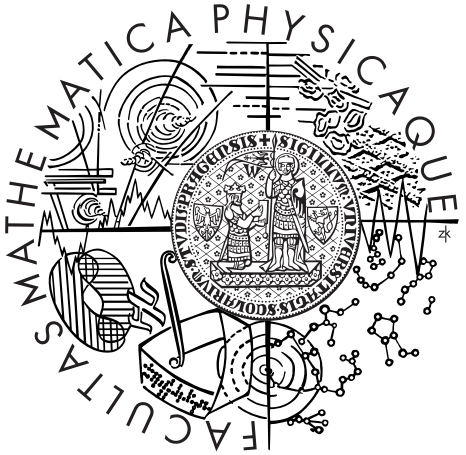
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Martin Svoboda

Correction of Invalid Trees with Respect to Regular Tree Grammars

Department of Software Engineering

Thesis supervisor:	doc. RNDr. Irena Holubová, Ph.D.
Study programme:	Computer Science
Specialization:	Software Systems

Prague 2015

First of all I would to sincerely thank my supervisor Irena Holubová for her support and guidance throughout my doctoral studies, especially to her comments and prompt responses, whenever I needed.

Next, my thanks belong to Eva Mládková for her dedication and administrative support, as well as to members, colleagues and fellow students from the XML and Web Engineering Research Group. However, achieving results of my research would not be possible even without the effort of all the anonymous reviewers.

I am also deeply obliged to Xiaofang Zhou and Shazia Sadiq from the University of Queensland, Brisbane, Australia that they offered me an unforgettable opportunity to spend six months as an intern in the Data and Knowledge Engineering Group, and hence allowed me to meet so many inspiring and friendly people such as Farzaneh Pakzad.

Nevertheless, my greatest gratitude no doubt deserve my life partner Štěpánka, my parents, and all the other family members for their support, patience as well as encouragement.

Last but not least, I would like to thank for the financial support provided by the following institutions, grants and projects: Charles University Grant Agency (4105/2011, SVV-2011-263312, SVV-2012-265312, SVV-2014-260100), Czech Science Foundation (201/09/P364, 201/09/0990, P202/10/0573), Ministry of Education, Youth and Sports of the Czech Republic (MSM0021620838), as well as European Union ICT FP7 (project 257943).

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

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

In Prague on January 31, 2015 Martin Svoboda

Title: Correction of Invalid Trees with Respect to Regular Tree Grammars

Author: RNDr. Martin Svoboda

Supervisor: doc. RNDr. Irena Holubová, Ph.D.

Department: Department of Software Engineering

Abstract: XML documents and related technologies represent one of the most widespread ways how data on the Web are maintained and interchanged. Unfortunately, many of the real-world documents contain various types of consistency issues that prevent their successful automated processing.

In this thesis we focus on the problem of the structural invalidity and its correction. In particular, having one potentially invalid XML document modeled as a tree, and a schema in DTD or XML Schema languages modeled as a regular tree grammar, our goal is to find all the minimal corrections of this tree.

The model we proposed builds on top of the recursively nested structures of correction multigraphs, where the shortest paths are being found. For this purpose we formally introduce three correction strategies with different pruning optimizations applied. According to the experiments we performed, the refinement correction strategy not only significantly outperforms all the other existing approaches, but also guarantees important characteristics the others cannot.

Keywords: XML, validity, correction, regular tree grammars, shortest paths.

Název: Oprava nevalidních stromů vůči regulárním stromovým gramatikám

Autor: RNDr. Martin Svoboda

Vedoucí: doc. RNDr. Irena Holubová, Ph.D.

Katedra: Katedra softwarového inženýrství

Abstrakt: XML dokumenty a související technologie reprezentují jednu z nejrozšířenějších cest údržby a výměny dat na Webu. Velké množství reálných dokumentů ale bohužel obsahuje nejrůznější formy nekonzistence, které brání jejich úspěšnému a automatizovanému zpracování.

V této práci se konkrétně věnujeme problému strukturální nevalidity a její korekce. Máme-li tedy jeden potenciálně nevalidní XML dokument modelovaný jako strom a současně jeho schéma v jazycích DTD nebo XML Schema modelované jako regulární stromová gramatika, naším cílem je najít všechny minimální opravy tohoto stromu.

Námi navržený model využívá rekurzivně vnořovaných struktur korekčních multigrafů, ve kterých hledáme nejkratší cesty. Za tímto účelem formálně představíme tři korekční strategie s rozdílnými úrovněmi aplikovaných optimalizací. S ohledem na provedené experimenty pak konkrétně Refinement strategie nejenom významně překonává všechny ostatní existující přístupy, ale zároveň garantuje důležité charakteristiky, které jiné přístupy zaručit nemohou.

Klíčová slova: XML, validita, opravy, regulární stromové gramatiky, nejkratší cesty.

Contents

1. Introduction

XML (*Extensible Markup Language*) [20] is a language that allows us to encode semi-structured data in textual documents that are self-describing and interpretable by both humans and machines. Notwithstanding its negative aspects such as verbosity and complