaches using tree automata are in fact very close to our grammars. Since we cannot directly use the mechanism of an automaton and we need to modify validation algorithm itself, we have chosen the grammar approach, although we will use production rules rather for validation than generation purposes.

2.3.1 Data Trees Validity

Now we will define the validity of a data tree with respect to a given regular tree grammar. The first step of this effort is an interpretation tree notion.

Definition 2.15 (Interpretation Tree)**.** *Let $\mathcal{T} = (D, lab, val, att)$ be a data tree and $\mathcal{G} = (N, T, A, S, P)$ is a regular tree grammar. An* interpretation tree *of a tree $\mathcal{T}$ against grammar $\mathcal{G}$ is a tuple $\mathcal{N} = (D, int)$, where*

- *D is the original underlying tree,*
- int *is a function $D \to N$ mapping nodes to nonterminals and satisfying the following conditions:*
  - *For each node $e \in D$ and its child nodes $e_1, \dots, e_i$ for some $i \in \mathbb{N}_0$, there exists a rule $[a,\ C,\ O,\ r \to n]$ in the set of production rules P such that:*
    - · $int(e) = n$,
    - · $lab(e) = a$,
    - · $\forall t \in C,\ \exists (t, v) \in att(e)$ *for some* $v \in \mathbb{V}$,
    - · $\forall (t, v) \in att(e),\ t \in C$ *or* $t \in O$, *where* $v \in \mathbb{V}$,
    - · $int(e_0).int(e_1) \dots int(e_i) \in L(r)$.
  - *If e is the root node, then $int(e) \in S$.*

Given data tree is valid with respect to a provided grammar, if there exists at least its one interpretation.

Definition 2.16 (Data Tree Validity)**.** *We say that a data tree $\mathcal{T} = (D,\ lab,\ val,\ att)$ is* valid *against a regular tree grammar $\mathcal{G} = (N,\ T,\ A,\ S,\ P)$, if there exists at least one interpretation $\mathcal{N}$ of $\mathcal{T}$ against $\mathcal{G}$.*

A set of trees is a regular tree language, if all trees in this set are valid against some regular tree grammar and no other trees are valid.

The goal of our correction algorithm is to correct a provided data tree in order to acquire another not too distant data tree that is valid against a given schema. This algorithm works step by step through the original data tree, thus we can use the following definition of local validity to gracefully describe the local working of our algorithm.

Definition 2.17 (Local Validity). *Assume that $\mathcal{T} = (D, lab, val, att)$ is a data tree, $p \in D$ is a node and $\mathcal{G} = (N, T, A, S, P)$ is a regular tree grammar. We say, that p is* locally valid, *if $\mathcal{T}_p^{tree}$ is valid with only one exception, that if $p \neq \epsilon$, we do not insist on the last partial condition of validity, thus we do not require that $int(\epsilon) \in S$.*

2.3.2 Grammar Classes

The main problem of validation against regular tree grammars is caused by competing nonterminal symbols.

Definition 2.18 (Competing Nonterminals). *Let $\mathcal{G} = (N, T, A, S, P)$ be a regular tree grammar and n_1, $n_2 \in N$, $n_1 \neq n_2$ are two different nonterminal symbols. We say that n_1 and n_2 are* competing *with each other, if there exist two production rules $[a, C_1, O_1, r_1 \to n_1]$ and $[a, C_2, O_2, r_2 \to n_2]$ sharing the same terminal symbol a.*

Assume that we have a data tree and its node e with $lab(e) = a$. This label corresponds to an equivalent terminal symbol (a), but if there are competing nonterminal symbols having this terminal on their left-hand sides, we do not know, which of these rules to chose and, therefore, which unique nonterminal symbol should be assigned to the given node.

We will next introduce two classes of regular tree grammars with limited nonterminals competition. The local tree grammar is the most restricted one, the single type tree grammar is less restricted. In [38] or other papers we can also find the definition of restrained competition tree grammar as another class, but for our purposes this class is not relevant.

Definition 2.19 (Local Tree Grammar). *A regular tree grammar is a* local tree grammar, *if it has no competing nonterminal symbols. A set of trees is a* local tree language, *if all trees in this set are valid for some local tree grammar and no other trees are valid.*

Definition 2.20 (Single Type Tree Grammar). *A regular tree grammar $\mathcal{G} = (N,\ T,\ A,\ S,\ P)$ is a* single type tree grammar, *if both following conditions hold:*

- *For each production rule $[a,\ C,\ O,\ r \to n]$ all nonterminal symbols in r do not compete with each other,*
- *Starting nonterminal symbols in S do not compete with each other.*

A set of trees is a single type tree language, *if all trees in this set are valid for some single type tree grammar and no other trees are valid.*

Any local tree grammar is necessarily a single type tree grammar. This implies directly from the given classes definitions. Providing particular examples it could be proven, that some regular tree grammars are not single type and

analogously that some single type tree grammars are not local ones. Similar conclusions can be derived for corresponding tree languages classes.

In other words local tree grammars have strictly less expressive power than single type tree grammars and these grammars have strictly less expressive power than regular tree grammars.

Following the propositions presented in [38], any data tree has at least one interpretation against a local tree grammar or against a single type tree grammar. As a consequence, if a given data tree is valid with respect to some grammar of those two types, there exists right one interpretation of a given data tree.

2.4 Document Type Definition

A Document Type Definition [18], abbreviated as a DTD, is a simple schema language for restricting content of XML documents. Its specification was introduced as a direct part of the Extensible Markup Language recommendation and this tight binding also results in a fact, that a DTD schema can be an integral part of a given XML document.

A DTD schema language allows describing the structure of an XML document, focusing in particular on nesting of elements and presence of attributes. Although this language can declare as well other types of constraints and declarations, e.g. notations, we are interested only in elements and attributes, since our model of XML documents is simplified too.

As shown in [38] the expressive power of a DTD is at the level of a local tree grammar. The detailed description of schema translation into a grammar will be shown in the next subsection. The basic idea is however hidden in a fact that element names directly correspond to their nonterminal symbols. Thus there can be no competing nonterminal symbols and thus the derived grammar is a local tree grammar.

2.4.1 Schema Translation

Inspired by [11] we will now show how constructs from a DTD schema can be translated into a grammar rules from Definition 2.14.

Formally we want to derive $\mathcal{G} = (N, T, A, S, P)$ grammar, therefore we need to define individual sets of symbols and the set of production rules.

Root Element

In order to simplify our notation, we will not bother with precise white spaces notion and we will only use simple blank character. Each DTD schema therefore has the form:

<`!DOCTYPE` *RootElement* `[` *ListOfSpecifications* `]` >

Item *RootElement* represents the name of the outermost element, i.e. the label of the root element of an XML document conforming to a given DTD. Because no other outermost element names are not allowed, we can define $S = \{RootElement\}$.

Item *ListOfSpecifications* contains individual specifications for element and attributes definitions. Both of them are discussed in the following paragraphs.

Element Declarations

An element declaration has the following form:

<!ELEMENT *ElementName*
(EMPTY | ANY | *ElementContent* | *MixedContent*) >

Item *ElementName* defines the name of an element the given declaration describes. There are generally four different types of content, which is permitted in a given element. In our work we will only consider *ElementContent*, `EMPTY` and `#PCDATA`, which is the simpliest case of *MixedContent*.

For each element declaration in a given DTD we insert *ElementName* into a set N of nonterminals and also into a set T of terminals. This correspondence between terminal and nonterminal symbols causes the resulting grammar to be a local tree grammar as discussed earlier.

Model *ElementContent* allows specifying a regular expression describing permitted sequences of child elements, the processed element can have nested inside. Since these expressions may contain in addition to standard operators also ? and $^+$ operators, we translate them first. All regular expressions have to be 1-unambiguous.

For an element declaration with element content, we add a production rule of the form

$$[ElementName, \emptyset, \emptyset, r \rightarrow ElementName]$$

into a set P. In this case r is a adjusted regular expression corresponding to *ElementContent*.

For an element declaration with only a textual content, i.e. `#PCDATA` as the case of *MixedContent*, we add a production rule of the form

$$[ElementName, \emptyset, \emptyset, \texttt{data} \rightarrow ElementName]$$

into a set P. Symbol `data` is a special distinct nonterminal symbol for data nodes, as introduced in Definition 2.6. Hence, we need to insert it both in N and T sets of symbols and also we need to add rule of the form

$$[data, \emptyset, \emptyset, \varnothing \rightarrow data]$$

into a set P. Using this rule we are able to recognise data nodes.

Attribute Declarations

An attribute declaration has the following form:

$<$!ATTLIST *ElementName SetOfAttributes* $>$

It declares a set of attributes, that can appear inside an *ElementName* element. If a given DTD has no specification for *ElementName* element itself, we can ignore such attribute declaration.

Each individual attribute declaration in *SetOfAttributes* has the form:

AttributeName AttributeType AttributeDefaults

Item *AttributeName* defines an attribute name and *AttributeType* chooses one of the allowed attribute types from the predefined set or represents an enumeration of values defined by the user. Since we are not interested in attributes typing, we only need to discuss *AttributeDefaults*, which tailors the presence of attributes themselves inside elements.

Let $[ElementName, C, O, r \to ElementName]$ be a production rule from P corresponding to *ElementName*. In case of `#REQUIRED` we add *AttributeName* into a set C and in case of `#IMPLIED`, `#FIXED` or in case with specified attribute default value we add *AttributeName* into a set O.

In all cases we also add *AttributeName* into a set A of all attribute names defined in a given DTD.

Other Declarations

Declarations of entities or notations are ignored, because we are only interested in structural schema constraints on elements nesting and presence of attributes.

2.5 XML Schema

An XML Schema language proposed in [26, 53, 6] is another example of representative language for writing schemata for XML documents. Its expressive power is mainly within the single type tree grammar class, but some constructs violate restrictions of this class. As in [38] we will ignore these constructs and, therefore, we are not forced to use the most general form of validation algorithm for regular tree grammars.

The considered constructs are especially `element`, `attribute`, `attributeGroup`, `complexType`, `simpleType`, `group`, `all`, `choice` and `sequence`. On the other hand we will ignore constructs like `annotation`, `any`, `anyAttribute` or integrity constraints like keys.

The essential difference between DTD and XML Schema is that in XML Schema nonterminal and terminal symbols do not correspond to each other. Therefore we gained more possibilities how to express structural constraints on elements and attributes.

We will however not discuss all constructs in detail, since it would be out of the scope of this work. The purpose of this section is thus not to provide

a complete and formal framework for XML Schema translation into a tree grammar.

2.5.1 Schema Translation

Our goal is to transform a given XML Schema document into a $\mathcal{G} = (N, T, A, S, P)$ grammar. This involves definition of listed sets of symbols and a the set of production rules.

Although we will not specify any prefix for names in the following examples, we expect that names of schema components are from the standard XML Schema namespace.

Element Declarations

A declaration of `element` can appear directly in `schema` or nested somewhere in `complexType`, i.e. under a particular `all`, `choice` or `sequence` construct. The former case represents globally defined elements, the latter locally defined elements.

As discussed previously, content model of elements depends on a context, i.e. terminal and nonterminal symbols are not equivalent and cannot be directly derived from only element names themselves.

A type of a given `element` is declared either by a specification of named type in a `type` property, or by an anonymous type declaration using nested `simpleType` or `complexType`.

```
<element
    name="ElementName"
    type="ElementType"
    ref="ElementReference"
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
    (simpleType|complexType)
</element>
```

Declarations of `complexType` are either named explicitly by the user or we assume that the anonymous ones can be associated with a unique name using some provider. These names can be based for example on sequential numbering or an adoption and adjustment of names of superior `element` declarations.

For each `element` declaration we define corresponding nonterminal symbol as *ElementNonterminal* = *ElementName*.`@`.*ContextName*, where *ContextName* is equal to `global` for globally defined elements and *TypeName* of corresponding `complexType` for locally defined ones, whether this name was defined explicitly or automatically assigned. We only suppose that `@` is a character not allowed in names of XML Schema components and that `global` cannot conflict with remaining names *TypeName*.

We are now ready to formulate required production rule for a given element declaration. We first add *ElementName* into a set T of terminal symbols and *ElementNonterminal* into a set N of nonterminal symbols. Finally, we add a rule of the form

$$[ElementName,\ C,\ O,\ r \rightarrow ElementNonterminal]$$

into a set P of production rules. Sets of attributes C and O are initialized to empty sets and content model r is derived from an element type. Mechanisms of all these derivations will be the subject of the following paragraphs.

If `ref` is present, we adopt target `element` declaration of name *ElementReference*. In this case we use target *ElementName* as a terminal symbol, but for the derivation of a nonterminal symbol we use the local context, not target context.

Root Elements

Any element that is globally declared in a schema may be used as a root element of an XML document valid with respect to a given schema.

```
<schema>
    <element name="ElementName" .../>
    ...
</schema>
```

Since *ElementName*.`@global` is a nonterminal symbol associated with such element with name *ElementName*, we add this nonterminal symbol into a set S of starting symbols of the grammar.

Attribute Declarations

An `attribute` declaration can appear either in `attributeGroup` or `complexType`. Each of these constructs in fact represents a set of such attribute declarations and via an anonymous or named type this set is finally associated to an `element` declaration.

Hence, let $[ElementName,\ C,\ O,\ r \rightarrow ElementNonterminal]$ be a corresponding production rule from the set P of the grammar. *ElementName* represents a name (terminal symbol) for a processed element and *Element-Nonterminal* a corresponding nonterminal symbol. Now we are provided a set of `attribute` declarations. Each of them is of the form:

```
<attribute
    name="AttributeName"
    ref="AttributeReference"
    use="AttributeUsage"
>
    (simpleType)?
</attribute>
```

We first add *AttributeName* into a set A of attribute names. If *AttributeUsage* is equal to `required`, then this attribute is required and we therefore add *AttributeName* into a set C. If it is `optional`, we extend a set O. Otherwise we do nothing, since we do not handle `prohibited` alternative.

If the declaration contains `ref` property, we simply adopt the referenced attribute declaration, i.e. globally declared attribute with a name *AttributeReference*. Properties `type`, `default`, `fixed` and others are not interesting for our work.

Attribute Groups

As described ealier, `attributeGroup` directly or indirectly represents a set of `attribute` declarations. These declarations are finally associated with a particular `element` declaration.

```
<attributeGroup
    name="GroupName"
    ref="GroupReference"
>
    (attribute|attributeGroup)*
</attributeGroup>
```

If a property `ref` is present, we simply adopt referenced attribute group, i.e. globally declared group with a name corresponding to *GroupReference*.

Simple and Complex Types

A declaration of `complexType` can appear as a global definition in `schema` or locally within `element` declaration.

```
<complexType
    name="TypeName"
    mixed="MixedContent"
>
    (simpleContent) | (complexContent) | (
        (group|all|choice|sequence)?,
        (attribute|attributeGroup)*
    )
</complexType>
```

The only thing we need to do is to gather a content model specification and a set of attribute declarations from subordinated constructs. When a particular type is assigned to an `element` declaration, all required steps are performed to translate declarations into a grammar rule.

If we are processing `complexType`, we propagate a regular expression over nonterminal symbols. This expression is composed from subordinated `group`, `all`, `choice` and `sequence` constructs.

In case of `simpleType` declaration, we propagate `data` as the only nonterminal symbol of the content model. In order to recognize data elements, we finally need to add a special rule

$$[data, \emptyset, \emptyset, \varnothing \rightarrow data]$$

into a set P of production rules. The symbol `data` is a special distinct nonterminal symbol for data nodes, as introduced in Definition 2.6. Hence, we need to insert it both in N and T sets of symbols.

Restriction and Extension

Constructs `extension` and `restriction` are used within `simpleContent` and `complexContent` declarations. The particular usage conditions depend on the context. Since we are not interested in simple types and we treat data values as not distinguished `data` nodes, we only need to gather and later on propagate untouched declarations of attributes or in latter case a content model as well.

```
<simpleContent|complexContent>
    (restriction|extension)
</simpleContent|complexContent>
```

Model Groups

At first we need to create a mechanism for the translation of values `minOccurs` and `maxOccurs` into appropriate operators and subexpressions of regular expressions describing content models.

Let r be a regular expression corresponding to the nested constructs such as `element`, `group`, `all`, `choice` or `sequence`.

If `maxOccurs` is equal to `unbounded` and if `minOccurs` is equal to $i \in \mathbb{N}_0$, then the resulting regular expression is equal to $(r)_i.r^*$, where $(r)_i$ is the concatenation of i times repeated r.

If `maxOccurs` is equal to $j \in \mathbb{N}_0$ and `minOccurs` to $i \in \mathbb{N}_0$, then the resulting expression is equal to $(r)_i.(r|\epsilon)_{(j-i)}$. We assume that i and j represents a valid range.

```
<group
    name="GroupName"
    ref="GroupReference"
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
    (all|choice|sequence)?
</group>
```

If a `ref` is used, we adopt target declarations of a group with name *GroupReference*. The resulting regular expression for a `group` content model is adjusted from nested `all`, `choice` or `sequence` by `minOccurs` and `maxOccurs` properties.

```
<all
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
    (element)*
</all>
```

In case of `all` the XML Schema requires that all nested `element` declarations have restricted domains for `minOccurs` and `maxOccurs` properties, such that each individual element can appear either 0 or 1 times.

Let $P = \{p_1, \ldots, p_n\}$ be a set of content model expressions from all nested `element` constructs for some $n \in \mathbb{N}$. The initial idea of creating the resulting expression can be based on the alternation and the Cartesian product of items in P. However, we have to ensure, that the resulting expression is 1-unambiguous.

```
<choice
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
    (element|group|choice|sequence)*
</choice>
```

Let $p_1, \ldots, p_n$ be expressions of content models from all nested `element`, `group`, `choice` or `sequence` constructs for some $n \in \mathbb{N}$. We first prepare a regular expression for the alternation defined by $(p_1|\ldots|p_n)$ and after it we create the resulting expression by adjusting the alternation expression by `minOccurs` and `maxOccurs` properties.

```
<sequence
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
    (element|group|choice|sequence)*
</sequence>
```

Let $p_1, \ldots, p_n$ be expressions of content models from all nested `element`, `group`, `choice` or `sequence` constructs for some $n \in \mathbb{N}$. We first prepare a regular expression for the concatenation defined by $(p_1.\ldots.p_n)$ and after it we create the resulting expression by adjusting the concatenation expression by `minOccurs` and `maxOccurs` properties.

```
<element
    name="ElementName"
    type="ElementType"
    ref="ElementReference"
    minOccurs="MinOccurs"
    maxOccurs="MaxOccurs"
>
```

```
    (simpleType|complexType)
</element>
```

Assume that an `element` declaration is nested directly or indirectly in some `complexType` and let *ElementNonterminal* be a nonterminal symbol corresponding to a given `element`. The resulting regular expression of this element is equal to the nonterminal symbol adjusted by `minOccurs` and `maxOccurs` properties.

Chapter 3

Analysis

Since the processing of incorrect XML data not only involves proposing structural corrections focused in this work, this section will provide overview of existing theoretical approaches and discussion of general aspects of correction frameworks for XML documents.

Generally we can classify errors in XML documents primarily into three levels, starting with well-formedness violation, continuing with structural invalidity and ending with data inconsistency. However, we do not need only to find corrections for incorrect data, we can also leave the invalid data untouched and extend possibilities of querying over them or we can treat such data as incomplete or unreliable.

There are also several aspects that need to be clearly answered in order to create meaningful correction framework that could be able to find fulfilment in practise. In the last part of this chapter we will discuss five basic theoretical approaches for correcting structural invalidity or integrity constraints violation in XML documents.

3.1 General Aspects

Except the correction possibilities themselves, we need to at least take into account several aspects that may radically influence the possibilities and behaviour of the proposed correction framework. We will successively discuss the most important of them and we will also outline several interesting alternative approaches not attempting to provide repairs of incorrect documents.

Errors Classification

As we have already outlined, we can determine essentially three basic levels of incorrectness that can be found in XML documents. The first fundamental question is the inspection of well-formedness of documents. If a given document is not well formed, we cannot view it as a tree. Once we have a well formed tree we can focus on its validity. This means we can detect violations of its structure,

which is defined by a tree grammar. Finally, having a valid document, we are able to solve questions about correctness of data values contained in it.

This classification can naturally be simplified or extended, but the main idea will probably be preserved. This principle stands on the fact that it is quite reasonable to first ensure the given document is well formed, then valid and, finally, consistent. The inverse order would probably not lead to good results or transparent frameworks.

However, another question is, whether we are able to use some sort of helping knowledge from latter levels within the prior ones. For example, if we are provided a not well formed document, we can harness the knowledge of the structure defined by the tree grammar and directly use it in order to find more suitable repairs of formedness. Similarly we can utilise the type of data values to find more suitable structural corrections.

At the level of structural repairs we can generally focus on allowed content models, order of sibling nodes, labels of elements, correct nesting or presence of attributes. Particularly we can refer to [19, 16, 54, 44, 9]. At the level of data consistency we can consider corrections with respect to integrity constraints, functional dependencies, keys or foreign keys, type checking or consistency of `ID` and `IDREF/S` values of attributes. Papers discussing this level are for example [27, 28, 51, 52].

Correction Objectives

When proposing a new approach for corrections, we first need to answer several principal questions. It is apparent that we can generally achieve probably much better corrections, if we are not limited by space or time complexity. The more complex and interesting correction strategies, the higher probability of finding suitable repairs is. Otherwise we can limit ourselves and then we are forced to settle for worse results.

We can also study the characteristics of documents to be corrected. It seems that at least two basic groups can be distinguished. Some documents are textual designated especially to human readers, other documents have strictly data nature, are precisely structured and created especially for programs.

The former group puts an attention probably on data values, the latter one on correct structure or consistency. Even the given documents may not have specified explicit schema, applications processing them nearly always assume the schema at least implicitly. For these reasons it is a good idea to introduce a framework capable at least to achieve well-formedness and validity, leaving other questions to particular applications. Only they can perfectly master the semantics that cannot be caught in the schema even if it is provided.

If we know what we need to correct, we also need to know the allowed range of errors. For example, an approach discussed in [44, 9] assumes only documents that are not too far from their explicitly provided schema.

Processing Strategies

Without any doubt the better correction results can be achieved in situations, when we can work with the entire data tree loaded into the system memory. This approach is represented by parsers [25]. We can walk through the tree without limitations, we can inspect the tree from the global context, perform extensive or distant repairs, but we are probably limited in the size of a document that can be processed this way. On the other hand, we can propose a framework which can dynamically fetch only required portions of data from hard disk and thus facilitate correction of huge documents.

For streaming XML documents we can use for example parser [36]. However, it is a question whether this approach would at all allow at least some correction strategies.

Finally, we can discuss one special problem connected with the incremental validation of XML documents. Anyway we do not always need to process and correct entire trees, but we can only inspect smaller parts of documents, assuming that the rest of them is valid. This efficient way of revalidation connected with correction proposals is introduced in [16, 14].

In this case we can harness one interesting idea that is based on refusing those suitable local corrections, which are too close to the original document. If the user once decided to modify it, proposing backward repairs would not be a good idea.

Edit Operations

Probably one of the crucial aspects of any correction framework is the set of permitted operations used for transforming trees in order to correct them. Generally we can distinguish three basic strategies. First we can insert new information into a tree, next we can modify the existing information and, finally, we can delete it from the tree.

When deleting, we obviously lose some portion of the information that was originally stored in a tree. It is a question, whether we should perform deletions in situations, in which we are in fact not sure about the absolutely fitting repair and we only do it because it seems that no better solution can be found. However, this can only mean, that we are not able to find a better solution, not that it cannot generally exist.

Therefore, some papers work with the idea of keeping incorrect information in a tree and marking it as not reliable. This approach is presented in [27].

Anyway the correction framework needs the definition of comparison of found repairs in order to decide, what repairs are better than others. Having a set of proposed elementary operations, for example an addition or deletion of a leaf node and a node label rename, we can base this comparison on a cost function that assigns to each operation some non-negative cost. The lower cost, the better repair. This idea is generally used, but we can name for example [16]. Another alternative based on a greatest lower bound of all found repairs can be found in [51].

The introduced model of measuring distances between trees can easily be extended to the notion of distance between a tree and a grammar. The problem of document correction with respect to a given schema, therefore, can be viewed as finding those repairs that have minimal distance to such grammar.

Alternative Approaches

Having incorrect XML data, we not only need to correct them. In the area of querying we are not forced to correct the data, we can as well adjust prepared query statements to deal with incorrectness or we can harness errors to provide more extensive answers.

The latter named approach is presented for example in [48, 27, 28]. We can process the potentially invalid XML documents and propose their corrections. The goal is, however, not to find the best repair and its consequent application, but we consider several suitable repairs together concurrently. Then we are able to introduce notions of possible and certain answers with the meaning that a possible answer is involved at least in one considered repair and a certain answer in all repairs.

Following this extended way of querying and query evaluation we are able to access potentially damaged documents and retrieve more information.

Furthermore, we can find another alternative approaches that directly do not propose corrections. For example the framework presented in [1] maintaining the incomplete information. The purpose of this work is to introduce storage for XML documents that need not be complete. Thus the authors established a simple mechanism for the description of missing or incomplete values. Similar strategies can be used in the context of corrections too.

Interactivity and Learning

There are essentially two options how the best suitable repair can be chosen from the set of all found suitable ones. We have already outlined the possibilities of cost functions, but these do not solve the problem definitely. If we need to find best corrections, we had better interact with the user. The algorithm then attempts to find repairs, it orders them using the defined mechanism of comparison and then yields the final decision up to the user.

In this approach the algorithm should probably be able not only provide the minimal repairs, but also some other ones, since we cannot assume that the cost function can be defined perfectly to fit expectations of the user. When there is no interaction, the algorithm can select the right one best repair on its own and in this case the algorithm obviously can automatically throw away all candidates with not minimal cost.

Finally, we can discuss the question of learning possibilities of the correction framework. From the first point of view it would be interesting, if the correction algorithm would be able to gather experiences from already proposed repairs confirmed by the user, but the problem is quite more complicated.

The learning itself would only be useful, if the algorithm would be able to derive new knowledge from already learnt situations and apply it under circumstances that are not equal to that learnt. Otherwise the learning would bring the advantage only if we could apply learnt situations repeatedly, thus the input document must repeatedly provide these situations. However, in these cases it seems better to even manually find the suitable correction and then massively apply it using standard XML transformations tools like XSLT [21]. The expressive power of such tools obviously exceeds the scope of a correction framework itself.

3.2 Existing Implementations

Except the prototype implementations from some theoretical research papers dealing with corrections of XML, there not exists any complex tool for correcting XML documents in a way related to structural corrections focused in this thesis.

However, we can find several tools correcting HTML [29]. Although they process primarily HTML documents, there is a clear connection to XML. Since these tools do not have a solid theoretical background and they have many constructs firmly associated only with a schema for HTML, we will provide their short listing with brief descriptions.

HtmlCleaner

HtmlCleaner [31] is an HTML parser available as a Java [32] library or a command line tool. Its original motivation was to correct existing HTML documents in order to acquire valid XML documents prepared for querying using XPath [22], XQuery [7] or XSLT [21].

However, it primarily serves for transforming provided HTML documents into well formed ones. The parser is for example able to reorder individual elements, correct invalid attributes or nesting of elements and is also partially configurable allowing a custom tags filtering and balancing functionality.

NekoHTML

NekoHTML [39] is another example of a tool for correcting HTML documents, which in fact produces valid XML trees. It represents a simple scanner and tag balancer, generating documents capable to be processed via standard XML interfaces.

It is particularly able for example to add missing parent elements, automatically close elements or can handle mismatched inline element tags.

HTML Tidy

HTML Tidy [30] is the last example of an existing implementation for corrections of HTML documents and, moreover, it is able to directly process XML

documents as well. Users are again provided with a library and a command line utility.

In case of HTML documents this tool is able to correct their well-formedness and propose other repairs based on the most common coding errors. Usually it is used for transforming HTML documents into valid XML. When processing native XML documents, HTML Tidy is able to correct them in a way of producing well formed documents, but the question of validity is not considered.

3.3 Theoretical Research

In this section we will discuss existing papers related to the correction of XML documents. First three approaches pursue structural correction to achieve valid documents with respect to a given local tree grammar. The remaining two papers focus the problem of repairing documents against a set of integrity constraints.

The major attention is put only to first two approaches, since the correction framework presented in this thesis adopts several aspects right from these two studies.

3.3.1 Incremental Validation and Correction

Suppose that we have an XML document valid with respect to a given schema and the user modifies it. In order to verify validity of the resulting document, we are not necessarily forced to validate the entire document, but we can focus only on its modified parts. This problem is discussed for example in papers [5, 12, 15].

Authors of these papers later on attempted to enrich the concept of incremental validation by correction proposals in [19, 16] and, finally, they summarised the proposed incremental correction framework in [14]. This paper represents one of two crucial starting points of this thesis.

Incremental Correction

Let us first return to the problem of the incremental validation. At the beginning we have a valid document, which is subsequently updated by the user. This means that the user performs a sequence of trivial modifications like a leaf node insertion or an element label renaming. This sequence can be viewed using the paradigm of transactions. We are naturally not interested in their proceeding, but at the end we need to obtain a document, which is again valid with respect to a provided schema.

Since the user probably did not perform radical modifications, it seems to be more efficient only to focus on those local parts of the tree, which have been affected by realised updates. Anyway, if the resulting document is not valid, we can prompt the user for corrections. Or we can as well attempt to guess

some suitable corrections and offer them. The user thus can easier find out what is wrong and he/she is even provided by some suitable solutions.

The proposed correction algorithm processes the updated tree from leaves towards the root node, i.e. we are inspecting a given node only after all its child nodes are already inspected. Whenever we find out that the given node has not its child nodes matching the allowed content model, we invoke the local correction routine, whose purpose is to locally repair these invalid child nodes.

The corrections are based on a consecutive generation and inspection of words in the language defined by the regular expression restricting the content model of a given locally invalid node. We dynamically generate all possible sequences of child nodes, bypassing not perspective ways and propose suitable corrections. The authors, however, only considered a DTD schema belonging to the class of local tree grammars with no competing nonterminal symbols and identical sets of terminals and nonterminals.

Formal Model of XML Trees

XML documents are modelled using unranked ordered labelled trees. The underlying tree is composed from a set of nodes numbered using the prefix numbering schema over non-negative integers, thus the tree itself is defined only implicitly via this numbering schema as in the approach proposed in this thesis. However, attributes are completely ignored. Not only that they are not considered in the tree model, but they are not involved in corrections too.

The validity of data trees is based on the definition of unranked tree automaton, similarly to our grammars. The validation algorithm starts the processing of the tree at leaf nodes and if there exists the transition with corresponding element label and the child nodes sequence matches the regular expression, the automaton assigns the inspected node its state and then continue towards the root node. If the root matches any of the final states, then the document is valid. Whenever we cannot use any transition rule, we attempt to correct the given node and then continue with the rest of the validation traversal.

Edit and Update Operations

Both the correction routine and the user express the required document changes using the provided edit operations, which are composed together to form more complex update operations.

The purpose of discussed papers was not to propose the set of most suitable operations, thus only three elementary edit operations were introduced. They are the removal of an existing leaf node, insertion of a new leaf node and a label renaming of an existing node in a tree.

Each of these edit operations is assigned a unit cost and the framework itself does not discuss other variants, even though it is clear that less restrained costs would be also acceptable.

Update operations represent possibilities the correction routine can choose from in order to correct provided sequence of nodes. The first update operation is the insertion of a subtree. Because the schema generally allows the recursion and even via Kleene star operation in regular expressions we can be threaten by potentially infinite trees, the insertion operation only allows the insertion of minimal trees.

The deletion update operation is intended for the complete removal of existing subtrees. Finally, the replace operation may change the label of a given node and then recursively attempts to process the sequence of all its child nodes and subtrees in order to correct them using any of introduced three update operations. We generally attempt to find those repairs that are as close to the original tree as it is possible.

This distance is measured using the introduced cost function, which is extended to update operations too. Since we can view update operations like sequences of edit operations, cost of an update operation is simply the sum of costs of all its component edits.

Correction Routine

Suppose that we are traversing the updated tree and we failed to validate some node. There are essentially two reasons why the local validation could fail. First the label of a given node may not be defined by the provided schema and thus we have no transition rule to even take into account. Although the described formal framework explicitly does not handle these situations, we can simply move forward one level up and then correct this unknown node as an ordinary node in a certainly not valid sequence of nodes.

The second and probably more common reason is that we have a regular expression describing the content model, but the sequence of child nodes does not conform to it. The proposed correction routine attempts to rectify this situation by finding those sequences that are suitable and also with minimal distance.

Note, however, that the problem is composed of two orthogonal dimensions. First we horizontally step by step generate words from the language of a given regular expression. These words are over the alphabet of element names, since we do not need to distinguish between terminal and nonterminal symbols. However each of these symbols in a generated word represents the entire subtree. As a consequence we also need to proceed vertically.

At the horizontal level our goal is to generate words of element names, i.e. new possible sequences of child nodes. For this purpose we use the Glushkov automaton, whose state space we traverse to the depth following defined transitions. Although the algorithm processes dynamically, suppose now that we have generated some suitable word. Using the standard Levenshtein distance model we resolve the distance between the original invalid word and the generated one.

The Levenshtein metric considers a symbol insertion, deletion or replacement. In our case we have update operations, which follow the same idea, only

these operations involve also nested subtrees and costs are not trivial. When we decide to insert a new node in the horizontal plan, we not only need to add a new leaf node using the corresponding edit operation, but we also need to insert all its potential child nodes in order to ensure the given newly inserted subtree will be locally valid.

Distance Matrix and Threshold

It is easy to see that the algorithm would be significantly inefficient if it would really first generate a word and only then inspect its distance to the original invalid one. Therefore, authors use the general concept of dynamic programming and via the edit distance matrix we successively generate words and concurrently we straight away compute the distance to the original one.

Having any cell of this distance matrix, the row represents the number of processed symbols from the original invalid word and the column number stands for the size of the generated word. The first column of the matrix contains values ∞ standing for very great costs and the second column contains costs of the original subtrees deletions. Similarly the first row again contains infinite values, the second one costs representing subtrees insertions.

At the beginning we are provided the original word and the generated word is so far empty. Therefore, we initialize only the first two technical columns of the matrix. During the algorithm execution, we successively try suitable allowed symbols, thus we extend the matrix with new columns and incrementally compute the distance.

Each cell is computed from three nearest neighbouring cells located to the left and top including the diagonal one. These three options represent possible actions that can be taken into account at a given position. We can ignore the next symbol and first rather insert a new subtree, or we can delete the existing one or, finally, we can repair the existing one in the connection to the newly generated symbol. All cheapest possibilities win.

If we reach the accepting state of the Glushkov automaton, we have generated allowed sequence of nodes and the value located in the last column and the last row represents the distance of this sequence to the original invalid one. The cells not only contain cost values, but also particular sequences of accounted operations, which are of the minimal cost and through which we can completely repair the original tree.

However, the authors had to introduce the concept of pruning, because due to recursive tree grammars or Kleene star operation, the algorithm might never stop and incessantly generate longer and longer words. Therefore, at the beginning we specify the fixed constant with the meaning of threshold and whenever during the traversal we exceed this value, we treat the inspected direction as not perspective, we backtrack and attempt to choose some other direction.

Repairs Gathering and Merging

Whenever the algorithm reaches the accepting state of the automaton and the distance does not exceed the allowed threshold, we have found a suitable correction. The horizontal generation of words continue until the entire state space of the automaton is examined. All successively found corrections are gathered.

The local correction routine finally terminates and we continue processing the original tree towards the root node. Whenever we discover another invalid node which is the ancestor of some previously corrected node, we can encompass already found corrections and do not need to compute them again.

Finally, when the tree is completely processed, we locate isolated areas of the tree and combine proposed solutions for their local correction into the aggregated global correction proposals.

Approach Assessment

Even though the set of introduced update operations and the model of corrections itself is sufficient to enable basic XML trees correction, we can identify several problematic aspects and disadvantages.

First there is a problem in the way, how the algorithm processes the entire tree. Whenever we have processed all sibling nodes and enter their parent node, we only correct this parent, if it is not locally valid. If so, we can only work with its child nodes, but we cannot change the label of the parent node itself. Assume a situation when we have explored such node and we have attempted to repair it, thus find another sequence of child nodes that is suitable and with the minimal distance. Suppose that the grandparent node is locally valid, thus there is no way how to change the label of the original parent. But potentially there may exist other labels, which would perfectly fit either the context of this renamed parent and also the grandparent. Since the proposed model does not permit this technique, we can throw away potentially very cheap repairs.

However, there are more crucial disadvantages. The algorithm ensures that we always find all minimal corrections and also some other non minimal ones. Unfortunately always does not mean really always. We are only assured to find corrections in situations, when there exists some correction with cost at most equal to the threshold value. Since the threshold is a constant, there can always exist some invalid data tree, which could not be repaired, since all promising directions during the horizontal and vertical generation are pruned too early due to the exceeded cost. As a consequence the algorithm may not be able to find local corrections and, in addition to this disadvantage, the problem itself is not formally discussed in the paper.

It seems that we can set the threshold value high enough. And in fact we really can and probably there might also exist a formal calculation of needed value for each input tree. However, high threshold values bring other difficulties and thus do not solve the problem itself.

The reason is quite simple, because the algorithm prunes unpromising di-

rections only using the idea of exceeding the given threshold. The higher value for threshold, the more time the algorithm will spend inspecting ways that obviously do not lead to the minimal corrections. Just consider the situation when we are processing a sequence of nodes with respect to the regular expression for example $(a|b)^*$. We will be forced to generate the enormous number of evidently non perspective words over symbols a and b using the insertion update operation, without any opportunity to stop this hopeless working earlier.

And, moreover, we can find another disadvantage connected with this problem, namely repeated computations. Each symbol a or b in the previous example may represent a nontrivial subtree, which need to be repeatedly constructed. The same problem can be detected in any update operation wherever in the original tree. Generally we not only need to generate suitable sequences of nodes dynamically, but we are forced to repeatedly execute the same computations.

3.3.2 Validity Sensitive Querying

The problem of querying potentially invalid XML documents with respect to DTD schemata is focused in [48]. Although the main goal of this paper is the framework for querying, its integral part is also the model for proposing repairs for structurally invalid documents.

Users querying databases of XML documents usually formulate their queries with at least some knowledge of a schema, the given documents conform to. If documents are not valid, standard queries evaluation may not provide as sufficient answers as they could be returned, if the evaluation algorithm also attempts to propose some corrections of invalid data.

The main idea adopted by this thesis from the querying framework is the notion of a trace graph, which is capable to compactly represent all repairs of a sequence of child nodes of a given inspected node.

Edit and Update Operations

XML documents are modelled similarly to the previously discussed papers via unranked ordered labelled trees including the fact that attributes are completely ignored. However, the numbering schema is not the integral part of trees and we only use it for determining positions of nodes.

A DTD schema is viewed as a function assigning to each declared element name the allowed content model, thus a regular expression over element names. Similarly the mixed content is not handled and since only a local tree grammar is considered, we do not need to distinguish between terminal and nonterminal symbols.

Having a locally invalid node in a tree, we can use two basic macro operations for correcting its child nodes. These operations are analogous to update operations from the previous work and they are again composed from sequences of elementary edit operations. These are, however, only a leaf node insertion and removal at the level of edit operations and thus a new subtree

insertion, an existing subtree deletion or recursive repairing at the level of update operations. The recursive repairing however does not permit a node label modification.

The authors remark that it would be possible to extend the set of introduced operations with a rename label edit operation, but the paper does not work with it.

Each edit operation is assigned a unit cost, the cost of macro operations is derived as a sum of costs of all edit operations in a given sequence. The distance between two trees and between a tree and a schema is based on this cost function.

Restoration Graph

The provided XML tree is processed in a bottom up manner. Starting with leaves we continue towards the root node and for each locally invalid node we construct the restoration graph capable to compactly represent all possible repairs.

The main idea of sibling nodes correction is once again based on the exploration of the automaton for detecting words of the language defined by a provided content model regular expression. However, we do not dynamically generate word by word, but we statically construct the restoration graph and store all allowed corrections within it.

It is worth noting that the notion of this graph has a very close connection to the edit distance matrix presented in [14]. Vertices of this graph are divided into disjoint sets called columns and have a connection to rows of the distance matrix, thus represent the range of original sequence of nodes already involved into the correction. Each column is comprised from all states of the automaton for the recognition of the given regular expression. If the original sequence has n nodes, then the graph has $n + 1$ columns.

Edges between these vertices represent macro operations that can be performed. Between vertices inside a column we define edges representing insertions of new subtrees and between two adjacent columns we define edges representing deletions of existing subtrees or recursive repairs, depending on the correspondence of automaton transitions and node labels from the sequence to be processed.

Having constructed such restoration graph and assigned costs to all its edges, we can transform the problem of finding corrections to the problem of finding shortest paths in this graph. Each path starting in the first column at the initial state of the automaton and ending in the last column at any accepting state of the automaton represents the correction proposal. The shortest paths logically stand for corrections with the minimal cost.

Approach Assessment

The essential advantage of this approach lies in avoiding the dynamical traversal of the automaton state space and the generation of word by word. We can

directly focus the attention on processing only those directions of correction that seem perspective. We even do not need to construct the entire restoration graph, because we can use for example standard Dijkstra's algorithm and construct only the required parts of the graph.

Another advantage is that we can compactly represent all found corrections in a form of the trace graph, thus the subgraph of the restoration graph with only the shortest paths.

Finally, the processing is not limited by any threshold or pruning. Under any circumstances the algorithm is able to find corrections, which particularly means that it returns all corrections with the minimal cost. However, the introduced set of edit operations is obviously not sufficient to fulfil expectations of rational correction possibilities.

3.3.3 Correctors for XML Data

Authors in the series of papers [44, 9, 10] adopt the idea of correctors used in the theory of algorithm verification and use it in the context of XML documents correction. This approach represents the last analysed one, which pursues the problem of structural corrections of well formed but not valid documents.

A corrector is a function that transforms a provided potentially invalid XML document into a valid one, however, only under the condition that the original tree is not too far from the schema. Depending on a particular presented algorithm with the connection to permitted edit operations this distance needs to be constant or at most linear according to the particular schema.

An associated tester is capable to detect the approximate distance of a provided document to a schema with high probability in constant time, however, only under the same condition.

XML documents are represented using ranked ordered labelled trees, which are based on a set of nodes and associated relations for catching the structure of trees. Once again the paper only considers a DTD schema, modelled as a function returning content model regular expressions for declared element names.

The authors presented different models of allowed edit operations used for trees transformations and corrections. The first model is based on elementary operations for inserting a leaf node, deleting a leaf node and renaming an existing node. The second model is enriched by an operation for an atomic move of entire subtrees. Anyway, by the combination of these elementary operations we can acquire complex operations for inserting or deleting whole subtrees.

The measuring distances between trees are derived from the standard tree edit distance counting number of renamed labels and inserted or deleted nodes and edges.

The correction algorithm works in two basic phases. We first traverse the tree from leaves towards the root node in order to mark all nodes that are not locally valid. These nodes are marked using symbol $*$ behaving similarly as a wildcard. Other nodes are marked by their labels, since only a DTD schema

is taken into account.

During the second phase we process the tree from the root node towards leaves and recursively attempt to correct the neighbourhood of nodes marked with $*$. These corrections are based on finding new suitable labels for the invalid node and for each such label we attempt to locally correct the sequence of child nodes using introduced set of operations. We finally select such alternative, where the regular expression is the closest to the original sequence of nodes, and then we select the particular transformation with the lowest cost.

Although the authors provided a prototype implementation available at [8], the entire framework for corrections is not sufficiently transparent and especially the main disadvantage lies in the restriction that only documents with small number of errors can be corrected.

3.3.4 Repairs and Consistent Answers

Authors of [27, 28] proposed a framework for repairing XML documents with respect to a given set of integrity constraints. In these papers we, therefore, assume that provided documents are well formed and also valid with respect to a tree grammar, but we focus on functional dependencies between data values stored in a tree.

Proposed repairs are based on two combined concepts. First we can replace values of attributes or elements and second we can use the boolean function stating the reliability of nodes in a tree.

The functional dependencies are derived from the notion of tree tuples and paths. A path is a sequence of element and attribute names, which can be derived from the provided schema in a way of potential occurrence in a document valid with respect to this schema. Each path starts with a name of an element that is allowed to become the root node of a tree. All adjacent pairs in a path must conform to declarations of elements and attributes and, finally, we end in an attribute or element with only empty model permitted.

Now assume that we have a set of all paths that can be derived from a provided schema. A tree tuple is a maximal subtree such that for every existing path there exists at most one element in a given XML tree. The notion of the tree tuple can be used for determining elementary data components for which we can define functional dependencies.

These dependencies are again based on paths and we can use them to express demands on data values equalities. Generally we prefer marking incorrect data as unreliable rather than removing them permanently, because the deletion leads to the loss of information originally stored in such tree.

Having a particular functional dependency and incorrect values, we can change these values in case we are sure of the right correction, we can use the special symbol for unknown values or, finally, we can mark nodes as unreliable. Whenever a given node becomes unreliable, all descendant nodes in its subtree are automatically treated as unreliable too.

The algorithm successively processes all provided functional dependencies,

propose local repairs and merge these repairs into global ones. Having found all repairs we can extend the concept of the standard querying by the notions of possible and certain answers. Possible answers are those answers, which are involved at least in one possible repair, certain answers have to be located in all found repairs.

3.3.5 Repairing Documents using Chase

Similarly to the approach presented in the previous subsection authors in [51, 52] introduced a framework for repairing well formed and valid XML documents with respect to a given set of integrity constraints. These constraints, however, have wider expressive possibilities.

More particularly using them we are able to introduce constraints based on functional dependencies, keys and multivalued dependencies. The constraints themselves are structured expressions over variables and simple paths derived from the schema similarly to the previous approach.

In the first phase we construct a mend for the provided inconsistent XML document. Processing each provided constraint we inspect the tree and remove all conflicting data. This means that we can change values of nodes in a tree, replace constant values with variables and also delete nodes.

During the second phase we add new information into the prepared mend. We can change values of nodes, replace variables by other variables or by constants and, finally, we can insert new nodes. The resulting tree represents a repair.

It is easy to see that the algorithm can find more possible repairs. In these cases we select such repair that represents the greatest lower bound of all found repairs.

Authors of these papers also demonstrated that the strategy based on the successive creation of mends and only then repairs is not worse than a strategy producing fixes, thus the strategy that can mix all introduced operations without limits. Not only removal operations in the first phase and insertion in the second one.

Chapter 4

Corrections

Having discussed advantages and disadvantages of existing theoretical approaches especially for structural corrections of XML documents, we can move forward and focus on the detailed introduction of proposed correction framework.

First we will informally outline the entire model of corrections and in the remaining sections we will formally define all aspects of this model and proposed algorithms.

4.1 Framework Concept

The correction framework proposed in this thesis is capable to generate local structural repairs for locally invalid elements. These repairs are motivated by the classic Levenshtein metric for strings. For each node in a given XML tree and its sequence of child nodes we attempt to efficiently inspect new sequences that are allowed by the corresponding content model and that can be derived using the extended concept of measuring distances between strings. However, in our case we do not handle ordinary strings, but sequences of nodes, which in fact are not only labels, but also entire subtrees.

We pursue corrections for attributes and elements, however, only separately, i.e. we do not permit switching of attributes and elements mutually. All corrections are proposed in order to achieve valid documents with respect to a provided schema at the level of the single type tree grammar class. Although the framework primarily serves for the correction of the whole data trees, there is no problem using it in the context of the incremental validation for repairing only updated subtrees.

The correction algorithm starts processing at the root node and recursively moves towards leaf nodes. We assume that we have the complete data tree loaded into the system memory and, therefore, we have access to all its parts. Under all conditions the algorithm is able to find all minimal repairs, i.e. repairs with the minimal distance to the grammar and the original data tree according to the introduced cost function. Even though the algorithm does not consider any interaction with the user, it is possible to modify it.

In the following paragraphs we will shortly discuss all important aspects of the formal framework and correction algorithms. All these models and notions will be in detail described and formally defined in the remaining sections of this chapter.

Edit Operations

Edit operations are elementary transformations that are used for altering invalid data trees into valid ones. They behave as deterministically defined functions, performing small local modifications with a provided data tree. Despite the correction algorithm does not directly generate sequences of these edit operations, we can in the end acquire them using a translation of generated repairs, as it will be explained later.

For correcting attributes we have proposed edit operations for a new attribute insertion and an existing attribute deletion or rename. For nodes in a data tree there are operations capable to insert a new leaf node, delete an existing one, rename a label of a node, push a group of adjacent sibling nodes lower under a newly inserted internal node and, finally, an edit operation that pulls all sibling nodes one level higher deleting their original parent node.

Note that these operations are designed to correct attributes and elements separately.

Update Operations

Edit operations can be put together into sequences. And if these sequences fulfil certain qualities, they can be classified as update operations. We have proposed an insertion, deletion and repair elementary update operations for correcting attributes and a new subtree insertion, an existing subtree deletion, recursive repair of a subtree with an option of changing a label of its root node and, finally, push and pull operations based on corresponding edit operations. Update operations for transforming nodes are called complex update operations.

Anyway, the purpose of each update operation is to correct a local part of a data tree in order to achieve its local validity. Unfortunately the correction algorithm does not generate these operations. The algorithm generates repairs based on repairing instructions, which are subsequently translated into sequences of edit operations. And this is the reason, why update operations are not defined deterministically similarly to edit operations. Having a particular sequence of edit operations, we can inspect its segments and if all required conditions are satisfied, the given sequence can be viewed as an update operation of a corresponding type.

Repairing Instructions

Assume that we are in a particular node in a data tree and our goal is to locally correct the given node, which, passing over attributes, especially involves the

correction of the sequence of its child nodes.

Our objective is to find all minimal repairs for this inspected node. Since the introduced model for measuring distances uses only non-negative values for the cost function, in order to acquire the global optimum, we can simply find minimal combinations of local optimums, meaning minimal repairs for all subtrees of original child nodes of the inspected one.

However, we need to find all minimal repairs and because edit operations require particular positions in a current data tree to be specified, we cannot use them to describe all repairs. Assume for example that we have several options how to correct the first child node. If we delete it, all positions of nodes to the right must be shifted by one to the left, but if we accept the first child node, original positions preserve. Thus we are not able to use edit operations for describing multiple different repairing sequences.

The problem with the continuously changing numbers of positions is solved by the model of repairing instructions. We have exactly one instruction for each edit operation and these instructions represent the same transforming ideas, however, do not include particular positions to be applied on. Having a particular sequence of repairing instructions, we can easily translate it into the corresponding sequence of edit operations.

Correction Intents

Being in a particular node and repairing its sequence of child nodes, the correction algorithm generally has many ways to achieve the local validity proposing repairs for all these involved nodes. As already outlined, these actions follow the model of measuring distances between ordinary strings. The Levenshtein metric is defined as the minimal number of required elementary operations to transform one string into another. These operations are an insertion of one new symbol, deletion of an existing one and also replacement of an existing symbol with a new one.

We follow the same model, however, we have edit and update operations respectively and sequences of nodes. It is easy to see that the given sequence can be viewed as an ordinary string over labels of its nodes. For example, an insertion of a new subtree at a given position stands for the insertion of its label into the corresponding string of labels and of course recursive processing of such new subtree.

The algorithm attempts to examine all suitable new words that are in the language of the provided regular expression restraining the content model of the inspected parent node. We do not generate word by word, but we attempt to inspect all these words statically using a notion of an exploration and other derived multigraphs.

Anyway suppose that the algorithm has already processed first few nodes from the inspected sequence of sibling nodes, thus all nodes from the corresponding prefix of the original sequence are already involved into corrections. Now the algorithm must consider all possible actions that can be selected

in order to involve at least one next node from the original sequence. These possibilities are formally modelled using the notion of correction intents.

In other words, the correction algorithm in each parent node has a variety of options how to achieve its validity and particular steps performed with its child nodes are called correction intents. Intents because we always examine one possible action from more permitted ones.

Repairing Multigraphs

All existing correction intents in the context of a given node can be modelled using an exploration multigraph. Suppose that we need to process a sequence of child nodes with n nodes. This means that the graph will have $n + 1$ strata, numbered from 0 to n. Being on a stratum with number i, we have already processed right i first nodes from this sequence.

Except one small exception, which can be ignored for this moment, each stratum is constructed from the Glushkov automaton for recognising the provided regular expression restricting the given sequence. This means that there are vertices corresponding to states of the automaton and directed edges reflecting the transition function in each stratum. Each such edge represents a new tree insertion operation. Similarly, we can define edges between strata to represent other allowed operations.

In other words, the exploration multigraph represents all correction intents that can be derived for this sequence. And more precisely, each intent is represented by some edge in this multigraph. However, there can be intents that are represented by more edges.

In order to find best repairs for a provided sequence of nodes, we need to find all shortest paths in this multigraph, assuming that every edge is rated with an overall cost of corresponding nested correction intent associated with such edge. However, to resolve these costs, we need to fully evaluate associated intents. And this represents nontrivial nested recursive computations. Anyway, we require that each edge can be evaluated in a finite time, otherwise we would obviously not be able to find required shortest paths.

Repairs Construction

Each correction intent can essentially be viewed as an assignment to the nested recursive processing. This model in fact has a transparent relation with a structure of an underlying tree itself and its processing from the root node towards leaves. The entire correction of a provided data tree is initiated as a special starting correction intent for root node and processing of every intent always involves the construction of at least the required part of the introduced multigraph with other nested intents.

Therefore, we continuously invoke recursive computations of nested intents. When we reach the bottom of the recursion, we start backtracking, which involve gathering of found repairs. This means that after we have found the

desired shortest paths at a given level, we encapsulate them in a form of a repair structure and pass it one level up, towards the starting correction intent.

Having found shortest paths in the repairing multigraph for the starting intent, we have found repairs for the entire data tree. Each intent repair contains encoded shortest paths and related repairing instructions. Now we need to generate particular sequences of repairing instructions and translate them into standard sequences of edit operations. Having one such edit sequence, we can apply it on the original data tree and we obtain its valid correction with a minimal distance.

Correction Algorithms

Now we have completely outlined the model of the proposed correction framework. However, there are several related efficiency problems that would cause significantly slow behaviour, if we would strictly follow the formal model. Therefore, we will introduce four correction algorithms. They all produce the same repairs, but there are key differences in their efficiency.

First we start with a naive algorithm strictly following the formal model. Later on we will introduce a dynamic algorithm, which is able to directly search for shortest paths inside each intent computation and does not need the entire multigraphs to be constructed. The next improvement is based on caching already computed repairs, which causes that a caching algorithm never computes the same thing twice. The final extension is represented by an incremental algorithm, which is able to compute lazily even to the depth of the recursion.

4.2 Edit Operations

The first essential step in the description of proposed corrections approach is an introduction of transformations that can be used in order to correct provided data trees with respect to given schemata. The model of these modifications performed on data trees is based at two different levels of abstraction. The first level stands for elementary operations, whereas the second level represents some sort of macro operations.

The former type of operations is called *edit* operations, the latter one *update* operations. As already sketched, edit operations are elementary operations which represent basic alterations with a local impact. Formally, we define edit operations as partial functions, which are capable to transform the source data tree into another data tree.

On the other hand, update operations are more complex and represent transformations with potentially great and global impact on data trees. While edit operations are exactly defined, update operations are in fact only macro operations, i.e. complex operations that are composed from sequences of edit operations.

Auxiliary Sets of Nodes

Before we will formally define the set of proposed edit operations, we need to introduce several auxiliary sets of nodes. Doing it, we can make edit operations a little bit more readable.

Definition 4.1 (Auxiliary Sets of Nodes)**.** *Given an underlying tree D, we define $PosNodes$ as an auxiliary set of nodes:*

- $PosNodes(D) = \{u.i \mid i \in \mathbb{N}_0,\ u.i \notin D,\ u \in D$ *and either* $i = 0$ *or* $i > 0$ *and* $u.(i-1) \in D\}$.
  If D is an empty tree, then $PosNodes(D) = \{\epsilon\}$.

Given a tree D and a position $p \in D$, $p \neq \epsilon$, $p = u.i$, $u \in \mathbb{N}_0^*$, $i \in \mathbb{N}$, we define $ExpNodes$, $IncNodes$ and $DecNodes$ as sets of nodes:

- $ExpNodes(D, p) =$
  $\{u.k.v \mid k \in \mathbb{N}_0,\ i \leq k < fanOut(u),\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.
- $IncNodes(D, p) =$
  $\{u.(k+1).v \mid k \in \mathbb{N}_0,\ i \leq k < fanOut(u),\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.
- $DecNodes(D, p) =$
  $\{u.(k-1).v \mid k \in \mathbb{N}_0,\ i+1 \leq k < fanOut(u),\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.

Given a tree D, $s \in \mathbb{N}_0$ and a position $p \in D$, $p \neq \epsilon$, $p = u.i$, $u \in \mathbb{N}_0^*$, $i \in \mathbb{N}$, we define $ShlNodes$, $ShrNodes$, $PshNodes$ and $PulNodes$ as sets of nodes:

- $ShlNodes(D, p, s) =$
  $\{u.(k-s).v \mid k \in \mathbb{N}_0,\ i < k < fanOut(u),\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.
- $ShrNodes(D, p, s) =$
  $\{u.(k+s).v \mid k \in \mathbb{N}_0,\ i < k < fanOut(u),\ v \in \mathbb{N}_0^*,\ u.k.v \in D\}$.
- $PshNodes(D, p, s) =$
  $\{u.i.k.v \mid k \in \mathbb{N}_0,\ 0 \leq k \leq s,\ v \in \mathbb{N}_0^*,\ u.(i+k).v \in D\}$.
- $PulNodes(D, p) =$
  $\{u.(i+k).v \mid k \in \mathbb{N}_0,\ 0 \leq k < fanOut(p),\ v \in \mathbb{N}_0^*,\ u.i.k.v \in D\}$.

For a given data tree, $PosNodes$ stands for a set of all positions, where we can perform insertion of a new leaf node. These positions correspond to all existing nodes in a tree, positions after the last child of each node and, finally, positions for the first child node in current leaf nodes.

When we are inserting a new subtree at a given position in a data tree, we need to shift all original subtrees of sibling nodes located to the right from the insertion position by one to the right. Analogously when we are deleting a subtree, we need to shift all these sibling subtrees by one to the left. Obsolete positions for these shifts are defined in $ExpNodes$, new positions after shifting are in $IncNodes$ and $DecNodes$ respectively.

Sets $ShlNodes$ and $ShrNodes$ stand for sets of analogously shifted positions, but in this case we do not include the original base position and the shifting itself has a variable size. The last two sets $PshNodes$ and $PulNodes$ together with both previously described sets are used in conjunction with insertions or deletions of internal nodes, where group of nodes are lifted up or pressed down.

Edit Operations Model

We are now ready to formally introduce the set of proposed edit operations. To make the situation clearer, these operations are divided into two disjoint sets. One set is for operations over attributes, the second one contains operations for nodes.

In order to simplify the following definitions, we use a special notation of the form $f_1(p_1) \leftarrow f_0(p_0)$, where f_1, f_0 are functions and p_1, p_0 are values from corresponding domains, with this meaning: if $f_0(p_0)$ is defined, then we define $f_1(p_1) = f_0(p_0)$, else $f_1(p_1)$ remains undefined.

Definition 4.2 (Edit Operations)**.** *An* edit operation e *is a partial function that transforms a given data tree* $\mathcal{T}_0 = (D_0, lab_0, val_0, att_0)$ *into a new data tree* $\mathcal{T}_1 = (D_1, lab_1, val_1, att_1)$*, shortly denoted by* $\mathcal{T}_0 \xrightarrow{e} \mathcal{T}_1$*.*

For each edit operation we define $impactedNodes(e)$ *as a function returning a subset of* $\mathbb{N}_0^*$ *representing a set of directly impacted nodes.*

Before we continue, we have to emphasise that edit operations are functions, which are not generally defined for all data trees. If the provided prerequisites are not satisfied, the corresponding operation is undefined. We will start with operations for nodes and continue with operations for attributes. Anyway we will not consider any other operations than those defined in the following subsections.

4.2.1 Edit Operations for Nodes

We are going to define $addLeaf$, $removeLeaf$, $renameLabel$, $addNode$ and $removeNode$ edit operations for nodes. They represent insertion of a new leaf into a given data tree, deletion of an existing leaf node, changing the label of an existing node, insertion of an internal node and, finally, deletion of an internal node. Since the majority of these operations are little bit complicated, we will usually describe operations on the root node position separately to increase legibility.

The first two operations are also introduced in the correction approach proposed in [48], whereas authors in [14] append also the third named operation. We have adopted these operations, but since we have significantly different formal model for data trees, the corresponding definitions had to be adjusted.

Formal Model of Operations

Definition 4.3 (Node Edit Operations)**.** *Given data trees $\mathcal{T}_0$ and $\mathcal{T}_1$ from Definition 4.2, we define the following* node edit operations*:*

- $e = addLeaf(p, n)$ *for* $p \in D_0 \cup PosNodes(D_0)$, $p \neq \epsilon$ *and* $n \in \mathbb{E}$ *as a node label (not necessarily* $n \in attsDomain(D_0)$*). We assume that* $p = u.i$*, where* $u \in \mathbb{N}_0^*$ *and* $i \in \mathbb{N}_0$*. We require that* $u \notin DataNodes(D_0)$*.*
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p)] \cup IncNodes(D_0, p) \cup \{p\}$.
  - $\forall w \in D_0 \setminus ExpNodes(D_0, p)$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a position* p *we define:*
    - $lab_1(p) = n$.
    - *If* $lab_1(p) = \texttt{data}$*, then* $val_1(p) = \perp$.
    - *If* $lab_1(p) \neq \texttt{data}$*, then* $att_1(p) = \emptyset$.
  - $\forall (u.(k+1).v) \in IncNodes(D_0, p)$ *with appropriate* k, v:
    - $f_1(u.(k+1).v) \leftarrow f_0(u.k.v)$ *for* $f \in \{lab, val, att\}$.
  - $impactedNodes(e) = \{p\}$.
- $e = addLeaf(p, n)$ *for* $p = \epsilon$*,* n *a node label,* $D_0 = \emptyset$.
  - $D_1 = \{p\}$.
  - $lab_1(p) = n$.
  - *If* $n = \texttt{data}$*, then* $val_1(p) = \perp$*, else* val_1 *is undefined.*
  - att_1 *is undefined.*
  - $impactedNodes(e) = \{p\}$.
- $e = removeLeaf(p)$ *for* $p \in LeafNodes(D_0)$, $p \neq \epsilon$*. We assume that* $p = u.i$*, where* $u \in \mathbb{N}_0^*$ *and* $i \in \mathbb{N}_0$*. We require* $att_0(p) = \emptyset$.
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p)] \cup IncNodes(D_0, p)$.
  - $\forall w \in D_0 \setminus ExpNodes(D_0, p)$
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - $\forall (u.(k-1).v) \in DecNodes(D_0, p)$ *with appropriate* k, v:
    - $f_1(u.(k-1).v) \leftarrow f_0(u.k.v)$ *for* $f \in \{lab, val, att\}$.
  - $impactedNodes(e) = \{p\}$.
- $e = removeLeaf(p)$ *for* $p = \epsilon$, $p \in LeafNodes(D_0)$*. We require that* $att_0(p) = \emptyset$.
  - $D_1 = \emptyset$.
  - lab_1, val_1, att_1 *are not defined anywhere.*

- $impactedNodes(e) = \{p\}$.

- $e = renameLabel(p, n)$ *for* $p \in D_0$ *and* $n \in \mathbb{E}$ *as a node label (not necessarily* $n \in attsDomain(D_0)$*), supposing that* $n \neq lab_0(p)$.
  - $D_1 = D_0$.
  - $\forall w \in [D_0 \setminus \{p\}]$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a position* p *we define:*
    - $lab_1(p) = n$.
    - *If* $lab_1(p) = \texttt{data}$*, then* $val_1(p) = \perp$.
    - *If* $lab_1(p) \neq \texttt{data}$ *and* $lab_0(p) = \texttt{data}$*, then* $att_1(p) = \emptyset$.
    - *If* $lab_1(p) \neq \texttt{data}$ *and* $lab_0(p) \neq \texttt{data}$*, then* $att_1(p) = att_0(p)$.
  - $impactedNodes(e) = \{p\}$.

- $e = addNode(p_1, p_2, n)$ *for* $p_1, p_2 \in D_0$, $p_1, p_2 \neq \epsilon$, $p_1 = u.i$, $p_2 = u.j$, *where* $u \in \mathbb{N}_0^*$ *and* $i, j \in \mathbb{N}_0$, $i \leq j$. *Next* $n \in \mathbb{E}$, $n \neq \texttt{data}$ *is a node label (not necessarily* $n \in attsDomain(D_0)$*).*
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p_1)] \cup$
    $[ShlNodes(D_0, p_2, j - i) \cup PshNodes(D_0, p_1, j - i]$.
  - $\forall w \in [D_0 \setminus ExpNodes(D_0, p_1)]$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a newly added node on position* p_1*:*
    - $lab_1(p_1) = n$
    - *If* $lab_1(p_1) = \texttt{data}$*, then* $val_1(p_1) = \perp$
    - *If* $lab_1(p_1) \neq \texttt{data}$*, then* $att_1(p_1) = \emptyset$
  - $\forall (u.i.k.v) \in PshNodes(D_0, p, j - i)$ *with appropriate* k, v*:*
    - $f_1(u.i.k.v) \leftarrow f_0(u.(i + k).v)$ *for* $f \in \{lab, val, att\}$.
  - $\forall (u.(k - s).v) \in ShlNodes(D_0, p, s - 1)$ *with appropriate* k, v*:*
    - $f_1(u.(k - s).v) \leftarrow f_0(u.k.v)$ *for* $f \in \{lab, val, att\}$.
  - $impactedNodes(e) = \{u.k \mid k \in \mathbb{N}_0, i \leq k \leq j\}$.

- $e = addNode(p_1, p_2, n)$ *for* $p_1, p_2 = \epsilon$, $\epsilon \in D_0$ *and* $n \in \mathbb{E}$, $n \neq \texttt{data}$ *is a node label (not necessarily* $n \in attsDomain(D_0)$*).*
  - $D_1 = \{0.w \mid w \in D_0\} \cup \{\epsilon\}$.
  - $\forall w \in D_0$:
    - $f_1(0.w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a newly added root node:*
    - $lab_1(\epsilon) = n$.

- $\cdot$ $att_1(\epsilon) = \emptyset$.
- $-$ $impactedNodes(e) = \{\epsilon\}$.

- $e = removeNode(p)$ *for* $p \in D_0$, $p \notin LeafNodes(D_0)$, $p \neq \epsilon$. *We assume that* $s = fanOut(p)$ *and* $p = u.i$, *where* $u \in \mathbb{N}_0^*$ *and* $i \in \mathbb{N}_0$.
  - $D_1 = [D_0 \setminus ExpNodes(D_0, p)] \cup$ $[ShrNodes(D_0, p, s-1) \cup PulNodes(D_0, p)]$.
  - $\forall w \in [D_0 \setminus ExpNodes(D_0, p)]$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - $\forall (u.(i+k).v) \in PulNodes(D_0, p)$ *with appropriate* k, v:
    - $f_1(u.(i+k).v) \leftarrow f_0(u.i.k.v)$ *for* $f \in \{lab, val, att\}$.
  - $\forall (u.(k+s).v) \in ShrNodes(D_0, p, s-1)$ *with appropriate* k, v:
    - $f_1(u.(k+s).v) \leftarrow f_0(u.k.v)$ *for* $f \in \{lab, val, att\}$.
  - $impactedNodes(e) = \{p\}$.

- $e = removeNode(p)$ *for* $p = \epsilon$, $p \in D_0$, $p \notin LeafNodes(D_0)$. *We require that* $fanout(p) = 1$.
  - $D_1 = \{v \mid 0.v \in D_0,\ v \in \mathbb{N}_0^*\}$.
  - $\forall (0.v) \in D_0$ *with appropriate* v:
    - $f_1(v) \leftarrow f_0(0.v)$ *for* $f \in \{lab, val, att\}$.
  - $impactedNodes(e) = \{p\}$.

Description of Defined Operations

The first defined operation *addLeaf* inserts a new leaf node at a position, that allows such insertion. We require, that we do not nest this new leaf under a data node, since data nodes cannot be internal nodes. The label of a new node can be equal to some already known name, it can be either a newly introduced name or, finally, it can be a special name `data`, that is used for data nodes. In this case, the value is set to the undefined value $\perp$, since we are actually not interested in values themselves in our XML trees model and correction approach.

The *removeLeaf* operation deletes the existing leaf node from the data tree. Before we can apply such operation, the given node cannot have any defined attribute. These attributes have to be removed before.

The third operation *renameLabel* lies in the replacement of a node label. If we want to apply this operation, we really have to change the label. It is not allowed to replace it with the same original value. Furthermore we have to follow the condition for data nodes, since this operation as a consequence allows also changing the type of a node from a data node to an element node and vice versa.

The last but one operation *addNode* stands for the insertion of a new internal node. For doing it, we have to specify the valid range of sibling nodes, which will be moved one level lower and later on will be nested under a newly added internal node with a specified label. Since it is expected that this range is not empty, this label cannot be equal to `data`. The insertion of an internal node with no descendants, i.e. specification of an empty range, is not permitted, because it would lead to the standard leaf node insertion.

Finally, we will shortly describe the *removeNode* operation. It causes the deletion of an internal node and later on the relocation of all its original child nodes by one level upwards between sibling nodes of the former parent. This operation requires, that the base node is really an internal node, otherwise it would take the same effect as a simple remove leaf node operation. To make the description complete, before the node is deleted, all attributes must be removed.

4.2.2 Edit Operations for Attributes

The second type of edit operations are those for manipulation with attributes. Their definition is the subject of this subsection.

Formal Model of Operations

Definition 4.4 (Attribute Edit Operations). *Given data trees $\mathcal{T}_0$ and $\mathcal{T}_1$ from Definition 4.2, we define the following* attribute edit operations.

Furthermore we define a function impactedAtts assigning to each attribute edit operation e a set of names of impacted attributes.

- $e = addAttribute(p, a)$ *for* $p \in D_0$, $p \notin DataNodes(D_0)$ *and* $a \in \mathbb{A}$ *is an attribute name. We require that* $(a, v) \notin att_0(p)$ *for some appropriate* $v \in \mathbb{V}$.
  - $D_1 = D_0$.
  - $\forall w \in [D_0 \setminus p]$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a position p we define:*
    - $lab_1(p) = lab_0(p)$.
    - $att_1(p) = att_0(p) \cup \{(a, \perp)\}$.
  - $impactedNodes(e) = \{p\}$.
  - $impactedAtts(e) = \{a\}$.
- $e = removeAttribute(p, a)$ *for* $p \in D_0$, $p \notin DataNodes(D_0)$ *and* $a \in \mathbb{A}$ *is an attribute name. We require that* $(a, v) \in att_0(p)$ *for some appropriate* $v \in \mathbb{V}$.
  - $D_1 = D_0$.

- $\forall w \in [D_0 \setminus p]$:
  - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
- *For a position p we define:*
  - $lab_1(p) = lab_0(p)$.
  - $att_1(p) = att_0(p) \setminus \{(a, v)\}$.
- $impactedNodes(e) = \{p\}$.
- $impactedAtts(e) = \{a\}$.

- $e = renameAttribute(p, a_0, a_1)$ *for* $p \in D_0$, $p \notin DataNodes(D_0)$, a_0, $a_1 \in \mathbb{A}$, $a_0 \neq a_1$ *are attribute names. We require that* $(a_0, v_0) \in att_0(p)$ *and* $(a_1, v_1) \notin att_0(p)$ *for some appropriate* v_0, $v_1 \in \mathbb{V}$.
  - $D_1 = D_0$.
  - $\forall w \in [D_0 \setminus p]$:
    - $f_1(w) \leftarrow f_0(w)$ *for* $f \in \{lab, val, att\}$.
  - *For a position p we define:*
    - $lab_1(p) = lab_0(p)$.
    - $att_1(p) = [att_0(p) \setminus \{(a_0, v_0)\}] \cup \{(a_1, v_0)\}$.
  - $impactedNodes(e) = \{p\}$.
  - $impactedAtts(e) = \{a_0, a_1\}$.

Description of Defined Operations

Edit operations for attributes are quite simpler than operations for nodes. This is first of all because we do not change the structure of a given data tree, when we apply attribute edit operations. We only affect the set of pairs composed by attribute names and their values.

The first operation *addAttribute* inserts a new distinct attribute, operation *removeAttribute* deletes the existing attribute and, finally, the operation *renameAttribute* changes the name of some existing attribute. We are not forced to use only established words for new attribute names and generally we are not interested in data values of attributes. Thus we preserve them as long as they can be preserved and for newly added attributes we use a special value $\perp$ similarly to nodes themselves and their values.

4.2.3 Costs and Sequences of Operations

Our main goal is to find repairs of data trees that are as close to original trees as they can be. Thus we need to introduce some method of measuring these distances and a model for construction of more complex operations.

Sequences of Operations and Segmentation

The following definition sets up the notion of a sequence of edit operations, which can be understood as a next step towards update operations, which are built by a composition of edit operations.

Definition 4.5 (Sequence of Edit Operations)**.** *Given some $n \in \mathbb{N}_0$, let $\mathcal{T}_0, \ldots, \mathcal{T}_n$ be data trees and $e_1, \ldots, e_n$ edit operations, such that $\forall i \in \mathbb{N}_0$, $0 \leq i < n$: $\mathcal{T}_i \xrightarrow{e_{i+1}} \mathcal{T}_{i+1}$.*

We say that $\mathcal{T}_0 \xrightarrow{e_1} \mathcal{T}_1 \ldots \mathcal{T}_{n-1} \xrightarrow{e_n} \mathcal{T}_n$, simply denoted by $\mathcal{T}_0 \xrightarrow{\mathcal{S}} \mathcal{T}_n$ for $\mathcal{S} = \langle e_1, \ldots, e_n \rangle$, is a sequence of edit operations *transforming $\mathcal{T}_0$ into $\mathcal{T}_n$. A sequence is* empty*, if $n = 0$.*

Let $k_0, k_1 \in \mathbb{N}_0$, $1 \leq k_0 \leq k_1 \leq n$. Then a sequence $\langle e_{k_0}, \ldots, e_{k_1} \rangle$ is a subsequence *of the sequence $\mathcal{S}$.*

It will be shown in the next subsection, that update operations are in fact those sequences of edit operations that satisfy some restrictions. The notion of sequences segmentation helps us to describe such restrictions.

Definition 4.6 (Sequence Segmentation)**.** *Let $\mathcal{S} = \langle e_1, \ldots, e_n \rangle$ be a non-empty sequence of edit operations transforming $\mathcal{T}_0$ into $\mathcal{T}_n$, let $s \in \mathbb{N}$ and $1 = i_0, i_1, \ldots, i_{s-1}, i_s = n + 1$, $\forall k \in \mathbb{N}_0$, $0 \leq k < s$: $i_k \leq i_{k+1}$.*

We say, that $\mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ is a segmentation *of the sequence $\mathcal{S}$, if $\forall k$, $0 \leq k < s$: $\mathcal{P}_k = \langle e_{i_{(k-1)}}, \ldots, e_{(i_k)-1} \rangle$ is a subsequence called* segment*. Positive number s represents the number of segments.*

If $\mathcal{S}$ is an empty sequence, then $\mathcal{P} = \langle \rangle$ or the segmentation can lead to any number of empty segments.

Segments of edit operations are therefore ordinary subsequences of edit sequences. For technical reasons we also admit empty segments.

Costs of Operations and Sequences

Authors in [51, 52] based their idea of measuring edit distance on a definition of partial ordering of data trees, which is derived from the structure of trees themselves. This approach may be interesting from the theoretical point of view, but we have inclined to the model used for example in [14], i.e. a model based on costs assigned to performed operations.

Definition 4.7 (Edit Operations Costs)**.** *Given an edit operation e, we define $cost(e)$ to be a function assigning to an operation e its non-negative* cost*.*

The only restriction we put on the cost function is the non-negative range of values. We allow zero value, but negative values could potentially trigger problems in a phase of data tree correction. Thus we prohibit them.

It can be expected, that prospective implementations would pay nontrivial attention on searching the ideal configuration of the cost function. Moreover

they can also provide configurable environment, leaving the final adjustment up to the user.

The cost function can generally assign different values for different operations. However, it is a good idea to follow for example $cost(renameLabel) \leq cost(addLeaf) + cost(removeLeaf)$ restriction. In our approach we assume, that the cost function always returns value 1. The formal framework however does not assume this simplification. Having the cost function defined on single operations, we can extend this idea to sequences of edit operations.

Definition 4.8 (Cost of edit operations sequence)**.** *Assume that $\mathcal{S} = \langle e_1, \ldots, e_n \rangle$ is a sequence of edit operations transforming a data tree $\mathcal{T}_0$ into a data tree $\mathcal{T}_n$. We define a* cost *of a sequence of edit operations to be a function $cost(\mathcal{S}) = \sum_{i=1}^{n} cost(e_i)$. If $n = 0$, we define $cost(\mathcal{S}) = 0$.*

The cost of a sequence of edit operations is therefore equal to the sum of costs of all operations in such sequence.

4.3 Update Operations

Whereas edit operations are deterministic functions with a local impact on data trees, update operations are sequences of these primitive operations. But not each sequence can be treated as an update operation. We say, that the given sequence is an update operation of corresponding type, if that sequence fulfils all conditions that the definition of such update operation requires.

Assume that we have, for example, an update operation *insertTree*, which inserts a subtree into a data tree. Informally, any sequence doing this task correctly, is treated as an update operation *insertTree*. As a consequence, there can be many those sequences.

In order to restrain some strange behaviour, we restrict the description of several update operations more than it could be expected from the first point of view. For example it is not possible to use repeatedly nested operation for deleting internal nodes to achieve the deletion of the complete original subtree, because the same effect can be reached using the standard operation for deleting whole subtrees.

Definition 4.9 (Update Operations)**.** *Let $\mathcal{U} = \langle e_1, \ldots, e_n \rangle$ be a sequence of edit operations transforming a given data tree $\mathcal{T}_0 = (D_0, lab_0, val_0, att_0)$ into a new data tree $\mathcal{T}_n = (D_n, lab_n, val_n, att_n)$, thus in other words $\mathcal{T}_0 \xrightarrow{\mathcal{U}} \mathcal{T}_n$. We generally admit empty sequences to be update operations.*

The sequence $\mathcal{U}$ is an update operation *o, if it satisfies conditions of some elementary update operation or complex update operation.*

Furthermore we define $baseNode(o)$ to be a function, that assigns exactly one value from $\mathbb{N}_0^*$ to each update operation o.

We have two basic types of update operations. We will start with update operations for attributes. Their purpose is to provide transformations capable

of the complete correction of attributes in a given node. The second type represents complex update operations which perform structural modifications and which automatically involve invocation of potentially required elementary update operations for attributes.

Even the following definitions do not explicitly recall this fact, it is always required that a given sequence of edit operations is correctly defined. This means that each operation in a chain must be feasible, thus all prerequisites are correctly fulfilled in each step of the chain.

4.3.1 Elementary Update Operations

The first type of update operations represents elementary operations for manipulating attributes in a selected node. We have three basic strategies, that we can use depending on the initial conditions and with the connection to complex update operations in order to correct attributes.

Formal Model of Operations

Definition 4.10 (Elementary Update Operations)**.** *Given a sequence* $\mathcal{U} = \langle e_1, \ldots, e_n \rangle$ *and data trees* $\mathcal{T}_0$ *and* $\mathcal{T}_n$ *from Definition 4.9, we define the following* elementary update operations o*:*

- $o = insertAttributes(p)$ *for* $p \in \mathbb{N}_0^*$*:*
  - $p \in D_0$.
  - $\forall k \in \mathbb{N}$, $1 \leq k \leq n$: $e_k = addAttribute(p, a_k)$ *for some* $a_k \in \mathbb{A}$ *as an attribute name (optionally* $a_k \in attsDomain(D_{k-1})$*).*
  - $baseNode(o) = \{p\}$.
- $o = deleteAttributes(p)$ *for* $p \in \mathbb{N}_0^*$*:*
  - $p \in D_0$.
  - $\forall k \in \mathbb{N}$, $1 \leq k \leq n$: $e_k = removeAttribute(p, a_k)$ *for some* a_k *as an attribute name,* $(a_k, v) \in att(p)$.
  - $baseNode(o) = \{p\}$.
- $o = repairAttributes(p)$ *for* $p \in \mathbb{N}_0^*$*:*
  - $p \in D_0$.
  - $\forall k \in \mathbb{N}$, $1 \leq k \leq n$: e_k *satisfies one of the following:*
    - $e_k = addAttribute(p, a_k)$ *for some* $a_k \in \mathbb{A}$ *as an attribute name (not necessarily* $a_k \in attsDomain(D_{k-1})$*).*
    - $e_k = removeAttribute(p, a_k)$ *for some existing attribute name* $a_k \in attsDomain(D_{k-1})$.

    - $e_k = renameAttribute(p, (a_0)_k, (a_1)_k)$ *for some attribute names* $(a_0)_k$ *such that* $((a_0)_k, v) \in att(p)$ *and* $(a_1)_k \in \mathbb{A}$ *not necessarily from* $attsDomain(D_{k-1})$.
  - *Sequence* $\mathcal{U}$ *is without redundancies, i.e.* $\forall i, j \in \mathbb{N}$, $1 \leq i < j \leq n$: $impactedAtts(e_i) \cap impactedAtts(e_j) = \emptyset$.
  - $baseNode(o) = \{p\}$.

Description of Defined Operations

If we have inserted a completely new node into a data tree, we need to add all required attributes as well, in order to achieve local validity of this node. This is the meaning of the first update operation called *insertAttributes*. Conversely, the operation *deleteAttributes* removes all existing attributes from a given node. This operation has to be invoked before the given node itself can be removed from a tree. Finally, the operation *repairAttributes* attempts to correct the existing node, thus add missing and remove invalid attributes.

It is important to say that the particular implementation of these strategies is up to the correction routine. We have said that the operation *deleteAttributes* will be used in situations, when we need to delete all attributes from a given node, but the definition do not mention this demand.

Furthermore the definition also does not handle the order of such attribute edit operations in a sequence. It is easy to find out, that since attributes are modelled using a set paradigm, ordering is not relevant. Multiple sequences can therefore exist and bring the same effect applied on a node. We only require that we do not manipulate with one attribute repeatedly. This demand is broadly legitimate, because one edit operation on an attribute would cancel outcome of the previously performed one.

4.3.2 Complex Update Operations

Whereas elementary update operations handle only attributes, complex update operations manipulate both with the structure of a data tree and with attributes. The following definition gives the enumeration of all proposed complex operations. It is suitable to recall, that sequences of update operations can generally be empty.

Formal Model of Operations

Definition 4.11 (Complex Update Operations). *Given a sequence* $\mathcal{U} = \langle e_1, \ldots, e_n \rangle$ *and data trees* $\mathcal{T}_0$ *and* $\mathcal{T}_n$ *from Definition 4.9, we define the following* complex update operations o:

- $o = insertSubtree(p)$ *for* $p \in \mathbb{N}_0^*$:
  - $p \in D_0 \cup PosNodes(D_0)$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 2$ *such that:*

    - $\mathcal{P}_1 = \langle e_1 \rangle$ *and* $e_1 = addLeaf(p, l)$ *for some appropriate node label* $l \in \mathbb{E}$.
    - $\mathcal{P}_2 = insertAttributes(p)$, *where* $\mathcal{P}_2$ *can be empty.*
    - $\forall k \in \mathbb{N},\ 3 \leq k \leq s$: $\mathcal{P}_k = insertSubtree(p.(k-3))$.
  - $baseNode(o) = \{p\}$.

- $o = deleteSubtree(p)$ *for* $p \in \mathbb{N}_0^*$:
  - $p \in D_0$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 2$ *such that:*
    - $\forall k \in \mathbb{N},\ 1 \leq k \leq (s-2)$: $\mathcal{P}_k = deleteSubtree(p.0)$.
    - $\mathcal{P}_{s-1} = deleteAttributes(p)$, *where* $\mathcal{P}_{s-1}$ *can be empty.*
    - $\mathcal{P}_s = \langle e_n \rangle$ *and* $e_n = removeLeaf(p)$.
  - $baseNode(o) = \{p\}$.

- $o = repairSubtree(p)$ *for* $p \in \mathbb{N}_0^*$:
  - $p \in D_0$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 1$ *such that:*
    - $\mathcal{P}_1 = repairAttributes(p)$, *where* $\mathcal{P}_1$ *can be empty.*
    - $\forall k \in \mathbb{N},\ 2 \leq k \leq s$: $\mathcal{P}_k$ *is one of the following update operations for some appropriate* $u_k,\ v_k \in \mathbb{N}_0 \setminus \{\epsilon\}$:
      - $insertSubtree(p.u_k)$, $deleteSubtree(p.u_k)$,
      - $repairSubtree(p.u_k)$, $renameSubtree(p.u_k)$,
      - $pushSiblings(p.u_k, p.v_k)$ *or* $pullSiblings(p.u_k)$.
    - $\forall k \in \mathbb{N},\ 2 \leq k < s$: $baseNode(\mathcal{P}_k) \leq baseNode(\mathcal{P}_{k+1})$.
  - $baseNode(o) = \{p\}$.

- $o = renameSubtree(p)$ *for* $p \in \mathbb{N}_0^*$:
  - $p \in D_0$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 2$ *such that:*
    - $\mathcal{P}_1 = \langle e_1 \rangle$ *and* $e_1 = renameLabel(p, l)$ *for some appropriate node label* $l \in \mathbb{E}$.
    - $\mathcal{P}_2 = repairAttributes(p)$, *where* $\mathcal{P}_2$ *can be empty.*
    - $\forall k \in \mathbb{N},\ 3 \leq k \leq s$: $\mathcal{P}_k$ *is one of the following update operations for some appropriate* $u_k,\ v_k \in \mathbb{N}_0$:
      - $insertSubtree(p.u_k)$, $deleteSubtree(p.u_k)$,
      - $repairSubtree(p.u_k)$, $renameSubtree(p.u_k)$,
      - $pushSiblings(p.u_k, p.v_k)$ *or* $pullSiblings(p.u_k)$.
    - $\forall k \in \mathbb{N},\ 3 \leq k < s$: $baseNode(\mathcal{P}_k) \leq baseNode(\mathcal{P}_{k+1})$.
  - $baseNode(o) = \{p\}$.

- $o = pushSiblings(p_1, p_2)$ *for* $p_1, p_2 \in \mathbb{N}_0^*$*:*
  - p_1, $p_2 \in D_0$. *Either* p_1, $p_2 = \epsilon$ *or* $p_1 = p.i$, $p_2 = p.j$ *for some* $p \in \mathbb{N}_0^*$ *and* i, $j \in \mathbb{N}_0$, $i \leq j$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 2$ *such that:*
    - $\mathcal{P}_1 = \langle e_1 \rangle$ *and* $e_1 = addNode(p_1, p_2, l)$ *for some appropriate node label* $l \in \mathbb{E}$.
    - $\mathcal{P}_2 = repairAttributes(p_1)$, *where* $\mathcal{P}_2$ *can be empty.*
    - $\forall k \in \mathbb{N}$, $3 \leq k \leq s$*:* $\mathcal{P}_k$ *is one of the following update operations for some appropriate* u_k, $v_k \in \mathbb{N}_0$*:*
      - $insertSubtree(p.u_k)$, $deleteSubtree(p.u_k)$,
      - $repairSubtree(p.u_k)$, $renameSubtree(p.u_k)$,
      - $pushSiblings(p.u_k, p.v_k)$ *or* $pullSiblings(p.u_k)$.
    - $\forall k \in \mathbb{N}$, $3 \leq k < s$*:* $baseNode(\mathcal{P}_k) \leq baseNode(\mathcal{P}_{k+1})$.
  - $baseNode(o) = \{p_1\}$.
- $o = pullSiblings(p)$ *for* $p \in \mathbb{N}_0^*$, $p \neq \epsilon$, $p = u.i$ *for some* $u \in \mathbb{N}_0^*$ *and* $i \in \mathbb{N}_0$*:*
  - $p \in D_0$, $p \notin LeafNodes(D_0)$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s \geq 3$ *such that:*
    - $\mathcal{P}_1 = deleteAttributes(p)$
    - $\mathcal{P}_2 = \langle e_j \rangle$ *and* $e_j = removeNode(p)$ *for some* $j \in \mathbb{N}$.
    - $\forall k \in \mathbb{N}$, $3 \leq k \leq s$*:* $\mathcal{P}_k$ *is one of the following update operations for some appropriate* $j_k \in \mathbb{N}_0$*:*
      - $insertSubtree(u.j_k)$, $deleteSubtree(u.j_k)$,
      - $repairSubtree(u.j_k)$, $renameSubtree(u.j_k)$,
      - $pullSiblings(u.j_k)$.
    - *Let* $f_0 = fanOut(u)$ *in* D_0 *and* $f_n = fanOut(u)$ *in* D_n. *Then* $\forall j_k$ *from the previous condition the following formula* $i \leq j_k \leq i + (f_n - f_0) - 1$ *must hold.*
    - $\forall k \in \mathbb{N}$, $3 \leq k < s$*:* $baseNode(\mathcal{P}_k) \leq baseNode(\mathcal{P}_{k+1})$.
  - $baseNode(o) = \{p\}$.
- $o = pullSiblings(p)$ *for* $p = \epsilon$*:*
  - $p \in D_0$, $p \notin LeafNodes(D_0)$.
  - $\exists \mathcal{P} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_s \rangle$ *segmentation of* $\mathcal{U}$ *for* $s = 3$ *such that:*
    - $\mathcal{P}_1 = deleteAttributes(p)$.
    - $\mathcal{P}_2 = \langle e_j \rangle$ *and* $e_j = removeNode(p)$ *for some* $j \in \mathbb{N}$.
    - $\mathcal{P}_3$ *is one of the following update operations:*
      - $repairSubtree(p)$, $renameSubtree(p)$,
      - $pullSiblings(p)$.
  - $baseNode(o) = \{p\}$.

Description of Defined Operations

The first update operation is *insertSubtree*. It causes the insertion of a whole subtree on a given suitable position of the original data tree. The first step is always the addition of a root node of such subtree using the edit operation for adding leaf nodes. After it we optionally insert required attributes to this node and then we can continue in a top-down and left to right manner, recursively processing the rest of the tree to be inserted.

The operation *deleteSubtree* removes the entire subtree on a given position from a data tree. This deletion is performed in a left to right and bottom-up manner. Sibling nodes are thus removed from the left and we cannot remove a node before all its child nodes are already removed. The last but one step at a given level is therefore the removal of attributes and the last step deletes the node itself.

Operations *renameSubtree* and *repairSubtree* are practically identical, except that the former one invokes the rename label edit operation on a given position, whereas the latter one does not allow this modification. Either way the next step is the correction of attributes using the repair attributes elementary operation. Child nodes of the given node are always processed in a left to right manner and in case of these two operations we are not limited, which operations can be used recursively.

The operation *pushSiblings* selects the non-empty subsequence of sibling nodes, pushes them one level lower and inserts a new internal node into the tree on the former position of the first node in such sequence. This new node becomes a new parent node for these pushed siblings. The first step is always the insertion of a new parent followed by the insertion of attributes. All pushed nodes are subsequently processed recursively in a left to right manner allowing all defined update operations to be used.

Finally, the operation *pullSiblings* represents the deletion of an internal node followed by lifting up its original child nodes one level higher. First we need to remove potentially existing attributes and then we remove such internal node from a tree using the remove node edit operation. The next step represents processing of all fetched former child nodes. We handle them from left to right and we only cannot use nested *pushSiblings* operation. This restriction is quite strict, since the problem is in fact hidden in the consecutive application of the push siblings operation on the whole sequence of pulled nodes, not its proper subsequence.

General Problems of Update Operations

At first we recall once again, that all edit operations in a sequence must be correctly defined. Thus restrictions in the previous definition are only general and have to be respectively extended.

An update operation is by the definition a sequence of edit operations. Having some sequence, we can inspect it and declare it to be the corresponding update operation, if all declared conditions are fulfilled. The goal of update

operations is of course to provide the mechanism for corrections. Therefore, it is better to insight update operations as transactions. The consistency requirements are satisfied at the end, bud we cannot guarantee them inside the transaction.

Our model of update operations is mainly derived from the approach presented in [14] and related papers, but there is a significant difference. Update operations in this paper are defined deterministically in a way, how the presented correction algorithm works. Contrary to this attitude, our correction routine generates repairing instructions, which are mapped to defined update operations later on.

As already sketched in the previous paragraphs, we can permit modalities in some of the introduced operations. Moreover we can also provide the configurable environment to the user, letting up to him which operations should be used. But it is easy to see, that we need at least some minimal set of operations to be allowed, in order to ensure the correction routine will always find repairs.

Another problem lies in the computation of repairs, i.e. finding sequences reflecting introduced update operations. Both *insertSubtree* and *pushSiblings* allows generation of potentially infinite subtrees. The correction routine itself must restrict the semantics of these two operations and provide some sort of mechanism preventing recursion without the bottom. Other update operations do not cause this problem, since they only handle the existing finite information stored in a given data tree.

4.3.3 Costs and Sequences of Operations

Analogously to edit operations and their sequences we can introduce the notion of a sequence of update operations. Since updates are sequences of edit operations, sequences of update operations in fact represent ordinary sequences of edit operations.

Sequences and Cost

Definition 4.12 (Sequence of Update Operations)**.** *Given some $n \in \mathbb{N}_0$, let $\mathcal{T}_0$, $\dots$, $\mathcal{T}_n$ be data trees and $o_1, \dots, o_n$ are update operations, such that $\forall i \in \mathbb{N}_0$, $0 \le i < n$: $\mathcal{T}_i \xrightarrow{o_{i+1}} \mathcal{T}_{i+1}$. We say that $\mathcal{T}_0 \xrightarrow{o_1} \mathcal{T}_1 \dots \mathcal{T}_{n-1} \xrightarrow{o_n} \mathcal{T}_n$, simply denoted by $\mathcal{T}_0 \xrightarrow{\mathcal{U}} \mathcal{T}_n$ for $\mathcal{U} = \langle o_1, \dots, o_n \rangle$, is a* sequence of update operations *transforming $\mathcal{T}_0$ into $\mathcal{T}_n$.*

Now we are able to easily extend the cost function notion to the cost of sequences of update operations.

Definition 4.13 (Cost of Update Sequence)**.** *Assume that $\mathcal{U} = \langle o_1, \dots, o_n \rangle$ is a sequence of update operations $o_1, \dots, o_n$ transforming a data tree $\mathcal{T}_0$ into a data tree $\mathcal{T}_n$. We define a* cost *of a sequence of update operations as a function $cost(\mathcal{U}) = \sum_{i=1}^{n} cost(o_i)$. If $n = 0$, we define $cost(\mathcal{U}) = 0$.*

Sequences Equivalence

It has already been outlined, that there can exist different update sequences leading to the same target data tree. The first reason is hidden in the elementary update operations for attributes, because the nature of their occurrence in elements is based on sets and sets are not ordered contrary to sequences. We have therefore more possibilities how to arrange required attribute edit operations into one segment representing an elementary update operation, but the result will be the same.

However, we can obtain the same target data tree even if we use different complex update operations. For example using the *pullSiblings* operation we can achieve the complete subtree removal, but this removal can be done directly via *deleteSubtree*. Although the result is in both cases the same, it is not a good idea to treat such sequences as equivalent, since the correction intent is not the same. In fact not only formally, but these two sequences of edit operations can have different costs and thus they should be both considered.

The following definition introduces the update sequences equivalence relation reflecting the discussed opposing interests.

Definition 4.14 (Update Operations Equivalence)**.** *Two update operations* $e = \langle e_1, \ldots, e_n \rangle$ *and* $f = \langle f_1, \ldots, f_n \rangle$ *are equivalent,* $e \equiv f$*, if they are of the same type, the same length* $n \in \mathbb{N}_0$ *and all the following conditions hold for a given case of the update operation type.*

For complex update operations we assume that $\mathcal{P}^e = \langle \mathcal{P}^e_1, \ldots, \mathcal{P}^e_s \rangle$ *is the segmentation for* e *and analogously* $\mathcal{P}^f = \langle \mathcal{P}^f_1, \ldots, \mathcal{P}^f_s \rangle$ *for* f*, where both* $\mathcal{P}^e$ *and* $\mathcal{P}^f$ *conform to Definition 4.11.*

- $e = insertAttributes(p^e)$ *and* $f = insertAttributes(p^f)$*:*
  $e = deleteAttributes(p^e)$ *and* $f = deleteAttributes(p^f)$*:*
  $e = repairAttributes(p^e)$ *and* $f = repairAttributes(p^f)$*:*
  - $p^e = p^f$.
  - $\exists j_1, \ldots, j_n \in \mathbb{N}$ *permutation of indices* $1, \ldots, n$ *such that* $\forall i \in \mathbb{N}$, $1 \leq i \leq n$*:* $e_i = f_{j_i}$.
- $e = insertSubtree(p^e)$ *and* $f = insertSubtree(p^f)$*:*
  - $p^e = p^f$.
  - $e_1 = f_1 = addLeaf(p^e, l)$ *for some appropriate* $l \in \mathbb{E}$.
  - $\mathcal{P}^e_2 \equiv \mathcal{P}^f_2$.
  - $\forall k \in \mathbb{N}$, $3 \leq k \leq s$*:* $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*
- $e = deleteSubtree(p^e)$ *and* $f = deleteSubtree(p^f)$*:*
  - $p^e = p^f$.
  - $\forall k \in \mathbb{N}$, $1 \leq k \leq (s - 2)$*:* $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*

- $\mathcal{P}^e_{s-1} \equiv \mathcal{P}^f_{s-1}$.
- $e_n = f_n = removeLeaf(p^e)$.

- $e = repairSubtree(p^e)$ *and* $f = repairSubtree(p^f)$*:*
  - $p^e = p^f$.
  - $\mathcal{P}^e_1 \equiv \mathcal{P}^f_1$.
  - $\forall k \in \mathbb{N},\ 2 \le k \le s$: $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*
- $e = renameSubtree(p^e)$ *and* $f = renameSubtree(p^f)$*:*
  - $p^e = p^f$.
  - $e_1 = f_1 = renameLabel(p^e, l)$ *for some appropriate* $l \in \mathbb{E}$.
  - $\mathcal{P}^e_2 \equiv \mathcal{P}^f_2$.
  - $\forall k \in \mathbb{N},\ 3 \le k \le s$: $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*
- $e = pushSiblings(p^e_1, p^e_2)$ *and* $f = pushSiblings(p^f_1, p^f_2)$*:*
  - $p^e_1 = p^f_1$ *and* $p^e_2 = p^f_2$.
  - $e_1 = f_1 = addNode(p^e_1, p^e_2, l)$ *for some appropriate* $l \in \mathbb{E}$.
  - $\mathcal{P}^e_2 \equiv \mathcal{P}^f_2$.
  - $\forall k \in \mathbb{N},\ 3 \le k \le s$: $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*
- $e = pullSiblings(p^e)$ *and* $f = pullSiblings(p^f)$*:*
  - $p^e = p^f$.
  - $\mathcal{P}^e_1 \equiv \mathcal{P}^f_1$.
  - *Assuming* $\mathcal{P}^e_2 = (e_j)$ *and* $\mathcal{P}^f_2 = (f_j)$ *for some* $j \in \mathbb{N}$*:* $e_j = f_j = removeNode(p^e)$.
  - $\forall k \in \mathbb{N},\ 3 \le k \le s$: $\mathcal{P}^e_k \equiv \mathcal{P}^f_k$ *inductively.*

Our correction algorithm attempts to produce only mutually not equivalent update sequences. Internally we in fact do not store completely unrolled sequences of particular edit operations during the processing of provided data tree, but we use compact data structures called repairs, which are not unrolled until the phase of presenting found corrections.

Generated update sequences also do not contain redundancies, i.e. useless operations which have no effect due to some another subsequent edit operations. For example at a given position in a data tree we can insert a completely new subtree and then immediately remove it. It is probable, that the cost function would prevent these strange correction intents, but nonetheless our definition of update operations do not allow these redundancies, because we need always to process the tree from left to right. Once a new subtree is inserted, there is no chance to remove it later on.

Data Tree Distance

Having completely introduced the model of update operations, we can define the way, how distances between data trees and a data tree to tree language are measured.

Definition 4.15 (Data Tree Distances)**.** *Assume that $\mathcal{T}_1$ and $\mathcal{T}_2$ are two data trees and 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 $dist(\mathcal{T}_1, \mathcal{T}_2) = \min_{\mathcal{S} \in S} cost(\mathcal{S})$.*

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 general idea is obviously to find those repairs of an invalid data tree that are as close as possible to the original tree. However, the proposed correction algorithm is not able to produce all potentially existing repairs and then choose the cheapest ones using the cost function, since for example the subtree insertion connected with a recursive grammar or grammar with Kleene star can lead to unlimited number of locally valid subtrees. This case can be solved by the appropriately configured cost function, but the problem with *pushSiblings* is more complicated.

As a consequence, the correction algorithm is generally not able to always find repairs conforming to the distance of a given tree and provided grammar.

4.4 Correction Intents

Our correction algorithm is based on the top down processing of a provided data tree. We start at the root node and using the general concept of divide et impera we walk through the data tree towards individual leaf nodes. This is the basic difference to the approach presented in [14], where only local tree grammars are considered, but we also have another way how to generate repairs.

Although the processing is motivated by the top down traversal, we actually only invoke the correction routine at a given level of tree and then we need to wait until repairs at descendant levels are evaluated. Hence, suppose that we are in some node of a data tree. We first need to correct this node (e.g. by changing its label), then we optionally need to correct its attributes (e.g. add missing compulsory or remove not allowed ones) and then recursively process all its child nodes.

Suppose that we have computed repairs for this sequence of child nodes, then we combine these repairs with attribute repairing instructions and, finally and optionally, with an instruction manipulating a given node itself. Then we can encapsulate this repair and return it to the ascendant level of the recursion.

These repairs cannot be expressed directly using introduced edit or update operations, since these operations generally require specifying a particular position in a tree. Our algorithm however attempts to generate at once all possible

closest repairs, and thus it is impossible to know exact positions that would be valid for all found repairs.

In order to solve this problem, we will introduce repairing instructions. They are connected to introduced edit operations, but do not contain explicitly specified positions. Having one particular sequence of repairing instructions, we can derive these positions implicitly. Therefore we are able to translate such sequence into a standard sequence of edit operations that can be offered to the user or immediately used to modify the source tree.

As already outlined, the correction algorithm starts the processing at the root node and then recursively invokes the correction of descendant levels. Being at a particular level, we can generally choose from more possible actions looking forward to find the right repair. These actions are called correction intents and can be compared with update operations.

The correction approach in [14] tries to dynamically generate repairs for a sequence of sibling nodes one by one, recursively invoking the processing of their subtrees. In contrast we generate at each level a correction multigraph, which is able to statically describe all possible repairs. Then we only need to find the shortest paths in this multigraph according to the given cost function.

4.4.1 Grammar Context

The purpose of the grammar context notion is to provide an encapsulation of conditions from a single type tree grammar, that should be respected by nodes in a data tree.

Local Context of Tree Grammar

Having a single type tree grammar, we are assured that starting nonterminal symbols do not compete with each other and also that nonterminal symbols in a regular expression of each production rule do not compete with each other. As a consequence, if we are in a given node of a data tree and we have already decided, that this node and its child nodes should conform to one particular production rule, we are able to uniquely assign a nonterminal symbol to each child node, i.e. to its label viewed as a terminal symbol. Therefore we are able to choose right one production rule that should be used to restrict the given child node content. The only problem is that there can be a node with label, which is not permitted in a given context. Then we need to use special nonterminal symbol and later on we will be forced to correct such node.

Definition 4.16 (Grammar Context)**.** *Let $\mathcal{G} = (N, T, A, S, P)$ be a single type tree grammar and $\mathcal{R} = [a_R, C_R, O_R, r_R \to n_R]$ be a production rule from the set P. We define $\mathcal{C}_\mathcal{R}$ to be a* general context *of the grammar $\mathcal{G}$ for the rule $\mathcal{R}$ as a tuple $\mathcal{C}_\mathcal{R} = (a_R, n_R, N_c, P_c, map, r, C, O)$ such that:*

- *a_R is equal to $\perp$ or to a terminal symbol from $\mathcal{R}$.*
- *n_R is equal to $\perp$ or to a nonterminal symbol from $\mathcal{R}$.*

- $N_c \subseteq N$ *is a set of* allowed *nonterminal symbols, where* $N_c = \{n \mid n \in symbols(r_R)\}$.

- $P_c \subseteq P$ *is a set of* active *production rules, where* $P_c = \{\mathcal{U} \mid \mathcal{U} = [a_U, C_U, O_U, r_U \to n_U]$ *and* $n_U \in N_c\}$.

- map *is a partial function* $T \to N_c \cup \{\perp\}$ *such that* $\forall\, \mathcal{U} \in P_c$, $\mathcal{U} = [a_U, C_U, O_U, r_U \to n_U]$ *we define* $map(a_U) = n_U$ *and for all other* $a_X \in T$ *we define* $map(a_X) = \perp$ *supposing that* $\perp \notin N$ *is a special symbol.*

- $r = r_R$, $C = C_R$ *and* $O = O_R$.

Next, we define a starting context *to be* $\mathcal{C}_\bullet = (\perp, \perp, N_c, P_c, map, r_\bullet, \emptyset, \emptyset)$, *where* $N_c = S$ *and both* P_c *and map are defined in the same way as for a general context. Expression* $r_\bullet$ *is a* starting *regular expression and is equal to* $(n_1|\ldots|n_s)$, *where* $s = |S|$, $\forall i \in \mathbb{N}$, $1 \leq i \leq s$, $n_i \in S$ *and* $\forall i, j \in \mathbb{N}$, $1 \leq i < j \leq s$, $n_i \neq n_j$.

Finally, we define an empty context *to be* $\mathcal{C}_\varnothing = (\perp, \perp, \emptyset, \emptyset, map, r_\varnothing, \emptyset, \emptyset)$, *where* $r_\varnothing = \varnothing$ *and map is defined in the same way as for a general context, i.e.* $\forall a_X \in T$ *we define* $map(a_X) = \perp$.

The purpose of the grammar context is to encapsulate conditions, which should be used in order to validate a given node of a data tree. The context contains a set of allowed nonterminal symbols, mapping function from node labels to these nonterminals and, finally, a subset of production rules to be used for child nodes.

Due to the last restriction in Definition 2.14, there cannot be two production rules in a given grammar with the same terminal and concurrently nonterminal symbol. As a consequence each grammar context is uniquely determined by the pair of a terminal a_R and nonterminal symbol n_R. Using this pair we can uniquely determine one particular production rule $\mathcal{R}$ in a grammar and thus to derive its context $\mathcal{C}_\mathcal{R}$.

For technical reasons we have defined two special grammar contexts. The starting context describes the context for a root node of a data tree and the corresponding regular expression represents the alternation of all allowed starting nonterminals. The empty context is used e.g. in situations, when we need to delete whole subtrees, that can generally contain unknown labels, thus terminal symbols for which we cannot known the mapping to corresponding nonterminals and production rules.

Node Sequence Imprint

Having defined the function for mapping terminal labels to nonterminal symbols of the grammar, we can use it for sequences of nodes.

Definition 4.17 (Imprint of a Sequence of Nodes)**.** *Let* $\mathcal{T} = (D, lab, val, att)$ *be a data tree and* $u = \langle u_1, \ldots, u_n\rangle$ *a sequence of nodes for some* $n \in \mathbb{N}_0$ *such that* $\forall i \in \mathbb{N}$, $1 \leq i \leq n$, $u_i \in D$.

Given a general, starting or empty context $\mathcal{C} = (a_R, n_R, N_c, P_c, map, r, C, O)$ of a grammar $\mathcal{G}$, we define an imprint *of a sequence u of nodes in the context $\mathcal{C}$, denoted by $imprint(u)$, to be a word over an alphabet N_c such that $imprint(u) = m_1, \ldots, m_n$, where $m_i = map(lab(u_i))$.*

Given a grammar context and a sequence of nodes, usually sequence of all sibling nodes of a given parent node, the imprint of this sequence represents a word from nonterminal symbols mapped from element labels.

Finally, we will shortly describe the meaning of $\perp$ symbol. It is a special nonterminal symbol and it is used in situations, when we are not able to map a given node label to nonterminal symbol, since this label is not allowed by the schema itself, or is only not allowed in a given context.

4.4.2 Repairing Instructions and Repairs

Suppose that the correction algorithm at a given level of recursive processing finds the best suitable repairs. It has been explained, that the goal of our algorithm is to find all these repairs, but the problem is, that applying each individual found repair, we can obtain a subtree with a different fan-out. Thus it is impossible to elegantly describe explicit positions for all operations in found repairs. And we even do not need them.

On that account we use repairing instructions, which always assume the processing of a given sequence of nodes from left to right, and, therefore, can be later on easily translated into the already introduced set of edit or update operations respectively.

Repairing Instructions

Having *addAttribute*, *removeAttribute* and *renameAttribute* edit operations for attributes and $addLeaf$, $removeLeaf$, *renameLabel*, $addNode$ and *remove-Node* edit operations for nodes, we can introduce the corresponding set of repairing instructions. Their meaning is equivalent to that defined in the context of edit operations.

Definition 4.18 (Repairing Instructions)**.** *We define the following* attribute repairing instructions*, for some appropriate a, a_0 and $a_1 \in \mathbb{A}$:*

- $(\texttt{addAttribute}, a)$.
- $(\texttt{removeAttribute}, a)$.
- $(\texttt{renameAttribute}, a_0, a_1)$.

Next, we define the following node repairing instructions*, for some appropriate $n \in \mathbb{E}$ and $i \in \mathbb{N}$:*

- $(\texttt{addLeaf}, n)$.
- $(\texttt{removeLeaf})$.

- $(\texttt{renameLabel}, n)$.
- $(\texttt{addNode}, n, i)$.
- $(\texttt{removeNode})$.

For each repairing instruction we define cost *to be a function returning the same value as the cost function of corresponding edit operation.*

Even though we did not mention it explicitly, we need to be able to translate these instructions into valid edit operations, therefore the correction algorithm must consider prerequisites defined for edit operations in order to produce only suitable repairs.

Definition 4.19 (Sequences of Instructions)**.** *Having a sequence of repairing instructions $s = \langle s_1, \ldots, s_n \rangle$ for some $n \in \mathbb{N}_0$, we define* cost *of a sequence of repairing instructions to be a function $cost(s) = \sum_{k=1}^{n} cost(s_k)$.*

Two sequences of attribute repairing instructions are equivalent, if we can reorder their items and obtain the identical sequences. This notion follows the idea of equivalence of edit sequences.

Definition 4.20 (Equivalence of Attribute Instruction Sequences)**.** *Let $s = \langle s_1, \ldots, s_n \rangle$ and $t = \langle t_1, \ldots, t_n \rangle$ be two sequences of attribute repairing instructions for some $n \in \mathbb{N}_0$. We say that s and t are* equivalent*, $s \equiv t$, if $\exists j_1, \ldots, j_n \in \mathbb{N}$ permutation of indices $1, \ldots, n$ such that $\forall i \in \mathbb{N}$, $1 \le i \le n$: $s_i = t_{j_i}$.*

Repairs for Attributes

When we are inserting a new node into a data tree, we need to add all compulsory attributes. Analogously when we are deleting a node, we need to remove all its attributes. In a situation we are correcting an existing node, we need to add those compulsory attributes, which are missing, and remove those attributes, that are not allowed.

Definition 4.21 (Attribute Repairs)**.** *Assume that C, $O \subset \mathbb{A}$ are sets of names of required and optional attributes respectively and that P is a set of existing attributes, i.e. pairs of the form (a, v), $a \in \mathbb{A}$, $v \in \mathbb{V}$ conforming to the specification of the att function in Definition 2.6.*

If P is provided, we derive $E = \{a \mid \exists v \in \mathbb{V}, (a, v) \in P\}$. We define the following attribute repairs *as pairs (R, A), where R, $A \subset \mathbb{A}$ and:*

- *$insertAttributes(C)$ represents $(\emptyset, C)$.*
- *$deleteAttributes(P)$ represents $(E, \emptyset)$.*
- *$repairAttributes(P, C, O)$ represents $(E \setminus (C \cup O), C \setminus E)$.*

Assuming that c_a is the cost of addAttribute, c_r of removeAttribute and c_n of renameAttribute repairing instruction, we define a cost *to be a function returning the minimal cost of all sequences of attribute repairing instructions capable to correct attributes of a given node:*

- *If $c_n < c_a + c_r$, then cost $= max(0, |R| - |A|) * c_r + max(0, |A| - |R|) * c_a + min(|R|, |A|) * c_n$.*
- *If $c_n > c_a + c_r$, then cost $= |R| * c_r + |A| * c_a$.*
- *If $c_n = c_a + c_r$, then we can define cost using any of two previous formulae, since the returned values will always be the same.*

Repair containers are generally data structures that are capable to store all corrections, thus sequences of repairing instructions, subsequently translated to sequences of edit operations. This however does not mean, that we need to store all these sequences explicitly, we only need to store the information, from which we can later on produce such sequences.

In case of repairs for attributes we can therefore store only two sets of attributes. This is useful, because especially *repairAttributes* can produce rather large number of possible sequences.

Repairs for Nodes

A repair for a node represents a simple data structure, that only stores one optional repairing instruction meeting the criteria of corresponding correction intent.

Definition 4.22 (Node Repairs)**.** *If relevant, assume that $n \in \mathbb{E}$ and $i \in \mathbb{N}$. We define the following* node repairs *as sequences of repairing instructions:*

- *$insertSubtree(n)$ represents $\langle(\texttt{addLeaf}, n)\rangle$.*
- *$deleteSubtree()$ represents $\langle(\texttt{removeLeaf})\rangle$.*
- *$repairSubtree()$ represents $\langle\rangle$.*
- *$renameSubtree(n)$ represents $\langle(\texttt{renameLabel}, n)\rangle$.*
- *$pushNodes(n, i)$ represents $\langle(\texttt{addNode}, n, i)\rangle$.*
- *$pullNodes()$ represents $\langle(\texttt{removeNode})\rangle$.*

Furthermore, we define a cost *to be a function returning the overall cost of a node repair, i.e. for a node repair $R = \langle r_1, \ldots, r_n\rangle$ we define $cost(R) = \sum_{k=1}^{n} cost(r_k)$.*

Repairs for Trees

We have already introduced repairs for attributes and nodes, now we are first going to discuss repairs for sequences of nodes and after it repairs for correction intents. The only problem is that we currently do not have defined all required notions. Thus the following definitions will be tentative and will be elaborated later on with the knowledge of formally introduced correction intents.

A repair of a sequence of nodes stores the information describing all potential sequences of repairing instructions capable to correct a given sequence of nodes into a locally valid data tree.

Definition 4.23 (Node Sequence Repair)**.** *We tentatively define a* node sequence repair *to be a structure, that is capable to store repairs for a given sequence of nodes. It will be shown that this structure corresponds to a* repairing multigraph.

Each intent repair corresponds to particular correction intent and encapsulates sequences of repairing instructions correcting the inspected node and its entire subtree.

Definition 4.24 (Correction Intent Repair)**.** *We tentatively define a* repair *for a given correction intent to be a structure containing a node repair, an attribute repair and optionally a node sequence repair.*

Depending on the type of the associated intent, we define a cost *to be a function returning the overall cost of all involved repairs.*

The correction algorithm starts at the root node and proceeds subtrees of its child nodes recursively. When we move one level lower towards leaf nodes, we follow the idea of selected correction intents. When we are backtracking, we gather proposed corrections and in a form of a repair we pass it back towards the root node.

4.4.3 Correction of Data Trees

Being at a particular level of recursion, beside other we are given a sequence of nodes to process. Usually this sequence corresponds to all child nodes of a given parent node. But in case of the pull update operation, we can fetch all grandchild nodes and pull them at the level of their former parent.

Generally we can say that we have to process a given sequence with some other restrictions in order to find the best suitable repairs, thus sequences of repairing instructions capable to achieve a valid tree. We do not generate these sequences directly, but we choose a suitable action, which can be performed with a given node or a sequence of nodes, and then recursively repair its subtree or subtrees respectively.

These possible actions are called correction intents and their model is formally introduced in the following definition. Except the `correct`, they all correspond to already introduced update operations. Even their purpose is the

same, formally there are a few differences. First of all, correction intents are more assignment to the recursive subproblem rather than the particular specification of a sequence of operations. The subproblem is described, then solved and only then we are able to provide via repair structures particular sequences of operations. And in fact not operations, but repairing instructions, which first need to be translated, as already sketched.

Model of Correction Intents

First, we will provide the general description of correction intents. They can be seen from two levels of adjacent recursion levels differently in a way that each intent is represented by a tuple from which the first few parameters describe the situation at a level that decided to create that intent and the remaining parameters describe the subproblem to be solved at the nested level of recursion.

Definition 4.25 (Correction Intents)**.** *Let us consider all the following assumptions:*

- $\mathcal{T} = (D,\ lab,\ val,\ att)$ *is a data tree.*
- $u = \langle u_1,\ \ldots,\ u_n \rangle$ *is a sequence of nodes for some* $n \in \mathbb{N}_0$ *such that* $\forall i \in \mathbb{N},\ 1 \leq i \leq n,\ u_i \in D$.
- $\mathcal{G} = (N,\ T,\ A,\ S,\ P)$ *is a single type tree grammar.*
- $\mathcal{C} = (a_R,\ n_R,\ N_c,\ P_c,\ map,\ r,\ C,\ O)$ *is a general, starting or empty context of the grammar* $\mathcal{G}$.
- $\mathcal{A}_r = (Q,\ N_c,\ \delta,\ q_0,\ F)$ *is the Glushkov automaton for* r.

Next, assume that $Y = \{$`correct`, `insert`, `delete`, `repair`, `rename`, `push`, `pull`$\}$ *is a set of correction intent* types*. The first one is a* starting *intent, all remaining are* recursive *intents.*

We define a correction intent *to be any tuple* $\mathcal{I} = (y,\ R_N,\ R_A,\ s_I,\ q_I,\ s_E,\ Q_E,\ u',\ d',\ \mathcal{C}',\ f'_S,\ f'_T,\ q'_S,\ Q'_T,\ Y'_a)$ *which generally fulfils all the following conditions:*

- *Items describing the correction intent:*
  - $y \in Y$ *is a type of the correction intent* $\mathcal{I}$.
  - R_N *is a node repair.*
  - R_A *is an attribute repair.*
  - $s_I \in \mathbb{N}_0,\ s_I \leq n$ *is an* initial stratum.
  - $q_I \in Q$ *is an* initial state *in* s_I.
  - $s_E \in \mathbb{N}_0,\ s_E \leq n$ *is an* ending stratum.
  - $Q_E \subseteq Q$ *is a set of* ending states *in* s_E.

Only if y = `correct`, *items* R_N, R_A, s_I, q_I, s_E *and* Q_E *may not be defined.*

- *Items describing the recursive processing:*
  - $u' = \langle u'_1, \ldots, u'_{n'} \rangle$ *is a sequence of nodes to be processed for some* $n' \in \mathbb{N}_0$ *such that* $\forall i \in \mathbb{N}$, $1 \le i \le n'$, $u_i \in D$.
  - $d' \in \mathbb{N}$ *is a reached depth in a new data tree.*
  - $\mathcal{C}' = (a'_R, n'_R, N'_c, P'_c, map', r', C', O')$ *is a general, starting or empty context of the grammar* $\mathcal{G}$. *Then let* $\mathcal{A}_{r'} = (Q', N'_c, \delta', q'_0, F')$ *be the Glushkov automaton for* r'.
  - $f'_S \in \{$`expanded`, `collapsed`$\}$ *indicates how to start.*
  - $f'_T \in \{$`aggregated`, `separated`$\}$ *indicates how to terminate.*
  - $q'_S \in Q'$ *is a* source *state.*
  - $Q'_T \subseteq Q'$ *is a set of* target *states.*
  - $Y'_a \subset Y$ *is a set of allowed nested correction intent types.*

  If $r' = \varnothing$, *then we require that* $Q'_T = \{q'_0\}$. *If* f'_T = `separated`, *then* $Q_E = Q'_T$, *and if* f'_T = `aggregated`, *then* $|Q_E| = 1$.

Particular restrictions for all introduced correction intent types are the subject of two following definitions.

Correction intents in fact define the behaviour and possibilities of the correction algorithm itself. The tuple of the particular intent can be treated as the assignment for the nested problem and the solution from this subproblem is returned upwards in a form of an intent repair structure. The correction algorithm processes the given sequence of nodes from left to right and attempts to create new nested correction intents in the way the definition permits. Returned repair structure should represent those sequences of repairing instructions, that can be translated into sequences of edit operations corresponding to associated update operations.

Starting Correction Intent

Before we can define the set of recursive correction intents, we need to introduce the special intent, called *starting correction intent.* This intent describes the initiation of data trees correction.

Definition 4.26 (Starting Correction Intent)**.** *Let* $\mathcal{T} = (D, lab, val, att)$ *be a data tree and* $\mathcal{G} = (N, T, A, S, P)$ *a single type tree grammar.*

Assume that $\mathcal{C}_\bullet$ *is the starting context of the grammar* $\mathcal{G}$ *and that* $\mathcal{A}_r = (Q, N_c, \delta, q_0, F)$ *is the Glushkov automaton for* r.

We define $\mathcal{I}_\bullet$ = (`correct`, R_N, R_A, s_I, q_I, s_E, Q_E, u', d', $\mathcal{C}'$, f'_S, f'_T, q'_S, Q'_T, Y'_a) *to be a* starting *correction intent, where:*

- R_N, R_A, s_I, q_I, s_E *and* Q_E *are not defined.*
- *If* D *is not empty, then* $u' = \langle \epsilon \rangle$, *else* $u' = \langle \rangle$.
- $d' = 1$.
- $\mathcal{C}' = \mathcal{C}_\bullet = (a'_R, n'_R, N'_c, P'_c, map', r', \emptyset, \emptyset)$ *for introduced* a'_R, n'_R, N'_c, P'_c, map' *and* r'. *Then let* $\mathcal{A}_{r'} = (Q', N'_c, \delta', q'_0, F')$ *be the Glushkov automaton for* r'.
- $f_S = \texttt{expanded}$.
- $f_T = \texttt{aggregated}$.
- $q_S = q'_0$.
- $Q_T = F'$.
- $Y_a = Y \setminus \{\texttt{correct}\}$.

Since this correction intent starts the entire processing of a data tree, first six items in the tuple are not relevant, because there is no other recursion level before this one.

Recursive Correction Intents

Although the definition of the Glushkov automaton supposes the nondeterministic finite automaton, we require 1-unambiguous regular expressions for content models of elements and thus the Glushkov automaton is deterministic. We will harness this fact in the following definition of the allowed recursive correction intents.

Definition 4.27 (Recursive Correction Intents)**.** *Let* $\mathcal{T} = (D, lab, val, att)$ *be a data tree and* $\mathcal{G} = (N, T, A, S, P)$ *a single type tree grammar.*

Assume that $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$ *is an already defined correction intent, where* $u = \langle u_1, \ldots, u_n \rangle$ *is a sequence of nodes of a tree* D *for some* $n \in \mathbb{N}_0$, $\mathcal{C} = (a_R, n_R, N_c, P_c, map, r, C, O)$ *is a general, starting or empty context and* $\mathcal{A}_r = (Q, N_c, \delta, q_0, F)$ *is the Glushkov automaton for* r.

Furthermore, let $m = imprint(u) = m_1, \ldots, m_n$ *be the imprint of the sequence* u *in the context* $\mathcal{C}$.

Under the specified constraints we recursively define the following correction intents:

- $\texttt{insert}$. $\forall k \in \mathbb{N}_0$, $0 \leq k \leq n$ *in case of* $f_S = \texttt{expanded}$ *or* $1 \leq k \leq n$ *in case of* $f_S = \texttt{collapsed}$, $\forall q_s \in Q$, $\forall x \in N_c$ *such that* $\delta(q_s, x)$ *is defined,* $q_t = \delta(q_s, x)$, *we define a correction intent* $\mathcal{I}' = (\texttt{insert}, R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$, *where:*
  - $s'_I = k$, $q'_I = q_s$, $s'_E = k$ *and* $Q'_E = \{q_t\}$.

- $u' = \langle\rangle$.
- $d' = d + 1$.
- $\mathcal{C}' = \mathcal{C}_{\mathcal{R}} = (a,\ x,\ N_c',\ P_c',\ map',\ r',\ C',\ O')$ *for a production rule* $\mathcal{R} \in P_c$ *such that* $\mathcal{R} = [a,\ C',\ O',\ r' \to x]$ *for* $map(a) = x$*, some appropriate* C', O', r' *and derived* N_c', P_c', map'*. There is always right one such* $\mathcal{R}$*. Then let* $\mathcal{A}_{r'} = (Q',\ N_c',\ \delta',\ q_0',\ F')$ *be the Glushkov automaton for* r'.
- $f_S' =$ `expanded`.
- $f_T' =$ `aggregated`.
- $q_S' = q_0'$.
- *If* $r' \neq \varnothing$*, then* $Q_T' = F'$*, else* $Q_T' = \{q_0'\}$.
- $Y_a' = \{$`insert`$\}$.
- $R_N = insertSubtree(a)$.
- $R_A = insertAttributes(C')$.

Suppose that $\mathcal{I}^1,\ \ldots,\ \mathcal{I}^j$ *is the maximal chain of correction intents for some* $j \in \mathbb{N}_0$*, such that* $\forall i \in \mathbb{N},\ 1 \le i < j,\ y^i =$ `insert`*,* $\mathcal{I}^i$ *invokes* $\mathcal{I}^{i+1}$ *and, finally,* $\mathcal{I}^j = \mathcal{I}$*,* $y^j =$ `insert`*. We do not allow the previously described correction intent* $\mathcal{I}'$*, if* $\exists i,\ 1 \le i \le j$*:* $a_R^i = a$ *and* $n_R^i = x$*, where* a_R^i *and* n_R^i *are terminal and nonterminal symbols from context* $\mathcal{C}^i$ *respectively. In other words we do not allow repeated nesting of insertion correction intents with the same context.*

- `delete`*.* $\forall k \in \mathbb{N}_0,\ 0 \le k < n$ *in case of* $f_S =$ `expanded` *or* $1 \le k < n$ *in case of* $f_S =$ `collapsed`*,* $\forall q_s \in Q$ *and concurrently in addition for* $q_s = q_I$ *and* $k = 0$ *in case that* $f_s =$ `collapsed`*, we define a correction intent* $\mathcal{I}' = ($`delete`$,\ R_N',\ R_A',\ s_I',\ q_I',\ s_E',\ Q_E',\ u',\ d',\ \mathcal{C}',\ f_S',\ f_T',\ q_S',\ Q_T',\ Y_a')$*, where:*
  - $s_I' = k$, $q_I' = q_s$, $s_E' = k + 1$ *and* $Q_E' = \{q_s\}$.
  - $u' = \langle u_{k+1}.0,\ \ldots,\ u_{k+1}.(fanOut(u_{k+1}) - 1)\rangle$.
  - $d' = d + 1$.
  - $\mathcal{C}' = \mathcal{C}_{\varnothing} = (\bot,\ \bot,\ \emptyset,\ \emptyset,\ map',\ r_{\varnothing}',\ \emptyset,\ \emptyset)$ *for introduced* map' *and* $r_{\varnothing}'$*. Then let* $\mathcal{A}_{r'} = (Q',\ N_c',\ \delta',\ q_0',\ F')$ *be the Glushkov automaton for* r'.
  - $f_S' =$ `expanded`.
  - $f_T' =$ `aggregated`.
  - $q_S' = q_0'$.
  - $Q_T' = \{q_o'\}$.
  - $Y_a' = \{$`delete`$\}$.
  - $R_N = deleteSubtree()$.

– $R_A = deleteAttributes(att(u_{k+1}))$.

- `repair`. $\forall k \in \mathbb{N}_0$, $0 \leq k < n$ *in case of* $f_S =$ `expanded` *or* $1 \leq k < n$ *in case of* $f_S =$ `collapsed`, $\forall q_s \in Q$ *and concurrently in addition for* $q_s = q_I$ *and* $k = 0$ *in case that* $f_s =$ `collapsed`; *if* $m_{k+1} \neq \perp$, $\delta(q_s, m_{k+1})$ *is defined and* $q_t = \delta(q_s, m_{k+1})$, *we define a correction intent* $\mathcal{I}' = ($`repair`, R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', $\mathcal{C}'$, f'_S, f'_T, q'_S, Q'_T, $Y'_a)$, *where:*
  – $s'_I = k$, $q'_I = q_s$, $s'_E = k + 1$ *and* $Q'_E = \{q_t\}$.
  – $u' = \langle u_{k+1}.0, \ldots, u_{k+1}.(fanOut(u_{k+1}) - 1)\rangle$.
  – $d' = d + 1$.
  – $\mathcal{C}' = \mathcal{C}_\mathcal{R} = (lab(u_{k+1}), m_{k+1}, N'_c, P'_c, map', r', C', O')$ *for a production rule* $\mathcal{R} \in P_c$ *such that* $\mathcal{R} = [lab(u_{k+1}), C', O', r' \to m_{k+1}]$ *for some appropriate* C', O', r' *and derived* N'_c, P'_c, map'. *There is always right one such* $\mathcal{R}$. *Then let* $\mathcal{A}_{r'} = (Q', N'_c, \delta', q'_0, F')$ *be the Glushkov automaton for* r'.
  – $f'_S =$ `expanded`.
  – $f'_T =$ `aggregated`.
  – $q'_S = q'_0$.
  – *If* $r' \neq \varnothing$, *then* $Q'_T = F'$, *else* $Q'_T = \{q'_0\}$.
  – *If* $r' \neq \varnothing$, *then* $Y'_a = Y \setminus \{$`correct`$\}$, *else* $Y'_a = \{$`delete`$\}$.
  – $R_N = repairSubtree()$.
  – $R_A = repairAttributes(att(u_{k+1}), C', O')$.

- `rename`. $\forall k \in \mathbb{N}_0$, $0 \leq k < n$ *in case of* $f_S =$ `expanded` *or* $1 \leq k < n$ *in case of* $f_S =$ `collapsed`, $\forall q_s \in Q$ *and concurrently in addition for* $q_s = q_I$ *and* $k = 0$ *in case that* $f_s =$ `collapsed`; *if* $m_{k+1} = \perp$ *or* $\delta(q_s, m_{k+1})$ *is not defined, then* $\forall x \in N_c$ *such that* $\delta(q_s, x)$ *is defined,* $q_t = \delta(q_s, x)$, *we define a correction intent* $\mathcal{I}' = ($`rename`, R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', $\mathcal{C}'$, f'_S, f'_T, q'_S, Q'_T, $Y'_a)$, *where:*
  – $s'_I = k$, $q'_I = q_s$, $s'_E = k + 1$ *and* $Q'_E = \{q_t\}$.
  – $u' = \langle u_{k+1}.0, \ldots, u_{k+1}.(fanOut(u_{k+1}) - 1)\rangle$.
  – $d' = d + 1$.
  – $\mathcal{C}' = \mathcal{C}_\mathcal{R} = (a, x, N'_c, P'_c, map', r', C', O')$ *for a production rule* $\mathcal{R} \in P_c$ *such that* $\mathcal{R} = [a, C', O', r' \to x]$ *for* $map(a) = x$, *some appropriate* C', O', r' *and derived* N'_c, P'_c, map'. *There is always right one such* $\mathcal{R}$. *Then let* $\mathcal{A}_{r'} = (Q', N'_c, \delta', q'_0, F')$ *be the Glushkov automaton for* r'.
  – $f'_S =$ `expanded`.
  – $f'_T =$ `aggregated`.
  – $q'_S = q'_0$.

- *If* $r' \neq \varnothing$, *then* $Q'_T = F'$, *else* $Q'_T = \{q'_0\}$.
- *If* $r' \neq \varnothing$, *then* $Y'_a = Y \setminus \{\texttt{correct}\}$, *else* $Y'_a = \{\texttt{delete}\}$.
- $R_N = renameSubtree(a)$.
- $R_A = repairAttributes(att(u_{k+1}), C', O')$.

- `push`. $\forall k \in \mathbb{N}_0$, $0 \leq k < n$ *in case of* $f_S = \texttt{expanded}$ *or* $1 \leq k < n$ *in case of* $f_S = \texttt{collapsed}$, $\forall q_s \in Q$ *and concurrently in addition for* $q_s = q_I$ *and* $k = 0$ *in case that* $f_s = \texttt{collapsed}$; $\forall j \in \mathbb{N}$, $k < j < n$, $\forall x \in N_c$ *such that* $\delta(q_s, x)$ *is defined,* $q_t = \delta(q_s, x)$ *we define a correction intent* $\mathcal{I}'$ $= (\texttt{push}, R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$, *where:*

  - $s'_I = k$, $q'_I = q_s$, $s'_E = j + 1$ *and* $Q'_E = \{q_t\}$.
  - $u' = \langle u_{k+1}, \dots, u_{j+1} \rangle$.
  - $d' = d + 1$.
  - $\mathcal{C}' = \mathcal{C}_\mathcal{R} = (a, x, N'_c, P'_c, map', r', C', O')$ *for a production rule* $\mathcal{R} \in P_c$ *such that* $\mathcal{R} = [a, C', O', r' \to x]$ *for* $map(a) = x$, *some appropriate* C', O', r' *and derived* N'_c, P'_c, map'. *There is always right one such* $\mathcal{R}$. *Then let* $\mathcal{A}_{r'} = (Q', N'_c, \delta', q'_0, F')$ *be the Glushkov automaton for* r'. *We require that* $r' \neq \varnothing$.
  - $f'_S = \texttt{expanded}$.
  - $f'_T = \texttt{aggregated}$.
  - $q'_S = q'_0$.
  - $Q'_T = F'$.
  - $Y'_a = Y \setminus \{\texttt{correct}\}$.
  - $R_N = pushNodes(a, (j + 1) - (k + 1) + 1)$.
  - $R_A = insertAttributes(C')$.

  Suppose that $\mathcal{I}^1, \dots, \mathcal{I}^j$ *is the maximal chain of correction intents for some* $j \in \mathbb{N}_0$, *such that* $\forall i \in \mathbb{N}$, $1 \leq i < j$, $y^i = \texttt{push}$, $\mathcal{I}^i$ *invokes* $\mathcal{I}^{i+1}$, $n^i = n'$ *and, finally,* $\mathcal{I}^j = \mathcal{I}$, $n = n'$, $y^j = \texttt{push}$. *We do not allow the previously described correction intent* $\mathcal{I}'$, *if* $\exists i$, $1 \leq i \leq j$: $a^i_R = a$ *and* $n^i_R = x$, *where* a^i_R *and* n^i_R *are terminal and nonterminal symbols from context* $\mathcal{C}^i$ *respectively. In other words we do not allow repeated nesting of push correction intents with the same context and same sequence of nodes.*

- `pull`. $\forall k \in \mathbb{N}_0$, $0 \leq k < n$ *in case of* $f_S = \texttt{expanded}$ *or* $1 \leq k < n$ *in case of* $f_S = \texttt{collapsed}$, $\forall q_s \in Q$ *and concurrently in addition for* $q_s = q_I$ *and* $k = 0$ *in case that* $f_s = \texttt{collapsed}$; *if* $fanOut(u_{k+1}) > 0$ *and* $reachable(q_s) \neq \emptyset$, *then we define a correction intent* $\mathcal{I}' = (\texttt{pull}, R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$, *where:*

  - $s'_I = k$, $q'_I = q_s$, $s'_E = k + 1$ *and* $Q'_E = reachable(q_s)$.

- $u' = \langle u_{k+1}.0, \ldots, u_{k+1}.(fanOut(u_{k+1}) - 1)\rangle$.
- $d' = d$.
- $\mathcal{C}' = \mathcal{C}$.
- $f'_S = \texttt{collapsed}$.
- $f'_T = \texttt{separated}$.
- $q'_S = q_s$.
- $Q'_T = reachable(q_s)$.
- $Y'_a = Y \setminus \{\texttt{correct}, \texttt{push}\}$.
- $R_N = pullNodes()$.
- $R_A = deleteAttributes(att(u_{k+1}))$.

Each of the previously introduced correction intent of whatever type $\mathcal{I}' = (y', R'_N, R'_A, s_I, q_I, s_E, Q_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$ *is only defined if* $y' \in Y_a$.

We declare $intents(\mathcal{I})$ *to be a set of all correction intents* $\mathcal{I}'$*, that can be derived from* $\mathcal{I}$ *using the above descriptions.*

Finally, for correction intents $\mathcal{I}'$ *having type* $y' \in \{\texttt{delete}, \texttt{repair}, \texttt{rename}, \texttt{push}, \texttt{pull}\}$ *we define* $baseNode(\mathcal{I}') = u_{k+1}$.

Basic Description of Intents

Now we will describe the semantics of items in a tuple of a given correction intent. The first parameter y represents the type of the correction intent and must be equal to one of the defined intents.

Parameters R_N and R_A stand for a node repair and an attribute repair respectively. These repairs define instructions that have to be performed at a given node in order to follow selected intent on invoked levels of recursion.

Item s_I is the number of the stratum and q_I the state of the Glushkov automaton in this stratum, where the given intent has been generated. This means that after s_I processed nodes from the input sequence of nodes and in the corresponding state of automaton, the algorithm allows to perform a new correction intent with the following node and its subtree or nodes and their subtrees.

The correction algorithm in fact can be described as a simulation of the traversal of the state space of the corresponding Glushkov automaton. For example, if we choose `repair` intent, we accept the next node of a sequence, recursively initiate correction of descendant subtrees and then appear back in s_E stratum, i.e. one stratum to the right from s_I, and in a new state of automaton $q_E \in Q_E$, since we have accepted and used the transition in the Glushkov automaton from state q_I to state q_E via given nonterminal symbol. Item s_E describes the number of the target stratum and Q_E set of states in it.

All the previous items were describing the circumstances, from which the given correction item has been generated. The remaining items describe the parameters for the processing of the next level of recursion.

The first of them u stands for the sequence of nodes to be processed. As already explained, this sequence does not need to always conform to the sequence of all sibling nodes of a given parent node. Item d is the depth in a data tree. Usually we increase this depth by 1 whenever we enter the next recursion level, but this not holds always. In a *pull* intent we pull some nodes up by one level and continue their processing in the same depth of the tree.

Item $\mathcal{C}'$ is the grammar context to be used at a nested level. It conforms to the terminal and nonterminal symbol assigned to a given parent node (from the point of view of the nested level).

If the flag f_S is equal to `collapsed`, then the stratum with a number 0 of the nested level is collapsed into only one state q_S and we do not allow *insert* correction intent in that zeroth stratum. Otherwise if it is equal to `expanded`, the zeroth stratum is defined completely. The former variant is only used by `pull`, since we can perform any required insertion in the original depth before we decide to invoke this intent.

Flag f_T influences the way of aggregating found repairs at a given level. If it is equal to `aggregated`, which is the usual variant, we are trying to find all repairs and finish at the last stratum in any state $q_T \in Q_T$. We do not need to know in which state we have really finished, because all found repairs are worth the same. Conversely, if this flag is equal to `separated`, we are interested in particular target states and we need to return repairs separately.

The last parameter of each correction item Y_a is a set of correction intent types, that are allowed in the nested level. For example if we have once decided to insert a new subtree using `insert`, we really cannot use any other way of correction, since there would be nothing to correct.

Recursion Termination

The main idea of the correction algorithm is based on the recursive processing, therefore we need to explicitly define conditions, under which we stop further nesting. It is easy to determine, that the processing will certainly be stopped in such correction intents that to not allow any nested content and concurrently that should process only empty sequence of input nodes. We will call this type of correction intent as a *terminating correction intent.*

Definition 4.28 (Terminating Correction Intent)**.** *Assume that* $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$ *is a correction intent and* $\mathcal{C} = (a_R, n_R, N_c, P_c, map, r, C, O)$ *a grammar context.*

We say that $\mathcal{I}$ *is a* terminating correction intent*, if the following conditions concurrently hold:*

- $u = \langle\rangle$ *is an empty sequence of nodes.*
- $r = r_\varnothing = \varnothing$ *is an empty regular expression.*

Each terminating intent only has one stratum with only one state. This state is a source state, but there are no target states, i.e. the corresponding Glushkov automaton accepts only an empty language.

The essential problem is caused by the mechanism of correction intents evaluation. We process the tree from top to down, but in order to complete the processing at a given level of recursive nesting, we first need to evaluate all invoked nested correction intents. If we ensure, that each nesting must finish in finite time reaching some terminating correction intent, the recursion itself will be certainly finite.

The problem is therefore hidden only in the way, how we generate nested intents. And, in fact, only `insert` and `push` correction intents are really problematic. All other intents are safe, because they always handle only existing nodes and move one level lower towards leaves. Since data trees are finite, we must reach some terminating correction intent sooner or later.

In case of the `insert` intent, there are two constructs causing potential infinity of trees to be inserted. The first cause is Kleene star operation in content models. We can avoid this repeated generation of subtrees in a way, that we prohibit repeated walking through cycles in the corresponding Glushkov automaton. The second problem with `insert` is caused by recursive tree grammars. Because we consider only consistent grammars, there must always be a way in content models, how to bypass this recursion. Therefore, we can avoid this behaviour by forbidding repeated nested invocations of such insertion intent.

The similar situation is as well around the `push` correction intent, since we can insert new internal nodes without limits, if the provided tree grammar is recursive. The proposed solution is analogous to the `insert` intent.

4.5 Correction Multigraphs

The previous chapter introduced the notion of the correction intents. Their purpose is to describe one possible step of the correction algorithm in a way of a local action with usually one or exceptionally more sibling nodes followed usually by the processing of their child nodes.

We will successively introduce three levels of multigraph abstractions, starting with an exploration multigraph for description of all possible nested correction intents, then derived correction multigraph having all nested correction intents evaluated and, finally, a repairing multigraph compactly storing all found repairs.

4.5.1 Exploration Multigraph

The correction algorithm itself can be at each level of the recursion described by the exploration multigraph. Its vertices are separated into strata, which correspond to the size of processed subsequence of the input sequence of nodes, and its edges correspond to generated intents, yet without repairs passed from nested levels.

The notion of this multigraph is the subject of this subsection. Furthermore, this multigraph not only abstracts the working of the correction algorithm, but

in order to find all suitable repairs efficiently, we really need to construct it, or at least its required subgraph.

Vertices and Edges

At first we start with the notion of vertices and edges for the exploration multigraph.

Definition 4.29 (Exploration Vertex)**.** *Assume that* $\mathcal{I} = (y,\ R_N,\ R_A,\ s_I,\ q_I,\ s_E,\ Q_E,\ u,\ d,\ \mathcal{C},\ f_S,\ f_T,\ q_S,\ Q_T,\ Y_a)$ *is a correction intent,* n *length of* u *and* Q *the set of states of the Glushkov automaton* $\mathcal{A}_r$ *for* r *from* $\mathcal{C}$*.*

An exploration vertex *is a tuple* $v = (s,\ q)$ *where:*

- $s \in \mathbb{N}_0$*,* $0 \leq s \leq n$ *is a number of a* stratum*,*
- $q \in Q$ *is a state of the Glushkov automaton* $\mathcal{A}_r$*.*

An exploration vertex is a vertex of the exploration multigraph. Each vertex is defined by a number of a stratum and a state of the Glushkov automaton, which is constructed for a regular expression describing allowed content for a processed element node.

Since we are going to define the exploration multigraph, not only an ordinary graph, we have decided to formally introduce edges as special objects. Thus each directed exploration edge has its source and target vertex appended by the reference to the particular correction intent. We have to emphasise that correction intents usually produce only one exploration edge, but `pull` intent can generally produce more edges. This means that there can be more edges with the same associated intent under the circumstance $|Q_E| > 1$.

All exploration edges only describe the assignment for the nested recursive computations, but later on we will append to these edges also repair structures passed from the nested levels. This is the reason why `pull` generally need more edges, although from the nested level only one common repair structure is returned.

Definition 4.30 (Exploration Edge)**.** *Assume that* $\mathcal{I}$ *and* $\mathcal{I}'$ *are two correction intents. An* exploration edge *is a tuple* $e = (v_1,\ v_2,\ i)$ *where:*

- v_1 *and* v_2 *are exploration vertices derived from* $\mathcal{I}$*,*
- $i = \mathcal{I}' \in intents(\mathcal{I})$*.*

Exploration Multigraph Notion

In other words the exploration multigraph describes all ways of correcting a given sequence of nodes. This structure is static and we will show that we need to find all shortest paths to some specified vertices in order to find corrections closest to the original data tree. This is one of the main differences to the approach presented in [14]. The main idea of a graph abstraction itself is adopted from [48].

Definition 4.31 (Exploration Multigraph)**.** *Given a data tree* $\mathcal{T}$ = (D, lab, val, att) *and a single type tree grammar* $\mathcal{G}$ = (N, T, A, S, P)*, assume that* $\mathcal{I}$ = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, $\mathcal{C}$, f_S, f_T, q_S, Q_T, Y_a) *is a correction intent, where* $u = \langle u_1, \ldots, u_n \rangle$ *is a sequence of nodes,* $\mathcal{C}$ = (N_c, P_c, map, r, C, O) *a grammar context and* $\mathcal{A}_r$ = (Q, N_c, δ, q_0, F) *the corresponding Glushkov automaton.*

Let $intents(\mathcal{I})$ *be a set of all correction intents* $\mathcal{I}'$*, that can be derived from* $\mathcal{I}$ *using Definition 4.27.*

We define an exploration multigraph *for* $\mathcal{I}$ *to be a directed multigraph* $E(\mathcal{I})$ = (V, E)*, where* V *is a non empty set of exploration vertices derived from* $\mathcal{I}$*,* E *is a set of exploration edges derived from* $\mathcal{I}$ *and both* V *and* E *satisfies the following conditions:*

- *If* f_S = `expanded`*, then:*
  $V = \{(k, q) \mid k \in \mathbb{N}_0,\ 0 \leq k \leq n,\ q \in Q\}$.
- *If* f_S = `collapsed`*, then:*
  $V = \{(k, q) \mid k \in \mathbb{N},\ 1 \leq k \leq n,\ q \in Q\} \cup \{(0, q_S)\}$.
- *Starting with* $E = \emptyset$*, we add for* $\forall \mathcal{I}' \in intents(\mathcal{I})$*,* $\mathcal{I}'$ = (y', s'_I, q'_I, s'_E, Q'_E, u', d', $\mathcal{C}'$, f'_S, f'_T, q'_S, Q'_T, Y'_a)*,* $\forall q'_E \in Q'_E$ *an exploration edge* $((s'_I, q'_I), (s'_E, q'_E), \mathcal{I}')$ *into* E.

A stratum s *of an exploration multigraph* (V, E) *for some* $s \in \mathbb{N}$ *is a set* V_s *of exploration vertices such that* $V_s = \{v \mid v \in V,\ v = (s, q) \text{ for some } q \in Q\}$.

Having an exploration multigraph, all its vertices can be divided into disjoint strata. Vertices in each stratum correspond to states of the Glushkov automaton, that is used to recognise sequences of nodes, which conforms to the allowed content model. Being in the stratum with a number k, exactly k nodes from the input sequence have already been processed.

Moreover all edges except that for `insert` intents start in some vertex of one stratum and always have to end in some vertex of a stratum with strictly higher number. This means that these edges follow the processing of the sequence from left to right.

Conversely, edges for `insert` intents only start and end in the same stratum. And there are even no other edges than for insertion intents inside any stratum. The reason is simple, because the purpose of this intent is to insert a new subtree, thus we insert it, but do not touch the next waiting node from the input sequence.

Edges between strata are oriented only from left to right, thus from stratum with a lower number to a stratum with a strictly higher number. In a given non-collapsed stratum, edges directly correspond to transition function of the Glushkov automaton. Thus there can generally be cycles, because Kleene star operator causes them.

4.5.2 Correction Multigraph

The naive algorithm statically constructs the whole exploration multigraph and then evaluates all its edges in order to find all repairs. It will be shown, that we do not actually need to evaluate the complete multigraph, but the notion of the correction multigraph from theoretical reasons assumes the complete evaluation.

Correction Edges

First we will extend the notion of an exploration edge to enable storing of evaluated nested intent repairs directly in the structure of correction edges.

Definition 4.32 (Correction Edge). *Given an exploration edge* $(v_1, v_2, \mathcal{I}')$ *in an exploration multigraph* $E(\mathcal{I}) = (V, E)$*, where* $v_2 = (s_2, q_2)$ *and* $\mathcal{I}' = (y', R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$*, we define a* correction edge *to be a tuple* $(v_1, v_2, \mathcal{I}', \mathcal{R}_{\mathcal{I}'}, c)$ *such that:*

- $\mathcal{R}_{\mathcal{I}'}$ *is a repair corresponding to the correction intent* $\mathcal{I}'$*.*
- *If* $f'_T =$ `aggregated`*, then* $c = cost(\mathcal{R}_{\mathcal{I}'})$*.*
- *If* $f'_T =$ `separated`*, then* $c = cost(\mathcal{R}_{\mathcal{I}'}, q_2)$*.*

Each correction edge is derived from the corresponding exploration edge by adding the repair structure returned from the nested level of recursive processing and the corresponding value of cost function. Its computation will be introduced later on.

Correction Multigraph Notion

The correction multigraph itself comes out from the exploration multigraph and only contains all its edges evaluated.

Definition 4.33 (Correction Multigraph). *Given an exploration multigraph* $E(\mathcal{I}) = (V, E)$*, we define a* correction multigraph *to be a pair* $C(\mathcal{I}) = (V, E')$*, where:*

- V *is equivalent to the original set of exploration 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.*

Paths in Correction Multigraphs

It has been already outlined, that all corrections we want to find, are encoded in the shortest paths in the correction multigraph. We will first formally introduce paths in these multigraphs as sequences of distinct edges with no repeated vertices.

Definition 4.34 (Path and Shortest Path)**.** *Let $C(\mathcal{I}) = (V, E)$ be a correction multigraph. Given v_S, $v_T \in V$, we define a* path *to be a sequence $p_{v_S,v_T} = \langle e_1, \ldots, e_n\rangle$ of correcting edges, where $n \in \mathbb{N}_0$ is a* length *of the sequence and:*

- $\forall k \in \mathbb{N}$, $1 \leq k \leq n$, $e_k = (v_1^k, v_2^k, i^k, \mathcal{R}_{i^k}^k, c^k)$.
- *If $n > 0$, then $v_1^1 = v_S$ and $v_2^n = v_T$.*
- $\forall k \in \mathbb{N}$, $1 \leq k < n$: $v_2^k = v_1^{k+1}$.
- $\neg\exists j$, $k \in \mathbb{N}$, $1 \leq j < k \leq n$: $v_1^j = v_1^k$, $v_2^j = v_2^k$ *or* $v_1^j = v_2^j$.

If $v_S = v_T$, then $p_{v_S,v_T} = \langle\rangle$ is an empty sequence. Vertices v_S, v_T are called source *and* target *respectively. By P_{v_S,v_T} we denote a set of all paths from v_S to v_T. Next, we define a* cost *of the path to be a function defined by formula $cost(p_{v_S,v_T}) = \sum_{k=1}^{n} c^k$.*

We say that p_{v_S,v_T} is the shortest path *from source v_S to target v_T, if and only if $\neg\,\exists p'_{v_S,v_T}$ such that $cost(p'_{v_S,v_T}) < cost(p_{v_S,v_T})$. By $P^{min}_{v_S,v_T}$ we denote a set of all shortest paths from v_S to v_T.*

Let $m = \min_{v_T \in V_T} cost(p_{v_S,v_T})$ for some $V_T \subseteq V$. Then we define $P^{min}_{v_S,V_T}$ to be a set of all paths from v_S to any $v_T \in V_T$ with cost equal to m, i.e. $P^{min}_{v_S,V_T} = \{p \mid \forall v_T \in V_T, \forall p \in P^{min}_{v_S,v_T}$ such that $cost(p) = m\}$.

Finally, given a path p_{v_S,v_T} we define a function vertices *returning a set of vertices on a given path, i.e. $vertices(p_{v_S,v_T}) = \{v \mid \exists k \in \mathbb{N}, 1 \leq k \leq n, v_1^k = v$ or $v_2^k = v\}$.*

Shortest paths are computed using the concept of the cost function, first introduced for edit operations and then adopted by repairing instructions. It is important to recall that we only consider paths with distinct edges and vertices. This avoids involving whole cycles of the multigraph as parts of paths. Since we have presented that the correction multigraph can only contain cycles inside a stratum in the connection with `insert` intents, we can show that if the cost of *addLeaf* edit operation is strictly greater then 0, then the shortest path would never involve whole cycles even if paths with repeated edges or vertices would be allowed. According to this finding we can justify the restriction for simple paths.

4.5.3 Repairing Multigraph

We present two modifications of the repairing multigraph. The first one is dedicated to correction intents, that should return separated corrections for each individual target state in the last stratum, and the second one aggregate all found corrections to any specified target state into only one common repair. In both cases, the repairing multigraph is a subgraph of the corresponding correction multigraph. Its purpose is to therefore prune the correction multigraph and store only the required portion of information needed to encode found corrections.

Separated Repairing Multigraph

Definition 4.35 (Separated Repairing Multigraph)**.** *Let $C(\mathcal{I}) = (V, E)$ be a correction multigraph for some correction intent $\mathcal{I} = (y,\ R_N,\ R_A,\ s_I,\ q_I,\ s_E,\ Q_E,\ u,\ d,\ \mathcal{C},\ f_S,\ f_T,\ q_S,\ Q_T,\ Y_a)$, where $f_T =$ `separated` and $u = \langle u_1,\ \ldots,\ u_n\rangle$ for some $n \in \mathbb{N}_0$.*

Assume that $v_S \in V$ is a source vertex such that $v_S = (0, q_S)$ and $V_T \subseteq V$, $V_T = \{v_T \mid v_T = (n, q_T),\ q_T \in Q_T\}$ is a set of target vertices.

We define $R^S(\mathcal{I})$ to be a separated repairing multigraph, *where $R^S(\mathcal{I}) = (V',\ E',\ pathPrev,\ pathCost,\ termCost)$ is a directed subgraph of $C(\mathcal{I})$ such that:*

- $V' = \{v \mid \forall v_T \in V_T,\ \forall p \in P^{min}_{v_S,v_T}\colon v \in vertices(p)\}$.
- $E' = \{e \mid \forall v_T \in V_T,\ \forall p \in P^{min}_{v_S,v_T},\ \forall k \in \mathbb{N},\ 1 \leq k \leq n\colon e = e_k\}$.
- pathPrev *is a partial function $V' \to \mathcal{P}(V')$ assigning to a vertex $v \in V'$ a set of vertices preceding v on any of the shortest paths from v_S to v, i.e. $pathPrev(v) = \{w \mid \exists p \in P^{min}_{v_S,v},\ p = \langle e_1,\ \ldots,\ e_n\rangle,\ e_n = (w, v, i, R_i, c)$ for some intent i, repair R_i and cost $c\}$.*
- pathCost *is a partial function $V' \to \mathbb{R}^+_0$ assigning to a vertex $v \in V'$ the cost of the shortest path from v_S to v, i.e. $pathCost(v) = cost(p)$ for any $p \in P^{min}_{v_S,v}$.*
- termCost *is a function $V_T \to \mathbb{R}^+_0$ such that $\forall v_T \in V_T\colon termCost(v_T) = pathCost(v_T)$.*

The separated repairing multigraph contains all shortest paths to each individual target vertex separately, thus there can be paths with different costs, but whenever there is a path to a selected target vertex, then it is the shortest path to this vertex. Conversely, the aggregated repairing multigraph contains all shortest paths to any of the target vertices, but all of these paths are of the same and minimal cost.

Aggregated Repairing Multigraph

Definition 4.36 (Aggregated Repairing Multigraph)**.** *Let $C(\mathcal{I}) = (V, E)$ be a correction multigraph for some correction intent $\mathcal{I} = (y,\ R_N,\ R_A,\ s_I,\ q_I,\ s_E,\ Q_E,\ u,\ d,\ \mathcal{C},\ f_S,\ f_T,\ q_S,\ Q_T,\ Y_a)$, where $f_T =$ `aggregated` and $u = \langle u_1,\ \ldots,\ u_n\rangle$ for some $n \in \mathbb{N}_0$.*

Assume that $v_S \in V$ is a source vertex such that $v_S = (0, q_S)$ and $V_T \subseteq V$, $V_T = \{v_T \mid v_T = (n, q_T),\ q_T \in Q_T\}$ is a set of target vertices.

We define $R^A(\mathcal{I})$ to be an aggregated repairing multigraph, *where $R^A(\mathcal{I}) = (V',\ E',\ pathPrev,\ pathCost,\ termCost)$ is a directed subgraph of $C(\mathcal{I})$ such that:*

- $V' = \{v \mid \forall p \in P^{min}_{v_S,V_T}\colon v \in vertices(p)\}$.

- $E' = \{e \mid \forall p \in P^{min}_{v_S,V_T},\ \forall k \in \mathbb{N},\ 1 \leq k \leq n\text{: } e = e_k\}$.
- pathPrev *and* pathPrev *are partial functions defined in the same way as corresponding functions in Definition 4.35.*
- termCost *is a constant such that termCost* $= cost(p_{min})$ *for some* $p_{min} \in P^{min}_{v_S,V_T}$.

The former repairing multigraph is used only for `pull` intent, the latter one is used for all other correction intents. This can also be explained by a fact, that in the first case we continue processing at the same depth of a data tree, even though we have entered the next level of recursion. Thus we do not want to end at accepting states in the last stratum, but we need to carry the computed information up to the previous level of recursion and then resume and continue the processing from corresponding states.

4.5.4 Repairs Construction

Finally, having formally introduced the notion of the repairing multigraph, we can complete the definition of repairs for sequences of nodes and repairs for intents, which have tentatively been introduced in the Subsection 4.4.2.

Repairs of Node Sequences

Definition 4.37 (Node Sequence Repair)**.** *A* node sequence repair *for a correction intent* $\mathcal{I}$ *is a repairing multigraph* $R^S(\mathcal{I})$ *or* $R^A(\mathcal{I})$ *depending on the type of the correction intent.*

The previous definition directly follows expectations and we only note that the node sequence repair structure can be seen only as an encapsulation for the corresponding repairing multigraph.

Repairs for Correction Intents

Now we need to precisely define the overall repair for a correction intent. The correction algorithm processes the provided data tree from top to down, attempting to consider all correction ways derivable using introduced model of correction intents. During the backtracking, we gather the found solutions, encapsulate them in a compact data structure and pass them one level up towards the original root node.

Definition 4.38 (Correction Intent Repair)**.** *Assume that* $\mathcal{I} = (y,\ R'_N,\ R'_A,\ s_I,\ q_I,\ s_E,\ Q_E,\ u,\ d,\ \mathcal{C},\ f_S,\ f_T,\ q_S,\ Q_T,\ Y_a)$ *is a correction intent, where* $u = \langle u_1, \ldots, u_n \rangle$ *for some* $n \in \mathbb{N}_0$.

We define a repair *for correction intent* $\mathcal{I}$ *to be a tuple* $\mathcal{R}_\mathcal{I} = (R_N,\ R_A,\ R_S,\ cost)$*, where:*

- $R_N = R'_N$ *is a node repair.*

- $R_A = R'_A$ *is an attribute repair.*
- *If* $\mathcal{I}$ *is a terminating correction intent:*
  - $R_S = \bot$.
  - $cost = 0$.
- *If* $\mathcal{I}$ *is not a terminating intent and* $f_T =$ `aggregated`*:*
  - $R_S = R^A(\mathcal{I}) = (V', E', pathPrev, pathCost, termCost)$ *is a node sequence repair represented by the repairing multigraph for* $\mathcal{I}$.
  - $cost = cost(R_N) + cost(R_A) + termCost$.
- *If* $\mathcal{I}$ *is not a terminating intent and* $f_T =$ `separated`*:*
  - $R_S = R^S(\mathcal{I}) = (V', E', pathPrev, pathCost, termCost)$ *is a node sequence repair represented by the repairing multigraph for* $\mathcal{I}$.
  - $\forall q_T \in Q_T$*:* $cost(q_T) = cost(R_N) + cost(R_A) + termCost(q_T)$.

Each repair for a given correction intent has essentially three main components and one supporting function for easier requesting costs of involved nested repairs.

The first component stands for a node repair, the second for an attribute repair and the third for a node sequence repair, all associated and derived from the given correction intent. The last named one represents the nested recursive processing, the first two components directly define the correction intent itself.

Since we need to encapsulate either aggregated or separated repairing multigraphs, we need to differentiate the definition of the cost function. In the first case it behaves only as an ordinary constant representing the overall minimal cost, in the second case we need to provide direct access to all targets and provide both repairs and costs separately.

The special attention is needed in case of terminating correction intents, since these intents do not have any nested level of recursion and thus component R_S remains undefined. From the definition of correction intents we know, that there cannot be any terminating intent that would be separated.

4.6 Repairs Presentation

The correction algorithm starts the processing of a provided potentially invalid data tree at its root node. We have introduced the notion of correction intents in order to describe all possible directions, the correction algorithm can choose to find requested corrections. The algorithm recursively proceeds the tree and when the bottom of the recursion is reached, we backtrack and construct compact repair structures holding all found corrections.

The first problem of these repair structures is the already outlined fact, that we cannot directly use edit or update operations in them, because these

operations need specifying of explicit positions, i.e addresses of nodes in the underlying tree. Since we store different corrections together, there is no simple way how to determine the required positions already during the recursive processing.

Thus we use repairing instructions, which only define actions to be performed, but the particular position is omitted. Since we restricted processing of each sequence of nodes only in a left to right manner, we can retrospectively translate found repairs and obtain standard sequences of edit operations.

The purpose of this section is to first describe the mechanisms, how we unroll compact repairs into individual sequences of repairing instructions, and then how we perform the mentioned translation to update sequences.

The outlined problem also relates to a way, how we plan to present found corrections. The algorithm proposed in this thesis assumes that there is no interaction with the user and thus we only need to find the best corrections and directly apply them in order to gain a valid data tree. On that account we will describe the mechanism for creating monolithic edit sequences for root nodes, capable of correcting the entire original data tree.

On the other hand, we can also traverse the given data tree to the depth and find isolated locations, where corrections to errors are proposed. Then we can independently offer these local corrections to the user.

4.6.1 Repairs for Nodes and Attributes

Node repairs are very simple and since they contain only trivial sequences of repairing instructions, their evaluation and translation is easy. Attribute repairs are simple data structures too, however we first need to generate all suitable sequences of repairing instructions.

Attribute Repairs Translation

The problem with repairing sequences for attributes of a selected node in a data tree is hidden in the very quickly increasing number of combinations depending on the number of attributes to be removed or added. The prospective implementation of the correction algorithm should therefore consider to suggest repairs for attributes completely independently on corrections of the tree structure itself. However, we will follow the introduced formal model of edit operations.

As a consequence, we first need to generate all sequences of repairing instructions, that are capable to correct attributes with respect to given sets of required and optional attributes from the grammar context for a given node, thus with respect to given sets of invalid existing attributes and required missing attributes.

Definition 4.39 (Evaluation of Attribute Repairs)**.** *Assume that a tuple* $R_A = (R, A)$ *is an attribute repair as introduced in Definition 4.21, where* R, $A \in \mathbb{A}$ *are sets of attributes to be removed and added respectively.*

We define a value *of an attribute repair R_A to be a set $seq(R_A)$ of mutually not equivalent sequences of repairing instructions having the minimal cost, without redundancies and correcting attributes with respect to given sets of required and optional attributes.*

For $\forall m \in \mathbb{N}_0$, $0 \leq m \leq min(|R|, |A|)$ we put:

- $r_m = |R| - m$ *and* $a_m = |A| - m$.
- $R'_m = \{N \mid N \in \mathcal{P}(R)$ *and* $|N| = m\}$.
- $A'_m = \{N \mid N \in \mathcal{P}(A)$ *and* $|N| = m\}$.
- $X_m = R'_m \times A'_m$ *is the Cartesian product of* R'_m *and* A'_m.

For each selection of m source and target attributes to be involved in the attribute renaming procedure, i.e. $\forall x_m = (N_R, N_A) \in X_m$, we define sequence s_m to be any sequence of the form:

$$\langle(\texttt{renameAttribute}, n^S_1, n^T_1), \ldots, (\texttt{renameAttribute}, n^S_m, n^T_m),$$
$$(\texttt{removeAttribute}, n^D_1), \ldots, (\texttt{removeAttribute}, n^D_{r_m}),$$
$$(\texttt{addAttribute}, n^I_1), \ldots, (\texttt{addAttribute}, n^I_{a_m})\rangle$$

and satisfying the following conditions:

- $\forall i \in \mathbb{N}$, $1 \leq i \leq m$: $n^S_i \in N_R$, $n^T_i \in N_A$.
- $\forall i, j \in \mathbb{N}$, $1 \leq i < j \leq m$: $n^S_i \neq n^S_j$, $n^T_i \neq n^T_j$.
- $\forall i \in \mathbb{N}$, $1 \leq i \leq r_m$: $n^D_i \in R \setminus N_R$.
- $\forall i, j \in \mathbb{N}$, $1 \leq i < j \leq r_m$: $n^D_i \neq n^D_j$.
- $\forall i \in \mathbb{N}$, $1 \leq i \leq a_m$: $n^I_i \in A \setminus N_A$.
- $\forall i, j \in \mathbb{N}$, $1 \leq i < j \leq a_m$: $n^I_i \neq n^I_j$.

For $\forall m$ we define S_m to be a set of all such sequences s_m described above. In other words S_m represents a set of all sequences, where exactly m attribute pairs are corrected using the renameAttribute operation and all other using the appropriate addAttribute or removeAttribute.

Assuming $c_n = cost(renameAttribute)$, $c_r = cost(removeAttribute)$ and $c_a = cost(addAttribute)$, we define $seq(R_A)$ depending on the definition of the cost function:

- *If* $c_n < c_a + c_r$, *then* $seq(R_A) = S_{min(|R|,|A|)}$.
- *If* $c_n = c_a + c_r$, *then* $seq(R_A) = \bigcup_{m=0}^{min(|R|,|A|)} S_m$.
- *If* $c_n > c_a + c_r$, *then* $seq(R_A) = S_0$.

We only consider those sequences that are not equivalent. Since all these sequences modify only one particular node, their translation into edit sequences is simple. For technical reasons we will always use 0 as a base address for translations. This idea will be kept in all other translations too.

Note as well, that each sequence of repairing instructions generated using the previous definition has a cost corresponding to the minimal cost requested by Definition 4.21. We only need to discuss different relations between costs of individual edit operations and as a consequence to choose the minimal, whatever or maximal number of pairs of attributes to be involved in the renaming operation.

Definition 4.40 (Translation of Attribute Instructions)**.** *Given a sequence* $S_A = \langle s_1, \ldots, s_n \rangle$ *of attribute repairing instructions for some* $n \in \mathbb{N}_0$*, we define a* translation *of this sequence to be a sequence of attribute edit operations* $fix(S_A) = \langle f_1, \ldots, f_n \rangle$ *such that* $\forall k \in \mathbb{N}$*,* $1 \leq k \leq n$*:*

- *If* $s_k = (\texttt{addAttribute}, a)$*,*
  then $f_k = addAttribute(0, a)$*.*
- *If* $s_k = (\texttt{removeAttribute}, a)$*,*
  then $f_k = removeAttribute(0, a)$*.*
- *If* $s_k = (\texttt{renameAttribute}, a_0, a_1)$*,*
  then $f_k = renameAttribute(0, a_0, a_1)$*.*

We assumed that a*,* a_0 *and* $a_1 \in \mathbb{A}$ *were attribute names.*

Having a set S *of sequences of attribute repairing instructions, we define* $fix(S) = \{fix(S_A) \mid S_A \in S\}$*.*

Node Repairs Translation

Since a node repair contains only one sequence of repairing instructions, the evaluation simply encapsulates this sequence into a set with one element.

Definition 4.41 (Evaluation of Node Repairs)**.** *For a given node repair* R_N *we define* $seq(R_N) = \{R_N\}$ *to be a* value *of the node repair* R_N*.*

Each particular node repairing instruction is translated in a way presented in the following definition. We keep the concept of a base address equal to 0. One of the reasons to do it is connected with *addNode* edit operation. From the first point of view, address ϵ seems to be more suitable, but this would lead to inability to represent mentioned operation. The better idea is therefore to treat all base nodes as a sequence of nodes numbered from 0.

Definition 4.42 (Translation of Node Instructions)**.** *Given a sequence* $S_N = \langle s_1, \ldots, s_n \rangle$ *of node repairing instructions for some* $n \in \mathbb{N}_0$*, we define a* translation *of this sequence to be a sequence of node edit operations* $fix(S_N) = \langle f_1, \ldots, f_n \rangle$ *such that* $\forall k \in \mathbb{N}$*,* $1 \leq k \leq n$*:*

- *If* $s_k = (\texttt{addLeaf}, n)$*, then* $f_k = addLeaf(0, n)$.
- *If* $s_k = (\texttt{removeLeaf})$*, then* $f_k = removeLeaf(0)$.
- *If* $s_k = (\texttt{renameLabel}, n)$*, then* $f_k = renameLabel(0, n)$.
- *If* $s_k = (\texttt{addNode}, n, i)$*, then* $f_k = addNode(0, 0 + i, n)$.
- *If* $s_k = (\texttt{removeNode})$*, then* $f_k = removeNode(0)$.

We assumed that $n \in \mathbb{E}$ *was an element name and* $i \in \mathbb{N}_0$.

Having a set S *of sequences of node repairing instructions, we define* $fix(S) = \{fix(S_N) \mid S_N \in S\}$.

4.6.2 Repairs for Sequences and Intents

Now we will focus the problem of generation and translation of sequences for repairs of node sequences and intents. Contrary to the previously discussed repairs, these two are not isolated only within one level of recursion and we need to introduce an inductive definition. At the bottom of the recursion, we construct repairing sequences only from repairs of nodes and attributes. Backtracking towards the root of a processed data tree, we need to inspect repairing multigraphs depending on particular intent types.

Auxiliary Translation Functions

Before we can continue, we need to introduce three auxiliary functions, which will be used for manipulating addresses of nodes occurring in edit operations. The first function called *modPre* prepends specified non-negative integer before addresses of all nodes, function *modAlt* modifies the first number in addresses to a new number and, finally, function *modCut* truncates the first number in addresses.

Definition 4.43 (Auxiliary Translation Functions)**.** *Assume that* $S_E = \langle s_1, \ldots, s_n \rangle$ *is a sequence of edit operations for some* $n \in \mathbb{N}_0$. *We define* modPre, modAlt *and* modCut *to be functions transforming* S_E *into* $S'_E = \langle s'_1, \ldots, s'_n \rangle$ *such that* $\forall k \in \mathbb{N}$, $1 \le k \le n$:

- *If* $s_k = addAttribute(u, a)$,
  then $s'_k = addAttribute(f(u), a)$.
- *If* $s_k = removeAttribute(u, a)$,
  then $s'_k = removeAttribute(f(u), a)$.
- *If* $s_k = renameAttribute(u, a_0, a_1)$,
  then $s'_k = renameAttribute(f(u), a_0, a_1)$.
- *If* $s_k = addLeaf(u, n)$*, then* $s'_k = addLeaf(f(u), n)$.

- *If* $s_k = removeLeaf(u)$, *then* $s'_k = removeLeaf(f(u))$.
- *If* $s_k = renameLabel(u, n)$, *then* $s'_k = renameLabel(f(u), n)$.
- *If* $s_k = addNode(u_0, u_1, n)$, *then* $s'_k = addLeaf(f(u_0), f(u_1), n)$.
- *If* $s_k = removeNode(u)$, *then* $s'_k = removeNode(f(u))$.

We have assumed that $u, u_0, u_1 \in \mathbb{N}_0^*$, $a, a_0, a_1 \in \mathbb{A}$, $n \in \mathbb{E}$ *and that function* f *is defined depending on* modPre, modAlt *and* modCut *alternatives this way:*

- *For* $modPre(S_E, c)$, *where* $c \in \mathbb{N}_0$, *we define* $f(u) = c.u$.
- *For* $modAlt(S_E, c)$, *where* $c \in \mathbb{N}_0$, *we furthermore assume that* $u = i.v$ *for some* $i \in \mathbb{N}_0$, $v \in \mathbb{N}_0^*$, *and then we define* $f(u) = (i + c).v$.
- *For* $modCut(S_E)$, *we furthermore assume that* $u = i.v$ *for some* $i \in \mathbb{N}_0$, $v \in \mathbb{N}_0^*$, *and then we define* $f(u) = v$.

Having a set S *of sequences of edit operations, we analogously define:*

- $modPre(S, c) = \{modPre(S_E, c) \mid S_E \in S\}$,
- $modAlt(S, c) = \{modAlt(S_E, c) \mid S_E \in S\}$ *and*
- $modCut(S) = \{modCut(S_E) \mid S_E \in S\}$.

Node Sequence Repairs Translation

Having a particular repairing path in the repairing multigraph, we can generate all repairing sequences by combining repairs associated with edges of such path.

Our goal is however not only to generate all sequences of repairing instructions, we also need to translate them. For one particular path this translation is easy, because we need only to monitor correction intents along the processed path and determine the number of nodes the given intents involve at their top level.

Definition 4.44 (Translation of Node Sequence Repairs). *Assume that* R_S *is a node sequence repair corresponding to a repairing multigraph* $R(\mathcal{I}) = (V', E', pathPrev, pathCost, termCost)$ *for a correction intent* $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$, *where* $u = \langle u_1, \ldots, u_n \rangle$ *for some* $n \in \mathbb{N}_0$.

We define $fix(R_S)$ *or* $fix(R_S, q_T)$ *to be a* translation *of a node sequence repair* R_S, *where* $fix(R_S)$ *is a set of all* sequence mends *for shortest paths in the given aggregated repairing multigraph for correction intent* $\mathcal{I}$ *and* $fix(R_S, q_T)$ *for* $\forall q_T \in Q_T$ *a set of all sequence mends for shortest paths to state* q_T *in the last stratum in the given separated repairing multigraph.*

A sequence mend for a particular shortest path is a pair $(S, shift)$, *where* S *is a corresponding sequence of edit operations derived from the path and* $shift$ *is a number of processed nodes as defined subsequently.*

If f_T = aggregated*, then* $\forall p \in P^{min}_{v_S,V_T}$*,* $p = \langle e_1, \ldots, e_m \rangle$*, where* $m \in \mathbb{N}_0$ *and* $\forall i \in \mathbb{N}$*,* $1 \leq i \leq m$*,* $e_i = (v^i_1, v^i_2, \mathcal{I}^i, \mathcal{R}_{\mathcal{I}^i}, c^i)$*,* $v^i_2 = (s^i_2, q^i_2)$*:*

- $S_p = \{(modAlt(s^1, a_0).modAlt(s^2, a_1) \ldots modAlt(s^m, a_{m-1}), shift) \mid \forall i \in \mathbb{N},\ 1 \leq i \leq m,\ (s^i, shift^i) \in fix(\mathcal{I}^i),\ a_i = a_{i-1} + shift^i,\ a_0 = 0\}$.
- *Value* $shift$ *in each previous mend is defined this way:*
  - *If* $y \in \{\texttt{insert}, \texttt{repair}, \texttt{rename}, \texttt{push}\}$*, then* $shift = 1$*.*
  - *If* $y = \texttt{delete}$*, then* $shift = 0$*.*
  - *If* $y = \texttt{pull}$*, then* $shift = a_m$*.*
- *Finally, we define* $fix(R_S) = \bigcup_{p \in P^{min}_{v_S,V_T}} S_p$*.*

If f_T = separated*, then* $\forall q_T \in Q_T$*,* $v_T = (n, q_T)$*,* $\forall p \in P^{min}_{v_S,v_T}$*,* $p = \langle e_1, \ldots, e_m \rangle$*, where* $m \in \mathbb{N}_0$ *and* $\forall i \in \mathbb{N}$*,* $1 \leq i \leq m$*,* $e_i = (v^i_1, v^i_2, \mathcal{I}^i, \mathcal{R}_{\mathcal{I}^i}, c^i)$*,* $v^i_2 = (s^i_2, q^i_2)$*:*

- *We first determine* S_p *in the same way as above,*
- *and then we define* $fix(R_S, q_T) = \bigcup_{p \in P^{min}_{v_S,v_T}} S_p$ *for* $\forall q_T \in Q_T$*.*

The processing of aggregated and separated repairing multigraphs is very similar. In fact it differs only in paths, which are considered, and the way of their separation.

Intent Repairs Translation

Finally, we can inspect the translation of repairs for correction intents. Since individual intents combine associated node, attribute and node sequence repairs in different order, we need to handle each intent type separately. We also need to handle terminating intents specially, since these intents do not have nested the node sequence repair.

The general idea is fortunately common to all intents. We compute the translations for all sequences derived from the associated node and attribute repairs and then we combine these edit sequences with edit sequence already translated from shortest paths in the repairing multigraph. In case of `correct` starting correction intent, we only process the nested sequence repair and, finally, we need to truncate the leading 0 from addresses of all nodes occurring in any edit operation, since the root node of the entire data tree has an address equal to ϵ.

Definition 4.45 (Translation of Intent Repairs)**.** *Let* $\mathcal{R}_{\mathcal{I}} = (R_N, R_A, R_S, cost)$ *be a repair for a correction intent* $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$*.*

We define a translation *of the intent repair* $\mathcal{R}_{\mathcal{I}}$ *to be a set of all* intent mends *for sequences of edit operations having the minimal cost and reflecting the given correction intent* $\mathcal{I}$*.*

If $\mathcal{I}$ is an aggregated correction intent, we denote this translation by $fix(\mathcal{R}_\mathcal{I})$, otherwise for a separated intent we define a translation for $\forall q_T \in Q_T$ separately by $fix(\mathcal{R}_\mathcal{I}, q_T)$.

- *If* $y =$ `correct`*:*
  - $fix(\mathcal{R}_\mathcal{I}) = \{(modCut(r_S), \bot) \mid (r_S, shift) \in fix(R_S)\}$.
- *If* $y =$ `insert`*:*
  - *If $\mathcal{I}$ is a terminating intent, then* $fix(\mathcal{R}_\mathcal{I}) = \{(r_N.r_A, 0) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A))\}$.
  - *Otherwise* $fix(\mathcal{R}_\mathcal{I}) = \{(r_N.r_A.modPre(r_S, 0), shift) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A)),\ (r_S, shift) \in fix(R_S)\}$.

  $\forall f \in fix(\mathcal{R}_\mathcal{I})$: $f = insertSubtree(0)$ *update operation.*
- *If* $y =$ `delete`*:*
  - *If $\mathcal{I}$ is a terminating intent, then* $fix(\mathcal{R}_\mathcal{I}) = \{(r_A.r_N, 0) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A))\}$
  - *Otherwise* $fix(\mathcal{R}_\mathcal{I}) = \{(modPre(r_S, 0).r_A.r_N, shift) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A)),\ (r_S, shift) \in fix(R_S)\}$

  $\forall f \in fix(\mathcal{R}_\mathcal{I})$: $f = deleteSubtree(0)$.
- *If* $y =$ `repair`*, then:*
  - *If $\mathcal{I}$ is a terminating intent, then* $fix(\mathcal{R}_\mathcal{I}) = \{(r_A, 0) \mid r_A \in fix(seq(R_A))\}$.
  - *Otherwise* $fix(\mathcal{R}_\mathcal{I}) = \{(r_A.modPre(r_S, 0), shift) \mid r_A \in fix(seq(R_A)),\ (r_S, shift) \in fix(R_S)\}$.

  $\forall f \in fix(\mathcal{R}_\mathcal{I})$: $f = repairSubtree(0)$.
- *If* $y =$ `rename`*, then:*
  - *If $\mathcal{I}$ is a terminating intent, then* $fix(\mathcal{R}_\mathcal{I}) = \{(r_N.r_A, 0) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A))\}$.
  - *Otherwise* $fix(\mathcal{R}_\mathcal{I}) = \{(r_N.r_A.modPre(r_S, 0), shift) \mid r_N \in fix(seq(R_N)),\ r_A \in fix(seq(R_A)),\ (r_S, shift) \in fix(R_S)\}$.

  $\forall f \in fix(\mathcal{R}_\mathcal{I})$: $f = renameSubtree(0)$.

- *If* $y = \texttt{push}$*:*
  - $fix(\mathcal{R}_\mathcal{I}) = \{(r_N.r_A.modPre(r_S, 0), shift) \mid$
    $r_N \in fix(seq(R_N))$, $r_A \in fix(seq(R_A))$,
    $(r_S, shift) \in fix(R_S)\}$.

  $\forall f \in fix(\mathcal{R}_\mathcal{I})$: $f = pushSiblings(0, s_E - s_I - 1)$.

- *If* $y = \texttt{pull}$*, then for* $\forall q_T \in Q_T$*:*
  - $fix(\mathcal{R}_\mathcal{I}, q_T) = \{(r_A.r_N.r_S, shift) \mid r_N \in fix(seq(R_N))$,
    $r_A \in fix(seq(R_A))$, $(r_S, shift) \in fix(R_S, q_T)\}$.

  $\forall f \in fix(\mathcal{R}_\mathcal{I}, q_T)$: $f = pullSiblings(0)$.

Note that addresses of nodes are reconstructed in a bottom up way and we do not use any direct knowledge about positions from the original data tree. Moreover performing an edit operation we can achieve changes in addresses of former or newly subordinated nodes, thus we need to be aware of this addressing variability.

The previous definition also declares the connection of correction intents and update operations. Once having a particular sequence of repairing instructions translated into a sequence of edit operations, we can classify such sequence using definitions for update operations. As already outlined, the correction algorithm is not able to generate all potentially existing sequences that would conform to introduced update operations. But the algorithm is always able to find corrections and these can be viewed via updates.

4.7 Correction Algorithms

The purpose of the previous section was to completely introduce the correction capabilities of the proposed correction algorithm. We have abstracted the proceeding of the algorithm using multigraphs. Their purpose was to statically describe all repairs that can be generated in the limits of introduced correction intents.

The basic idea of these multigraphs originates from the complete traversal of the state space of the Glushkov automaton, which is derived from the regular expression restricting the element content of a given node in a data tree. We are generally able to perform this processing dynamically, or we can construct the multigraph and then inspect all possible repairs statically. This has also another advantage, because multigraphs not only record the algorithm flow, but also store the computed results in a form of the shortest paths to some specified vertices, usually corresponding to the accepting output states of the automaton itself.

In this section we will describe four different versions of the correction algorithm. Starting with the *naive* correction algorithm we will directly follow introduced formal framework of multigraphs. Next, we will discuss *dynamic*

algorithm, which directly constructs only the required portion of the repairing multigraph. The last but one algorithm stands for *caching* correction algorithm capable to avoid repeated computations of identical intents. Finally, we will present *incremental* algorithm, which is able to work efficiently even to the depth of recursively invoked intents.

4.7.1 Naive Correction Algorithm

The first presented algorithm strictly follows the idea of defined multigraphs. Therefore, the naive correction algorithm really first constructs the complete exploration multigraph, then it evaluates all its edges by invoking the computations of recursively nested correction intents and, finally, it constructs the repairing multigraph, which can be simply turned into the repair for the given intent.

Algorithm Description

The correction starts by the invocation of the recursive routine with the starting context $\mathcal{I}_\bullet$. At each level of the recursion we first prepare the entire exploration multigraph with its all vertices and edges.

The second step represents the evaluation of all nested intents. This means that we compute all generated correction intents and bind returned repairs to corresponding edges in the correction multigraph. The last but one step represents the searching for all shortest paths, since these paths involve all found suitable repairs, which are as close to the original data tree as the notion of correction intents admits.

The following enumeration gives an overview of all input and output parameters, which are passed into and from the correction routine respectively:

- *The problem defining input parameters:* data tree $\mathcal{T} = (D, lab, val, att)$ and single type tree grammar $\mathcal{G} = (N, T, A, S, P)$.
- *The intent description input parameters:* correction intent type y, node repair R_N and attribute repair R_A, initial stratum number s_I and state q_I in s_I, ending stratum number s_E and set of ending states in s_E.
- *The intent assignment input parameters:* sequence of nodes $u = \langle u_1, \ldots, u_n \rangle$ to be processed, depth d of processed nodes in a new tree, grammar context $\mathcal{C} = (a_R, n_R, N_c, P_c, map, r, C, O)$, flags f_S and f_T, starting state q_S of $\mathcal{A}_r$, set of terminating states Q_T of $\mathcal{A}_r$ and set of allowed correction intent types Y_a.
- *The output parameter:* repair structure $\mathcal{R}_\mathcal{I} = (R_N, R_A, R_S, cost)$.

Main Procedure Definition

The main procedure presented in Algorithm 1 reflects the sequence of steps that should be executed. First we prepare the exploration multigraph, then we compute the correction multigraph, derive the repairing multigraph and, finally, encapsulate the found repairs into the repair structure.

Algorithm 1: naiveCorrectionAlgorithm

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, intent $\mathcal{I}$.
Output: Repair $\mathcal{R}_{\mathcal{I}}$ for correction intent $\mathcal{I}$.

```
1 E(I) ← createExplorationMultigraph(T, G, I);
2 C(I) ← createCorrectionMultigraph(T, G, I, E(I));
3 R(I) ← createRepairingMultigraph(T, G, I, C(I));
4 R_I ← composeIntentRepair(I, R(I));
5 return R_I
```

Procedure for Exploration Multigraph Creation

The procedure introduced in Algorithm 2 for the exploration multigraph construction starts with an empty multigraph and then successively adds new vertices and edges following Definition 4.27 of recursive correction intents.

Algorithm 2: createExplorationMultigraph

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, intent $\mathcal{I}$.
Output: Exploration multigraph $\mathcal{E}(\mathcal{I})$.

1 $\mathcal{E}(\mathcal{I}) = (V_E, E_E) \leftarrow (\emptyset, \emptyset)$;
2 $V_E \leftarrow \{(0, q_S)\}$;
3 $NewVertices \leftarrow \{(0, q_S)\}$;
4 **while** $\exists v_1 \in NewVertices$, $v_1 = (s, q)$ **do**
5   **foreach** *correction intent* $\mathcal{I}' = (y', R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d', \mathcal{C}', f'_S, f'_T, q'_S, Q'_T, Y'_a)$ *defined in stratum s and state q* **do**
6   **foreach** $q'_E \in Q'_E$ **do**
7             $v_2 \leftarrow (s'_E, q'_E)$;
8   **if** $v_2 \notin V_E$ **then**
9   Add vertex v_2 into V_E and $NewVertices$;
10   Add edge $(v_1, v_2, \mathcal{I}')$ into E_E;
11   Remove v_1 from $NewNodes$;

Procedure for Correction Multigraph Creation

The purpose of Algorithm 3 is to derive the corresponding correction multigrap from a provided and fully constructed exploration multigraphh. Thus the main task is to invoke the recursive computations of prepared correction intents and then store returned repairs inside edges of the correction multigraph.

Algorithm 3: createCorrectionMultigraph

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, correction intent $\mathcal{I}$, exploration multigraph $\mathcal{E}(\mathcal{I}) = (V_E, E_E)$.
Output: Correction multigraph $\mathcal{C}(\mathcal{I})$.

```
1 C(I) = (V_C, E_C) ← (V_E, ∅);
2 foreach v_1 ∈ V_C, v_1 = (q, s) do
3     foreach correction intent I' = (y', R'_N, R'_A, s'_I, q'_I, s'_E, Q'_E, u', d',
      C', f'_S, f'_T, q'_S, Q'_T, Y'_a) defined in stratum s and state q do
4         repair ← naiveCorrectionRoutine(T, G, I');
5         foreach q'_E ∈ Q'_E do
6             v_2 ← (s'_E, q'_E);
7             if f'_T = aggregated then c ← cost(repair);
8             else c ← cost(repair, q'_E) ; f'_T = separated
9             Add correcting edge (v_1, v_2, I', repair, c) into E_C;
```

Procedure for Repairing Multigraph Creation

The repairing multigraph is a subgraph of the correction multigraph with only those vertices and edges that are involved in the found shortest paths to the given set of target vertices.

Its construction is presented in Algorithm 4, where we have just followed the idea of standard Dijkstra's algorithm [24] for searching shortest paths in graphs.

We start the processing of the multigraph in the source vertex specified by the correction intent itself. Then we step by step walk through the multigraph and in each step, we first select the previously reached vertex, which has the minimal *pathCost* between all reached vertices. The algorithm ensures that this cost becomes final for this vertex, since it cannot be improved any more.

If we need to generate an `aggregated` repair, we have already closed the first target vertex and if the current path cost has strictly greater value than the cost of the target vertex that has been wiped as the first item from the working set of target vertices, we can stop the loop. Analogously if we need to generate `separated` repair, we are forced to operate until the last vertex from the set of target vertices is marked as closed and then wait until the current cost exceeds the fixed value from the last wiped target vertex.

Algorithm 4: createRepairingMultigraph

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, correction intent $\mathcal{I}$ $= (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$, correction multigraph $\mathcal{C}(\mathcal{I}) = (V_R, E_R)$.

Output: Repairing multigraph $\mathcal{R}(\mathcal{I})$.

1 $\mathcal{R}(\mathcal{I}) = (V_R, E_R, pathPrev, pathCost, termCost) \leftarrow (V_R, E_R, \emptyset, \emptyset, \perp)$;

2 $v_S \leftarrow (0, q_S)$; $pathCost(v_S) \leftarrow 0$; $pathPrev(v_S) \leftarrow \emptyset$;

3 $reachedVertices \leftarrow \{v_S\}$; $closedVertices \leftarrow \emptyset$;

4 $targetVertices \leftarrow \{(n, q_T) \mid q_T \in Q_T\}$;

5 **while** $reachedVertices \neq \emptyset$ **do**

6 $\quad m \leftarrow min_{v \in reachedVertices} pathCost(v)$;

7 $\quad v_1 = (s, q)$ for some $v_1 \in reachedVertices$, $pathCost(v_1) = m$;

8 $\quad$ **if** $[(targetVertices = \{v_1\})$ *or* $(f_T =$ `aggregated` *and* $v_1 \in targetVertices)]$ *and* $[finalCost$ *is not defined*$]$ **then**

9 $\quad\quad finalCost \leftarrow pathCost(v_1)$;

10 $\quad$ Remove v_1 from $reachedVertices$ and $targetVertices$;

11 $\quad$ Add v_1 to $closedVertices$;

12 $\quad$ **if** $finalCost$ *is defined and* $finalCost < pathCost(v_1)$ **then**

13 $\quad\quad$ **break**

14 $\quad$ **foreach** $e \in E_R$, $e = (v_1, v_2, i, R_i, c)$ **do**

15 $\quad\quad c' \leftarrow pathCost(v_1) + c$;

16 $\quad\quad$ **if** $pathCost(v_2)$ *is defined and* $pathCost(v_2) = c'$ **then**

17 $\quad\quad\quad pathPrev(v_2) \leftarrow pathPrev(v_2) \cup \{v_1\}$;

18 $\quad\quad$ **if** $pathCost(v_2)$ *is not defined or* $pathCost(v_2) > c'$ **then**

19 $\quad\quad\quad pathCost(v_2) \leftarrow c'$; $pathPrev(v_2) \leftarrow \{v_1\}$;

20 $\quad\quad$ **if** $v_2 \notin closedVertices$ *and* $v_2 \notin reachedVertices$ **then**

21 $\quad\quad\quad$ Add v_2 to $reachedVertices$;

22 **if** $f_T =$ `aggregated` **then** $termCost \leftarrow finalCost$;

23 **else** $\forall v_T \in V_T$ $termCost(v_T) \leftarrow pathCost(v_T)$; $f_T =$ `separated`

Otherwise we process all outgoing edges from the current vertex and if we are able to reach the given ending vertex with the same or better path cost, we change it and update the set of preceding vertices. Thus we are able to backtrack and reconstruct all shortest paths.

Finally, we need to clean the multigraph from unnecessary vertices and edges. This step is easy and is not included in Algorithm 4.

4.7.2 Dynamic Correction Algorithm

The main disadvantage of the naive correction algorithm is that we usually uselessly construct the entire exploration multigraph and later on in order to create the derived creation multigraph, we evaluate all prepared nested correction intents. It is easy to see, that we can propose an algorithm, that directly attempts to search for all shortest paths and during this search it constructs only the required part of the multigraph. This can lead to time savings, since we do not need to compute all correction edges.

Suggested improvements are considered in Algorithm 5. It does not work in the consecutive steps based on the exploration, correction and repairing multigraphs construction, but it straightforwardly builds the repairing multigraph, which at the end only needs to be cleaned from vertices and edges, that are not located at any of the found shortest paths.

Although we now evaluate only those nested intents, that we need to inspect during the procedure of searching shortest paths, there still remains other problems causing significant inefficiency. We will thus attempt to make two more improvements.

4.7.3 Caching Correction Algorithm

During the computation of the naive or dynamic correction algorithm, we are quite often forced to compute the same information, as we have already computed probably many times before.

Suppose that we are in some of the nodes of a given data tree and we have decided for example to change its label. This means that we consequently need to recursively process all its child nodes in a few alternative ways, one for each pair of allowed terminal and nonterminal symbol used for renaming. It is highly probable that the processing of these nodes and all their subtrees will be similar. In other words there would be chance for any type of correction intent to be evaluated repeatedly.

As a particular example we can name for example the deletion intent. It is easy to sea, that there can be various different chains of nested intents, but whenever we decide to delete a subtree on a given position in the original data tree, the deletion will always proceed identically, since only the data subtree itself defines the generation of nested deletion intents and thus corresponding repair.

Similarly whenever we decide to insert a subtree with a specified terminal symbol as a label for a new root node and nonterminal symbol describing the context to be used, the proceeding of this insert correction intent will always be the same.

To harness this finding and in order to introduce much more efficient algorithm, we are going to introduce the model of caching. Its main purpose is to provide the environment, where nothing already computed is forgotten, since it can be reused again.

Algorithm 5: dynamicCorrectionRoutine

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, intent $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$.

Output: Repair $\mathcal{R}_\mathcal{I}$ for intent $\mathcal{I}$.

```
1  R(I) ← ({(0, q_S)}, ∅, ∅, ∅, ⊥);
2  v_S ← (0, q_S); pathCost(v_S) ← 0; pathPrev(v_S) ← ∅;
3  reachedVertices ← {v_S}; targetVertices ← {(n, q_T) | q_T ∈ Q_T};
4  while reachedVertices ≠ ∅ do
5      m ← min_{v∈reachedVertices} pathCost(v);
6      v_1 = (s, q) for some v_1 ∈ reachedVertices, pathCost(v_1) = m;
7      if [(targetVertices = {v_1}) or (f_T = aggregated and
       v_1 ∈ targetVertices)] and [finalCost is not defined] then
8          finalCost ← pathCost(v_1);
9      if finalCost is defined and finalCost < pathCost(v_1) then
10         break
11     foreach correction intent I′ = (y′, R′_N, R′_A, s′_I, q′_I, s′_E, Q′_E, u′, d′,
       C′, f′_S, f′_T, q′_S, Q′_T, Y′_a) defined in stratum s and state q do
12         repair ← dynamicCorrectionRoutine(T, G, I′);
13         foreach q′_E ∈ Q′_E do
14             v_2 ← (s′_E, q′_E);
15             if f′_T = aggregated then c ← cost(repair);
16             else c ← cost(repair, q′_E) ; f′_T = separated
17             if v_2 ∉ V_E then Add v_2 into V_E and reachedVertices;
18             Add correcting edge (v_1, v_2, I′, repair, c) into E_R;
19             c′ ← pathCost(v_1) + c;
20             if pathCost(v_2) is defined and pathCost(v_2) = c′ then
21                 pathPrev(v_2) ← pathPrev(v_2) ∪ {v_1};
22             if pathCost(v_2) is undefined or pathCost(v_2) > c′ then
23                 pathCost(v_2) ← c′; pathPrev(v_2) ← {v_1};
24     Remove v_1 from reachedVertices and targetVertices;
25 if f_T = aggregated then termCost ← finalCost;
26 else ∀v_T ∈ V_T termCost(v_T) ← pathCost(v_T) ; f_T = separated
27 return R_I ← composeIntentRepair(I, R(I));
```

Signatures of Correction Intents

Our goal is to describe correction intents, which certainly lead to the same repairs, i.e. intents with the identical assignment for a nested recursion level of processing. Having a particular type of a correction intent, the notion of an intent signature solves our problem.

Definition 4.46 (Correction Intent Signature). *Assume that* $\mathcal{T} = (D, lab, val, att)$ *is a data tree and* $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$ *is a correction intent where* $\mathcal{C} = (a_R, n_R, N_c, P_c, map, r, C, O)$ *is a grammar context and* $u = \langle u_1 \dots u_n \rangle$ *is a node sequence of length* n.

We define a signature $\mathcal{S}(\mathcal{I})$ *of a correction intent to be a tuple, which fulfils the following conditions:*

- *If* $y = \texttt{correct}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{correct})$.
- *If* $y = \texttt{insert}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{insert}, n_R, a_R)$.
- *If* $y = \texttt{delete}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{delete}, baseNode(\mathcal{I}))$.
- *If* $y = \texttt{repair}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{repair}, baseNode(\mathcal{I}), n_R)$.
- *If* $y = \texttt{rename}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{rename}, baseNode(\mathcal{I}), n_R, a_R)$.
- *If* $y = \texttt{push}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{push}, baseNode(\mathcal{I}), n, n_R, a_R)$.
- *If* $y = \texttt{pull}$, *then* $\mathcal{S}(\mathcal{I}) = (\texttt{pull}, baseNode(\mathcal{I}), n_R, a_R, q_S)$.

The `insert` correction intent is only described by the terminal symbol and a production rule to be used. This rule uniquely implies the nonterminal symbol. In case of `delete` intent, we only need to specify the position in the original data tree.

Correction intents `repair` and `rename` are similar, but the former one does not need to specify the terminal symbol, because this symbol equals to the node label in the original data tree.

The `push` intent needs to be described by the range of sibling nodes that should be pushed one level lower. The label and nonterminal symbol of their new parent node uniquely defines the nested processing. Finally, the `pull` intent is described by a pair of terminal and nonterminal symbol again, but these symbols belong to the original grandparent of pulled nodes, thus their new parent to whose context they should now conform. The source state is required too, since it influences the set of reachable target states.

Caching Repository Model

Now we can introduce the notion for a caching repository, which serves for storing already computed repairs. The caching version of the correction algorithm first attempts to ask this cache and only if the specified correction intent has not yet been computed, we invoke its computation. Otherwise we can access the cached repair we need.

Definition 4.47 (Caching Manager). *A* caching manager *is a data structure acting as a partial function, which is able to assign a corresponding repair to a given signature of an intent* $\mathcal{I}$, *denoted by* $cacheRepair(signature(\mathcal{I})) = R_{\mathcal{I}}$.

At the beginning of the correction algorithm the caching repository is empty and this means, that the internal *cache* function is not defined for any signature. During the recursive processing, whenever we finish computation of a given level and thus have evaluated a repair for a given correction intent, we extend this function, i.e. insert newly computed repair into the cache.

Algorithm Description

The newly presented caching Algorithm 6 is almost identical to the dynamic correction algorithm. We only need to modify the code near the place of the nested recursion level invocation. We first attempt to fetch the already computed repair from the caching repository and only if we are not successful, we invoke its computation.

Algorithm 6: cachingCorrectionRoutine

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, intent $\mathcal{I}$.
Output: Repair for correction intent $\mathcal{I}$.

```
// Lines 1 to 11 from Algorithm 5
// Adjusted line 12 from Algorithm 5
1 if cache(signature(I)) is defined then
2 |   repair ← cacheRepair(signature(I));
3 else
4 |   repair ← cachingCorrectionRoutine(T, G, I');
5 |   cacheRepair(signature(I)) ← repair;
// Lines 13 to 27 from Algorithm 5
```

4.7.4 Incremental Correction Algorithm

The dynamic correction algorithm efficiently processes each level of recursion, because it only invokes evaluation of those edges that need to be inspected in order to find all required shortest paths. Therefore, we are able to avoid processing of those directions in a given partial repairing multigraph, which seem not to be perspective. The main problem however remains. Although we ignore some edges, whenever we decide to evaluate one edge, we need to inspect the entire recursion hidden in this single edge. Even if this edge would not be the good direction to acquire the shortest path, we need to evaluate it completely.

The idea of the incremental correction algorithm is to work efficiently even with the recursion. Suppose that we are at the top level, thus we are processing the starting correction intent and we need to find the shortest paths to correct the entire tree. This means that we still need to follow standard Dijkstra's algorithm, but we can act lazily. This means that whenever we need to evaluate

a correction intent associated with a given edge, we only initiate the process of its incremental computation. The algorithm will work in steps, permanently attempting to give more precision to partially evaluated intents. Step by step we will obtain more precise estimations of edge costs and dynamically follow only directions, which seem to be the most promising.

Tasks Model

The main difference to all previously introduced correction algorithms is the fact that we cannot directly use nested calls of the correction routine. We need to introduce the model, where we only tell what to compute and the particular computations will be controlled by some manager. This also means that we need a data structure with ability to store the complete context describing the state of elaboration for each single correction intent.

Definition 4.48 (Intent Computation Task)**.** *Assume that* $\mathcal{I} = (y, R_N, R_A, s_I, q_I, s_E, Q_E, u, d, \mathcal{C}, f_S, f_T, q_S, Q_T, Y_a)$ *is a correction intent. We define a* task *for* $\mathcal{I}$ *to be a tuple* $\mathcal{K}_\mathcal{I} = (\mathcal{I}, \mathcal{R}(\mathcal{I}), state, stamp, V_{reached}, E_{reached}, E_{delayed}, V_T, c_{reached}, c_{fixed}, W_A, W_D)$*, where:*

- $\mathcal{I}$ *is the correction intent.*
- $\mathcal{R}(\mathcal{I})$ *is the partially evaluated repairing multigraph for* $\mathcal{I}$*. This means that the multigraph is not complete and function or constant* termCost *returns only estimated values.*
- $state \in \{\texttt{suspended}, \texttt{blocked}, \texttt{activated}, \texttt{signalled}\}$
- $stamp \in \mathbb{N}_0$ *is a timestamp of the last incremental processing.*
- $V_{reached} \subseteq V_C$ *is a set of* reached vertices*.*
- $E_{reached} \subseteq E_C$ *is a set of* reached edges*.*
- $E_{delayed} \subseteq E_C$ *is a set of* delayed edges*.*
- $V_T \subseteq V_C$ *is a set of* opened target vertices*.*
- $c_{reached}$ *is a value of* reached cost*.*
- c_{fixed} *is a value of* fixed cost *or* $\perp$*.*
- W_A *is a set of correction tasks waiting for* $K_\mathcal{I}$*.*
- W_D *is a set of correction tasks for which* $K_\mathcal{I}$ *is waiting.*

It is easy to see that we have in principle only encapsulated local variables from Algorithm 5. These are especially items $V_{reached}$ for storing reached and not yet closed vertices, V_T representing a set of not yet closed target vertices, $c_{reached}$ for reached cost of the inspected node and also c_{fixed} for storing the

fixed value of reached cost in order to stop the entire intent processing as soon as it is clear that no better paths cannot certainly be found.

Passing over the item for correction intent reference $\mathcal{I}$, we need to little bit discuss $\mathcal{R}(\mathcal{I})$. It has been sketched that this repairing multigraph is not fully constructed, but also not fully evaluated. The dynamic correction algorithm always fully evaluates the corresponding intent repair, whenever the correction edge is inserted into the graph. Unfortunately this is not our goal, thus we will support also partially defined edges. If the reference to the intent repair is not defined in the edge, we know that the given intent has not yet been completely computed. In this case the cost value stored in the given edge only estimates its final value. Moreover this estimation even needs not to be continuously current.

After a new task is created, it is assigned with `suspended` state. If another task requests computation of its nested and not yet computed intent, the nested one is assigned `activated` state and the parental one `blocked` state. If all tasks the given parent is waiting for are executed, we assign state `signalled` to this parental task.

In order to have the complete overview of existing relations between blocked tasks, there are two sets W_A and W_D storing lists of waiting tasks.

Item *stamp* is an auxiliary timestamp value representing the last time the given task was refined. Even though the incremental correction algorithm does not need to handle with these stamps, they enable easier detection of changes. Assume that a given intent has requested computation of several nested intents. It is easy to remember their references and after these computations are finished, we can renew the knowledge of cost estimations of all involved edges. Unfortunately there can be other edges that have also made some progress.

Finally, there are $E_{reached}$ and $E_{delayed}$ items representing the sets of reached and delayed edges respectively. Note that whenever we discover new vertices in the partially constructed repairing multigraph, we postpone their evaluation. Thus we need to know which edges are considered and from this set, which edges end at the currently most perspective vertex.

Computation Manager

The computation manager is a data structure capable to store all existing tasks. In other words it acts similarly as the caching manager for caching repairs. Furthermore it contains a provider for timestamps. We do not actually need to use real timestamps, ordinary increasing sequence of natural numbers fulfils the same objective.

Definition 4.49 (Computation Manager)**.** *A* computation manager *is a data structure containing a* cacheTask *function,* timeStamp *function and a tuple* $\mathcal{M} = (Suspended, Blocked, Signalled, Activated)$*, where:*

- cacheTask *is a partial function, which is able to assign to a given signature of an intent $\mathcal{I}$ assign its associated existing task, denoted by $cacheTask(signature(\mathcal{I})) = K_{\mathcal{I}}$.*

- timeStamp *provider is a function returning values from* $\mathbb{N}$ *serving as timestamps. Whenever this function is called, it returns a value equal to the previously returned value increased by* 1*.*

- *Suspended is a set of tasks having state* = `suspended`*.*

- *Blocked is a set of tasks having state* = `blocked`*.*

- *Signalled is a set of tasks having state* = `aignalled`*.*

- *Activated is a set of tasks having state* = `activated`*.*

All existing tasks are cached within the computation manager in four disjoint sets separated by task states. We actually only need the direct access to sets of *Signalled* and *Activated*, because the main correction routine works in a cycle and in each step it invokes the computation of some task corresponding one of these two states.

Whenever we first encounter a correction intent, we create a task encapsulating this intent and we put this task structure into the cache of tasks in the computation manager. After its computation is fully finished, we destroy the task structure and put the intent repair into the standard caching repository for repairs.

Main Correction Routine

The main routine of the incremental correction algorithm is presented in Algorithm 7. Its purpose is to create the starting intent for correction of the entire tree, start up its processing, continuously manage it and, finally, fetch the repair structure with corrections for the data tree itself.

The life cycle of each task begins in the state `suspended`. This is quite important, because this means that the given task cannot be passed to processing until some running task explicitly requests this. In a such situation there is a clear relation between the requesting task and the requested tasks.

The main loop continually fetches prepared and requested tasks and invokes their incremental computation. Note that there is at most one running task in each moment. Although we could probably extend the incremental correction algorithm to multithreaded environment with parallel computations, this version only works task by task.

After the execution of `activated` task is started, we attempt to do the best in order to complete its mission, i.e. to find repair for a sequence of nodes, thus to find shortest paths in the constructed repairing multigraph. But we need to manage this effort and therefore only the starting intent has no limitations. All other tasks have only one opportunity to request the further steps of nested intents computation. This stands for the explanation of the last boolean parameter in the procedure invoked in the loop.

Algorithm 7: incrementalCorrectionRoutine

```
   Input  : Data tree 𝒯, single type tree grammar 𝒢.
   Output: Repair ℛ_{ℐ•} for correction intent ℐ•.
1  ℳ = (Suspended, Blocked, Signalled, Activated) ← (∅, ∅, ∅, ∅);
2  cacheTask ← ∅;
3  createCorrectionTask(ℐ•);
4  activateCorrectionTask(⊥, ℐ•);
5  while ∃𝒦_ℐ ∈ ℳ.Signalled ∪ ℳ.Activated do
6      switch 𝒦_ℐ.state do
7          case signalled
8              creative ← (ℐ = ℐ•);
9              processCorrectionStep(𝒯, 𝒢, 𝒦_ℐ, creative);
10         case activated
11             processCorrectionStep(𝒯, 𝒢, 𝒦_ℐ, true);
12 ℛ(ℐ•) ← cacheRepair(signature(ℐ•));
13 return ℛ_{ℐ•};
```

Incremental Correction Step

The main principle of the incremental algorithm is to compute progressively into the depth. This means that at each level of the nested intents we first need to follow the idea of the shortest paths searching, but concurrently we need to suppress excessive activity to really achieve the comfortable behaviour inside the top intent. Thus whenever the particular intent task is requested for the next headway, we need to create a mechanism for its well-timed interruption.

Each activated intent except the top one is able to process without limits until it first requests computation of some not yet fully computed tasks. This condition seems rather strict, but after there are already some intents finished and their repairs cached, there is a chance to go through larger parts of the multigraph.

Suppose that we have selected the most perspective vertex from the set of reached vertices. The dynamic version of the correction algorithm does the same, but now we cannot automatically declare this vertex as fully defined, since we only have the estimation of costs of edges leading to this vertex. Even though it now seems the given node is the best one, further computations may reverse this state. Thus in order to declare the given candidate vertex as final, there must be at least one fully evaluated ingoing edge to this vertex from the set of reached edges and all other not yet fully evaluated edges must have strictly greater cost estimation.

Only under these conditions we can declare the given perspective vertex as final, terminate its processing and explore other probably not yet explored

vertices beyond it. If we cannot close the given candidate, we need to refine all its ingoing edges that have not yet been fully evaluated. We therefore request computation of these tasks and then we end in order to wait until these nested intents bring progress to edges in constructed multigraph.

Algorithm 8: processCorrectionStep

Input : Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, correction task $\mathcal{K}_\mathcal{I}$ = ($\mathcal{I}$, $\mathcal{R}(\mathcal{I})$, *state*, *stamp*, $V_{reached}$, $E_{reached}$, $E_{delayed}$, V_T, $c_{reached}$, c_{fixed}, W_A, W_D), boolean *creative*.

```
1  foreach e ∈ K_I.E_reached do
2      renewEdgeRatings(e, K_I.stamp);
3  foreach v ∈ K_I.V_reached do
4      renewVertexRatings(R(I), K_I, v);
5  while K_I.V_reached ≠ ∅ do
6      v_1 ← selectBestCandidate(R(I), K_I.V_reached);
7      K_I.c_reached ← pathCost(v_1);
8      if [(K_I.V_T = {v_1}) or (I.f_T = aggregated and v_1 ∈ K_I.V_T)] and
       [K_I.c_fixed is not defined] then K_I.c_fixed ← K_I.c_reached;
9      if K_I.c_fixed is defined and K_I.c_fixed < K_I.c_reached then
10         K_I.V_reached ← ∅; break
11     if resolveVertexCompleteness(R(I), K_I, v_1) then
12         processCompleteVertex(T, G, I, R(I), v_1);
13     else
14         if creative then
15             processIncompleteVertex(K_I, v_1);
16         break
17 if K_I.V_reached = ∅ and K_I.E_delayed ≠ ∅ and creative then
18     processDelayedEdges(R(I), K_I);
19 renewRepairRatings(I, R(I), K_I);
20 K_I.stamp ← timeStamp();
21 if K_I.V_reached = ∅ and K_I.E_delayed = ∅ then
22     repair ← composeIntentRepair(I, R(I));
23     cacheRepair(signature(I)) ← repair;
24     signalCorrectionTask(K_I);
25     destroyCorrectionTask(K_I);
26 else
27     if creative then blockCorrectionTask(K_I);
28     else signalCorrectionTask(K_I);
```

Structuring Algorithm 8 into main parts, we can start with the initial renewing of the knowledge about the acquired progress in nested intents, then the main loop of processing reached vertices and, finally, the code for task processing termination. This ending essentially consists of the intent repair structure composition in case we have successfully completed the entire processing of a given task, or blocking and signalling requests in case we need to wait for nested executions or, conversely, inform ascendent tasks respectively.

Vertices Processing Routines

Now we will present procedures for processing vertices believed to be perspective candidates in the main loop of each step of task proceeding. First, we will put attention to Algorithm 9 describing the processing of vertices that can be declared as final, thus satisfying the outlined conditions of existence of some fully evaluated ingoing edge having the cost better than all other estimations from only partially evaluated edges.

Algorithm 9: processCompleteVertex

Input: Data tree $\mathcal{T}$, single type tree grammar $\mathcal{G}$, correction intent $\mathcal{I}$, partial multigraph $\mathcal{R}(\mathcal{I})$, task $\mathcal{K}_{\mathcal{I}}$, vertex $v_1 = (s_1, q_1)$

```
1  Remove v1 from K_I.V_reached and K_I.V_T;
2  Remove ∀e' = (v'1, v'2, I', R', c'), v'2 = v1 from K_I.E_reached;
3  foreach I' ∈ intents(I), (I'.s_I, I'.q_I) = v1 do
4      s' ← signature(I');
5      if cacheTask(s') is not defined and cacheRepair(s') is not defined
       then createCorrectionTask(I');
6      foreach q'_E ∈ I'.Q_E do
7          v2 ← (I'.s_E, q'_E); e ← (v1, v2, I', ⊥, ⊥);
8          if v2 ∉ R(I).V_R then
9              Add v2 into R(I).V_R and K_I.V_reached;
10         if v2 ∈ K_I.V_reached then
11             Add correcting edge e into R(I).E_R and K_I.E_reached;
12         else
13             Add correcting edge e into R(I).E_R and K_I.E_delayed;
14         renewEdgeRatings(e, 0);
15         renewVertexRatings(R(I), K_I, v2);
```

Similarly to the original dynamic algorithm we need to inspect all outgoing edges and newly discovered vertices terminating these edges.

In Algorithm 10 we present the processing of vertices that cannot be closed, since we have not yet computed enough to be sure we can do it. In order to

achieve better estimations, we request further processing of all edges ingoing to the vertex currently declared as the perspective one.

Algorithm 10: processIncompleteVertex

Input: Correction task $\mathcal{K}_{\mathcal{I}}$, vertex $v_1 = (s_1, q_1)$.

```
1 foreach e = (v'_0, v'_1, I', R', c') ∈ K_I.E_reached, v'_1 = v_1 do
2     if e.R' = ⊥ then
3         activateCorrectionTask(K_I, I');
```

During the process of searching for the shortest paths we sometimes come across edges leading from the vertex being closed to some other already closed one. It is easy to see that we do not need to evaluate these edges immediately, because they cannot influence the traversal itself. However, we need to evaluate them at least partially in order to eliminate the possibility they can be involved in other shortest paths. In our correction algorithm we have decided to postpone their computation until we know that some superior task really needs to terminate such intent. In Algorithm 11 we present the procedure of deferred processing of these edges.

Algorithm 11: processDelayedEdges

Input: Partial multigraph $\mathcal{R}(\mathcal{I})$, task $\mathcal{K}_{\mathcal{I}}$.

```
1  foreach e = (v_1, v_2, I', R, c) ∈ K_I.E_delayed do
2      renewEdgeRatings(e, K_I.stamp);
3      c' ← pathCost(v_1) + e.c;
4      if c' > pathCost(v_2) then
5          Remove e from K_I.E_delayed;
6      else
7          if e.R' = ⊥ then
8              activateCorrectionTask(K_I, I');
9          else
10             pathPrev(v_2) ← pathPrev(v_2) ∪ {v_1};
11             Remove e from K_I.E_delayed;
```

Selection of Perspective Vertices

The candidate vertex regarded as the best vertex to work on is selected in a way presented in Algorithm 12. Similarly to basic algorithms we again only select such vertex from the set of reached vertices that have the lowest estimation of cost. Although the presented function does not contain it, we can extend

this selection criterion in a way that we prefer those vertices with lowest cost, which have also some already closed ingoing edge. This choice increases the probability of the candidate termination.

Algorithm 12: selectBestCandidate

Input : Partial multigraph $\mathcal{R}(\mathcal{I})$, set of vertices $V \subseteq \mathcal{R}(\mathcal{I}).V_R$.
Return: Selected perspective vertex v_m.

1 $m \leftarrow min_{v \in V}[\mathcal{R}(\mathcal{I}).pathCost(v)]$;
2 $v_m = (s, q)$ for some $v_m \in V$, $\mathcal{R}(\mathcal{I}).pathCost(v_m) = m$;
3 **return** v_m;

Algorithm 13 introduces the evaluation of condition the perspective candidate vertex should satisfy in order it can be treated as fully completed. Note that an ordinary vertex in the exploration multigraph usually has more ingoing edges, thus there is a chance that some edges need not to be fully evaluated and only the lowest ones win and cause earlier progress in the shortest path searching goal.

Algorithm 13: resolveVertexCompleteness

Input : Partial multigraph $\mathcal{R}(\mathcal{I})$, task $\mathcal{K}_\mathcal{I}$, vertex $v_2 = (s_2, q_2)$.
Return: `true` for completed vertex, `false` otherwise.

1 $InputEdges \leftarrow \{e \mid e \in \mathcal{K}_\mathcal{I}.E_{reached}$ and $e.v_2 = v_2\}$;
2 $ClosedEdges \leftarrow \{e \mid e \in InputEdges$ and $e.R \neq \bot\}$;
3 $OpenedEdges \leftarrow \{e \mid e \in InputEdges$ and $e.R = \bot\}$;
4 **if** $InputEdges = \emptyset$ **then return** `true`;
5 **if** $\exists e = (v_1, v_2, \mathcal{I}', R, c) \in ClosedEdges$ **then**
6     $m \leftarrow \min_{e \in ClosedEdges}(pathCost(e.v_1) + e.c)$;
7   Let e_0 be any edge from $ClosedEdges$ corresponding to m;
8   **if** $\forall e \in OpenedEdges,\ pathCost(e.v_1) + e.c > m$ **then return** `true`; **else return** `false`;
9 **else return** `false`;

Updates of Constructed Multigraph

Although the original purpose of Algorithm 14 was to renew values of estimated costs in all reached edges that have showed some progress during the given intent task was waiting blocked, we can also use it to correctly resolve values for edges newly reached during the processing of candidate vertices.

Algorithm 14: renewEdgeRatings

Input: Edge $e = (v'_1, v'_2, \mathcal{I}', R', c') \in \mathcal{R}(\mathcal{I}).E_R$, $v'_2 = (s'_2, q'_2)$, timestamp *stamp* of the last performed update of e.

1 **if** $e.R' = \perp$ **then**
2   **if** `cacheTask(`$signature(\mathcal{I}')$`)` *is defined* **then**
3         $\mathcal{K}' \leftarrow$ `cacheTask(`$signature(\mathcal{I}')$`);`
4   **if** $\mathcal{K}'.stamp > stamp$ **then**
5   **if** $\mathcal{I}'.f_T =$ `aggregated` **then**
6                 $e.c' \leftarrow \mathcal{K}'.\mathcal{R}(\mathcal{I}').termCost$;
7   **else** $\mathcal{I}'.f_T =$ `separated`
8                 $e.c' \leftarrow \mathcal{K}'.\mathcal{R}(\mathcal{I}').termCost(q'_2)$;
9   **else**
10         $e.R' \leftarrow$ `cacheRepair(`$signature(\mathcal{I}')$`);`
11   **if** $\mathcal{I}'.f_T =$ `aggregated` **then** $e.c' \leftarrow cost(e.R')$;
12   **else** $e.c' \leftarrow cost(e.R', q'_2)$; $\mathcal{I}'.f_T =$ `separated`

Whereas the purpose of the previous procedure was only to update records stored in the structure of the correction edge, Algorithm 15 introduces the mechanism of updating the *pathPrev* and *pathCost* functions of the constructed repairing multigraph.

Algorithm 15: renewVertexRatings

Input: Partial multigraph $\mathcal{R}(\mathcal{I}) = (V_R, E_R, pathPrev, pathCost, termCost)$, task $\mathcal{K}_\mathcal{I}$, vertex v to be updated.

1 **if** $v \in \mathcal{K}_\mathcal{I}.V_{reached}$ **then**
2     $pathCost(v) \leftarrow \infty$; $pathPrev(v) \leftarrow \{\}$;
3   **foreach** $e = (v_1, v_2, \mathcal{I}, R, c) \in \mathcal{R}(\mathcal{I}).E_R$, $v_2 = v$ **do**
4         $c' \leftarrow pathCost(v_1) + c$;
5   **if** $pathCost(v) = c'$ **then**
6             $pathPrev(v) \leftarrow pathPrev(v) \cup \{v_1\}$;
7   **if** $pathCost(v) > c'$ **then**
8             $pathCost(v) \leftarrow c'$; $pathPrev(v) \leftarrow \{v_1\}$;
9 **else**
10   **foreach** $e = (v_1, v_2, \mathcal{I}, R, c) \in \mathcal{R}(\mathcal{I}).E_R$, $v_2 = v$ **do**
11         $c' \leftarrow pathCost(v_1) + c$;
12   **if** $pathCost(v) = c'$ **then**
13             $pathPrev(v) \leftarrow pathPrev(v) \cup \{v_1\}$;

The mechanism of updating vertices is different to basic correction algorithms, since we need for each vertex inspect all its ingoing edges. In Algorithm 16 we renew the record about estimated cost for the entire intent repair we are attempting to construct in a given correction task.

Algorithm 16: renewRepairRatings

Input: Correction intent $\mathcal{I}$, partial repairing multigraph $\mathcal{R}(\mathcal{I}) = (V_R, E_R, \mathit{pathPrev}, \mathit{pathCost}, \mathit{termCost})$ and task $\mathcal{K}_\mathcal{I}$.

```
1 if I.f_T = aggregated then
2     if K_I.c_fixed is defined then termCost ← K_I.c_fixed;
3     else termCost ← K_I.c_reached;
4 else I.f_T = separated
5     foreach q_T ∈ I.Q_T do
6         v_T ← (n, q_T);
7         if v_T ∈ K_I.V_T then termCost(q_T) ← K_I.c_reached;
8         else termCost(q_T) ← pathCost(v_T);
```

Tasks Managing Procedures

Finally, we will introduce five auxiliary procedures serving for management of tasks processing. Their precise introduction is relatively important, since there are some crucial aspects, which may not be obvious from the first point of view.

Algorithm 17: createCorrectionTask

Input : Correction intent $\mathcal{I}$.

```
1  v_S ← (0, q_S);
2  R(I) = (V_R, E_R, pathPrev, pathCost, termCost);
3  R(I) ← ({v_S}, ∅, ∅, ∅, ⊥);
4  pathCost(v_S) ← 0; pathPrev(v_S) ← ∅;
5  V_reached ← {v_S}; E_reached ← ∅; E_delayed ← ∅;
6  V_T ← {(n, q_T) | q_T ∈ Q_T};
7  if f_T = aggregated then termCost ← 0;
8  else foreach q_T ∈ Q_T do termCost(q_T) ← 0 ; f_T = separated
9  K_I ← (I, R(I), suspended, timeStamp(), V_reached, E_reached, E_delayed,
   V_T, 0, ⊥, ∅, ∅);
10 M.Suspended ← M.Suspended ∪ {K_I};
11 cacheTask(signature(I)) ← K_I;
```

In Algorithm 17 we start with a new task creation routine. First a newly created task is not automatically activated. This is because new tasks are usually constructed during the closure of perspective vertices when exploring new parts of the multigraph, but in this step we do not invoke their computation. Next, we need to initialize all task internal records.

When the correction routine processes the candidate vertex which needs to be refined before its termination can be accounted, we need to request the execution of all not yet fully computed ingoing edges and thus associated correction tasks. For this purpose we use Algorithm 18.

Algorithm 18: activateCorrectionTask

Input: Parental intent task $\mathcal{K}_{\mathcal{I}}$ or $\perp$, nested intent $\mathcal{I}'$.

1 $\mathcal{K}_{\mathcal{I}'} \leftarrow \texttt{cacheTask}(signature(\mathcal{I}'))$;
2 $formerState \leftarrow \mathcal{K}_{\mathcal{I}'}.state$;
3 **if** $formerState \in \{\texttt{activated}, \texttt{blocked}, \texttt{signalled}\}$ **then**
4   **if** $\mathcal{K}_{\mathcal{I}} \neq \perp$ **then** $\mathcal{K}_{\mathcal{I}'}.W_A \leftarrow \mathcal{K}_{\mathcal{I}'}.W_A \cup \{\mathcal{K}_{\mathcal{I}}\}$;
5 **if** $formerState = \texttt{suspended}$ **then**
6     $\mathcal{K}_{\mathcal{I}'}.state \leftarrow \texttt{activated}$;
7   **if** $\mathcal{K}_{\mathcal{I}} \neq \perp$ **then** $\mathcal{K}_{\mathcal{I}'}.W_A \leftarrow \{\mathcal{K}_{\mathcal{I}}\}$;
8     $\mathcal{M}.Suspended \leftarrow \mathcal{M}.Suspended \setminus \{\mathcal{K}_{\mathcal{I}'}\}$;
9     $\mathcal{M}.Activated \leftarrow \mathcal{M}.Activated \cup \{\mathcal{K}_{\mathcal{I}'}\}$;
10 **if** $\mathcal{K}_{\mathcal{I}} \neq \perp$ **then** $\mathcal{K}_{\mathcal{I}}.W_D \leftarrow \mathcal{K}_{\mathcal{I}}.W_D \cup \{\mathcal{K}_{\mathcal{I}'}\}$;

Note that we also need to handle situations, when we want to activate a task that has already been activated, or even executed. These situations can really arise, since there can be other tasks pursuing the same aims. After we have requested computations of all suitable ingoing edges, we also need to block the original task. This functionality is offered by Algorithm 19.

Algorithm 19: blockCorrectionTask

Input: Intent task $\mathcal{K}_{\mathcal{I}}$ to be blocked.

1 $formerState \leftarrow \mathcal{K}_{\mathcal{I}}.state$;
2 $\mathcal{K}_{\mathcal{I}}.state \leftarrow \texttt{blocked}$;
3 **if** $formerState = \texttt{activated}$ **then**
4     $\mathcal{M}.Activated \leftarrow \mathcal{M}.Activated \setminus \{\mathcal{K}_{\mathcal{I}}\}$;
5 **if** $formerState = \texttt{signalled}$ **then**
6     $\mathcal{M}.Signalled \leftarrow \mathcal{M}.Signalled \setminus \{\mathcal{K}_{\mathcal{I}}\}$;
7 $\mathcal{M}.Blocked \leftarrow \mathcal{M}.Blocked \cup \{\mathcal{K}_{\mathcal{I}'}\}$;

The given task remains blocked until all its required nested tasks are not executed. After this situation arises, the computation manager can execute this task once again, but this time we can only continue its processing via candidate vertices that can be immediately finalised. Another recursive requests are not allowed, otherwise we would not be able to achieve determined efficient behaviour at the top level, thus in the starting intent. On the other hand we can easily imagine minor modifications that would for example allow repeating the requesting phase for some predefined number of attempts.

Anyway at the end we need to inform all superior tasks that have originally requested the computation of this newly interrupted task. For this purpose we use procedure presented in Algorithm 20.

Algorithm 20: signalCorrectionTask

Input: Intent task $\mathcal{K}_\mathcal{I}$ to announce signal.

```
1  formerState ← K_I.state;
2  K_I.state ← suspended;
3  M.Suspended ← M.Suspended ∪ {K_I};
4  if formerState = activated then
5     M.Activated ← M.Activated \ {K_I};
6  if formerState = signalled then
7     M.Signalled ← M.Signalled \ {K_I};
8  foreach K' ∈ K_I.W_A do
9     K'.W_D ← K'.W_D \ {K_I};
10    if K'.W_D = ∅ then
11       K'.state ← signalled;
12       M.Blocked ← M.Blocked \ {K'};
13       M.Signalled ← M.Signalled ∪ {K'};
14 K_I.W_A ← ∅;
```

Finally, we will present Algorithm 21, which is called after the given task has fulfilled its mission. Such task is removed from all structures of the computation manager and then we instantly insert composed repair structure into the caching repository.

Algorithm 21: destroyCorrectionTask

Input: Intent task $\mathcal{K}_\mathcal{I}$ to be destroyed.

```
1  M.Suspended ← M.Suspended \ {K_I};
2  cacheTask(signature(I)) ← ⊥;
```

Incremental Algorithm Summary

Having introduced all component routines of the incremental correction algorithm, we can shortly discuss several questions. First the presented algorithm does not directly allow the multithreaded environment and it is not sure, whether the necessary overhead would be suppressed by acquired advantages. On the other hand we can also modify the heuristic we use for detecting perspective vertices when attempting to find shortest paths. It is easy to see that we may not follow the standard Dijkstra's algorithm too strictly, allow more latitude in the multigraph exploration and in these cases there is higher probability of the contribution of such parallel enabled algorithm.

Chapter 5

Conclusion

We have presented the framework for correcting structurally invalid XML documents with respect to single type tree grammars. Proposed repairs of elements are based on finding new suitable sequences of their child nodes that conform to corresponding content model, using efficient traversal and inspection of the state space of associated finite automaton for recognising words from a language of a regular expression.

Under any circumstances our framework is able to find all minimal repairs according to the introduced cost function assigning to each edit operation a non-negative cost. Using these elementary operations we are able to insert or delete a leaf or internal node or rename a label of an existing node.

In order to compute repairs efficiently, we have presented four correction algorithms. The most extended one is capable to incrementally evaluate only promising directions from all possible corrections.

Approach Advantages

The first contribution of our framework lies in the consideration of the single type tree grammar class. All studied approaches assumed only schemata in DTD, thus at the level of the local tree grammar class. Although not all constructs from XML Schema in fact meet criteria for single type tree grammars, this classification is widely accepted.

The advantage of our correction model is the fact that we are able to generate repairs under any circumstances. Correction algorithms in [14, 44] are able to find repairs only if the original data tree is not too far from the grammar, i.e. if it does not contain too many errors. Our algorithm does not need any form of threshold or pruning, we always find all minimal repairs.

Next, we have wider set of elementary and complex operations, through which we can attempt to correct provided potentially invalid data trees. It is easy to see that the set of proposed operations is the key factor of correction framework abilities. Finally, it is worth saying that all proposed complex update operations can be implemented using the same mechanism, we do not need to introduce, for example, different routines for inserting or deleting subtrees. All correction intents are handled in the same way.

The last proposed incremental correction algorithm is able to search for repairs efficiently, ignoring directions that do not seem perspective. And we do not only find shortest paths representing minimal repairs using this heuristic, but we are able to evaluate repairs only partially, estimate their costs and harness these estimations in the process of finding repairs for the entire data tree. Moreover, the algorithm itself never computes any local repair repeatedly, since all completed computations are cached.

Approach Disadvantages

The main disadvantage of the proposed set of edit and update operations is the fact that they do not allow local transpositions within a sequence of sibling nodes. This is a rather significant deficiency, since we can assume that local moves would probably solve typical errors in content models with fixed order of elements.

Moreover we only produce local corrections. Even though we have proposed operations for pushing or pulling groups of sibling nodes connected with an internal node insertion or deletion respectively, we are not able to suggest, for example, moves of subtrees between distant parts of a data tree.

Another disadvantage is that we consider only corrections of attributes and elements separately and do not permit their mutual switching.

Approaches Comparison

In contrast to the approach presented in [48], we have presented a wider set of update operations. Although this extension can be seen only as a minor improvement, in fact the situation is rather more complicated. It is because this work only assumed operations for a subtree insertion, deletion and a recursive repair without an option to rename a label. All these operations can be implemented very efficiently and thus authors do not need to deal with related problems.

Comparing to the approach presented in [14], we have wider set of operations too, but, first of all, we are able to return repairs under all conditions, not only if the original data tree has small number of errors. In that case this algorithm is able to return at least some repairs, it returns all minimal repairs and also some other non-minimal. However, not all non-minimal. Our correction algorithm returns always all minimal but only minimal repairs.

Our correction algorithm is also able to propose new suitable labels for parental nodes with invalid sequence of their child nodes, which the discussed approach is only able under certain conditions. Finally we have presented much more efficient correction algorithm, which not only caches computed repairs, but also behaves lazily.

Performance Experiments

Although we have introduced four different correction algorithms following the same formal model, for practical applications are suitable only the caching algorithm and especially the incremental one.

Having a simple recursive tree grammar and trees with even only a root node, the naive correction algorithm may not be able to compute any repair, terminating unexpectedly on the lack of memory. This behaviour is caused by an enormously high number of pointlessly repeated computations. Even though the dynamic algorithm directly constructs only the shortest paths, the same problem remains.

Whilst these basic algorithms have problems with units of nodes in a data tree, the incremental algorithm is able to process under the same tree grammar hundreds of thousands nodes in a time less than a second. Not only that this algorithm does not need to evaluate all intents, it seems that only units of percents of reached edges in constructed repairing multigraphs are required to be fully evaluated.

Possible Extensions

Except dealing with already named disadvantages, we can propose several other interesting extensions that could improve the correction framework as it has been introduced in this thesis.

It has already been sketched that we can attempt to find not only all minimal repairs, but also some non-minimal or even all non-minimal repairs within a certain threshold. This extension would first need to slightly extend the model of multigraph edges in order to enable representation of repairs with different costs. However, we would consequently need to significantly modify the concept of finding paths. More paths bring more intents evaluation and, moreover, we can lose the advantages based on introduced efficiency concepts.

Next, we can propose new edit and associated update operations. For example, with the connection to presented existing implementations correcting well-formedness of both HTML and XML documents, there would obviously be interesting an operation dealing with potentially badly closed or opened elements that originally had omitted closing or opening tags respectively. The mentioned correction algorithm may not propose right locations for their placing and thus we can attempt to rectify these misses. An operation capable to lift up a prefix or postfix of a sequence of sibling nodes one level up before or behind their parent respectively and next an operation capable to press down a sequence of sibling nodes preceding or following a given node under this node as a prefix or postfix respectively seem sufficient for solving these situations.

Although we have ended with a presentation of the incremental correction algorithm, there is an option of its further improvement based on enabling parallel computations by more threads concurrently. On this account we can probably relax the proposed heuristic for finding shortest paths and permit more requested computations at once, not only working on edges ingoing to

the most perspective vertex in a multigraph.

Another extension can be introduced in a way how found repairs are presented to the user. Passing over an already outlined option of interactivity with the user having the veto for choosing the best local repairs, we can separate repairs for attributes and elements. First we can establish the final structure of a data tree and later on we can walk through nodes and repair attributes.

Future Work

The main direction of our future work around introduced correction framework is its integration into the Analyzer project [50, 3], robust framework designed for massive parallel analyses of especially XML related documents. Although it is able to provide an environment for computing miscellaneous analyses over any types of documents, we will focus on the problem of processing structurally invalid XML documents.

We will encapsulate the proposed correction algorithm into a plugin and then use it for computing characteristics of real-world XML documents and their degree and forms of invalidity.

Appendix A

Content of CD

The enclosed CD contains:

- PDF version of this thesis.
- Source files of the prototype implementation.
- Generated JavaDoc documentation.
- Performed experiments data.

Bibliography

[1] Abiteboul, Serge and Segoufin, Luc and Vianu, Victor. Representing and Querying XML with Incomplete Information. *ACM Trans. Database Syst.*, 31(1):208–254, 2006.

[2] Alur, Rajeev and Madhusudan, P. Visibly Pushdown Languages. In *STOC '04: Proceedings of the 36th annual ACM symposium on Theory of computing*, pages 202–211, New York, NY, USA, 2004. ACM.

[3] Analyzer 1.0. `http://urtax.ms.mff.cuni.cz/anaxml/`.

[4] Arenas, Marcelo and Fan, Wenfei and Libkin, Leonid. On Verifying Consistency of XML Specifications. In *PODS '02: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 259–270, New York, NY, USA, 2002. ACM.

[5] Barbosa, Denilson and Mendelzon, Alberto O. and Libkin, Leonid and Mignet, Laurent and Arenas, Marcelo. Efficient Incremental Validation of XML Documents. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*, pages 671–682. IEEE Computer Society, 30 2004.

[6] Biron, P. V. and Malhotra, A. XML Schema Part 2: Datatypes (Second Edition), 2004. `http://www.w3.org/TR/xmlschema-2/`.

[7] Boag, Scott and Chamberlin, Don and Fernandez, Mary F. and Florescu, Daniela and Robie, Jonathan and Simeon, Jerome. XQuery 1.0: An XML Query Language, 2007. `http://www.w3.org/TR/xquery/`.

[8] Boobna, Utsav and Rougemont, Michel de. Prototype Implementation of Correctors for XML Data. `http://www.lri.fr/ mdr/xml/`.

[9] Boobna, Utsav and Rougemont, Michel de. Correctors for XML Data. In *Database and XML Technologies*, volume 3186/2004 of *Lecture Notes in Computer Science*, pages 69–96. Springer-Verlag Berlin Heidelberg, 2004.

[10] Boobna, Utsav and Rougemont, Michel de. Correctors for Ranking XML Documents, 2008.

[11] Bouchou, B. and Duarte, D. and Halfeld, M. and Alves, Ferrari and Laurent, D. Extending Tree Automata to Model XML Validation under Element and Attribute Constraints. In *ICEIS 2003, Proceedings of the 5th International Conference on Enterprise Information Systems*, pages 184–190, 2003.

[12] Bouchou, Beatrice and Alves, Mirian Halfeld Ferrari. Updates and Incremental Validation of XML Documents. In *Database Programming Languages*, volume 2921/2004 of *Lecture Notes in Computer Science*, pages 139–140. Springer-Verlag Berlin Heidelberg, 2004.

[13] Bouchou, Beatrice and Alves, Mirian Halfeld Ferrari and Musicante, Martin A. Tree Automata to Verify XML Key Constraints. In *WebDB: International Workshop on Web and Databases*, pages 37–42, 2003.

[14] Bouchou, Beatrice and Cheriat, Ahmed and Alves, Mirian Halfeld Ferrari and Savary, Agata. Integrating Correction into Incremental Validation. In *BDA*, 2006.

[15] Bouchou, Beatrice and Cheriat, Ahmed and Ferrari, Mirian Halfeld and Musicante, Martin. Incremental Constraint Validation of XML Documents under Multiple Updates. Technical report, Université François-Rabelais de Tours, 2006.

[16] Bouchou, Beatrice and Cheriat, Ahmed and Ferrari, Myrian Halfeld and Savary, Agata. XML Document Correction: Incremental Approach Activated by Schema Validation. *Database Engineering and Applications Symposium, International*, pages 228–238, 2006.

[17] Bray, Tim and Paoli, Jean and Sperberg-McQueen, C. M. and Maler, Eve and Yergeau, Francois. Extensible Markup Language (XML) 1.0 (Fifth Edition), 2008. `http://www.w3.org/XML/`.

[18] Bray, Tim and Paoli, Jean and Sperberg-McQueen, C. M. and Maler, Eve and Yergeau, Francois and Cowan, John. Extensible Markup Language (XML) 1.1 (Second Edition), 2006. `http://www.w3.org/XML/`.

[19] Cheriat, Ahmed and Savary, Agata and Bouchou, Beatrice and Alves, Mirian Halfeld Ferrari. Incremental String Correction: Towards Correction of XML Documents. In *Stringology: Proceedings of the Prague Stringology Conference*, pages 201–215. Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005.

[20] Chitic, Cristiana and Rosu, Daniela. On Validation of XML Streams using Finite State Machines. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 85–90, New York, NY, USA, 2004. ACM.

[21] Clark, James. XSL Transformations (XSLT) Version 1.0, 1999. `http://www.w3.org/TR/xslt`.

[22] Clark, James and DeRose, Steve. XML Path Language (XPath) Version 1.0, 1999. `http://www.w3.org/TR/xpath/`.

[23] Cobena, Gregory and Abiteboul, Serge and Marian, Amelie. Detecting Changes in XML Documents. In *ICDE '02: Proceedings of the 18th International Conference on Data Engineering*, pages 41–52, Washington, DC, USA, 2002. IEEE Computer Society.

[24] Dijkstra, E. W. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[25] Document Object Model (DOM). `http://www.w3.org/DOM/`.

[26] Fallside, David C. and Walmsley, Priscilla. XML Schema Part 0: Primer (Second Edition), 2004. `http://www.w3.org/TR/xmlschema-0/`.

[27] Flesca, Sergio and Furfaro, Filippo and Greco, Sergio and Zumpano, Ester. Repairs and Consistent Answers for XML Data with Functional Dependencies. In *Database and XML Technologies*, volume 2824/2003 of *Lecture Notes in Computer Science*, pages 238–253. Springer-Verlag Berlin Heidelberg, 2003.

[28] Flesca, Sergio and Furfaro, Filippo and Greco, Sergio and Zumpano, Ester. Querying and Repairing Inconsistent XML Data. In *WISE '05: Proceedings of the 6th International Conference on Web Information Systems Engineering*, volume 3806/2005 of *Lecture Notes in Computer Science*, pages 175–188. Springer-Verlag Berlin Heidelberg, 2005.

[29] HTML 4.01 Specification, 1999. `http://www.w3.org/TR/html401/`.

[30] HTML Tidy. `http://tidy.sourceforge.net/`.

[31] HtmlCleaner 2.1. `http://htmlcleaner.sourceforge.net/`.

[32] Java 6 Standard Edition. `http://java.sun.com/javase/6/`.

[33] Klarlund, Nils and Schwentick, Thomas and Suciu, Dan. XML: Model, Schemas, Types, Logics, and Queries. In *In Logics for Emerging Applications of Databases*, pages 1–41. Springer, 2003.

[34] Kumar, Viraj and Madhusudan, P. and Viswanathan, Mahesh. Visibly Pushdown Automata for Streaming XML. In *WWW '07: Proceedings of the 16th International Conference on World Wide Web*, pages 1053–1062, New York, NY, USA, 2007. ACM.

[35] Lu, Shiyong and Sun, Yezhou and Atay, Mustafa and Fotouhi, Farshad. A Sufficient and Necessary Condition for the Consistency of XML DTDs. In *ER (Workshops)*, volume 2814 of *Lecture Notes in Computer Science*, pages 250–260, 2003.

[36] Megginson, David. SAX 2.0. `http://www.saxproject.org/`.

[37] Mlynkova, Irena and Toman, Kamil and Pokorny, Jaroslav. Statistical Analysis of Real XML Data Collections. In *Proceedings of the 13th International Conference on Management of Data*, 2006.

[38] Murata, Makoto and Lee, Dongwon and Mani, Murali and Kawaguchi, Kohsuke. Taxonomy of XML Schema Languages using Formal Language Theory. *ACM Trans. Internet Technol.*, 5(4):660–704, 2005.

[39] NekoHTML Parser 1.9.14. `http://nekohtml.sourceforge.net/`.

[40] Neven, Frank. Automata Theory for XML Researchers. *SIGMOD Rec.*, 31(3):39–46, 2002.

[41] Ng, Patrick K. L. and Ng, Vincent T. Y. Structural Similarity between XML Documents and DTDs. In *Computational Science — ICCS 2003*, volume 2659/2003 of *Lecture Notes in Computer Science*, pages 412–421. Springer-Verlag Berlin Heidelberg, 2003.

[42] Nierman, Andrew and Jagadish, H. V. Evaluating Structural Similarity in XML Documents. In *WebDB '02: Proceedings of the 5th International Workshop on the Web and Databases*, pages 61–66, 2002.

[43] RELAX NG. `http://www.relaxng.org/`.

[44] Rougemont, Michel de. The Correction of XML Data. In *ISIP '03: First Franco-Japanese Workshop on Information Search, Integration and Personalization*, 2003.

[45] Schematron. `http://www.schematron.com/`.

[46] Schewe, Klaus Dieter and Thalheim, Bernhard and Wang, Qing. Validation of Streaming XML Documents with Abstract State Machines. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 147–153, New York, NY, USA, 2008. ACM.

[47] Segoufin, Luc and Vianu, Victor. Validating Streaming XML Documents. In *PODS '02: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 53–64, New York, NY, USA, 2002. ACM.

[48] Slawomir Staworko, Jan Chomicky. Validity-Sensitive Querying of XML Databases. In *Current Trends in Database Technology – EDBT 2006, 2nd International Workshop on Database Technologies for Handling XML Information on the Web (DataX'06)*, volume 4254/2006 of *Lecture Notes in Computer Science*, pages 164–177. Springer-Verlag Berlin Heidelberg, 2006.

[49] Suzuki, Nobutaka. Finding an Optimum Edit Script between an XML Document and a DTD. In *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*, pages 647–653, New York, NY, USA, 2005. ACM.

[50] Svoboda, Martin and Starka, Jakub and Sochna, Jan and Schejbal, Jiri and Mlynkova, Irena. Analyzer: A Framework for File Analysis. In *BenchmarX '10: Proceedings of the 2nd International Workshop on Benchmarking of Database Management Systems and Data-Oriented Web Technologies of DASFAA '10: 15th International Conference on Database Systems for Advanced Applications*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[51] Tan, Zijing and Wang, Wei and Xu, Jian Jun and Shi, Baile. Repairing Inconsistent XML Documents. In *Knowledge Science, Engineering and Management*, volume 4092/2006 of *Lecture Notes in Computer Science*, pages 379–391. Springer-Verlag Berlin Heidelberg, 2006.

[52] Tan, Zijing and Zhang, Zijun and Wang, Wei and Shi, Baile. Computing Repairs for Inconsistent XML Document Using Chase. In *Advances in Data and Web Management*, volume 4505/2007 of *Lecture Notes in Computer Science*, pages 293–304. Springer-Verlag Berlin Heidelberg, 2007.

[53] Thompson, H. S. and Beech, D. and Maloney, M. and Mendelsohn, N. XML Schema Part 1: Structures (Second Edition), 2004. `http://www.w3.org/TR/xmlschema-1/`.

[54] Xing, Guangming and Malla, Chaitanya R. and Xia, Zhonghang and Venkata, Snigdha Dantala. Computing Edit Distances between an XML Document and a Schema and its Application in Document Classification. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 831–835, New York, NY, USA, 2006. ACM.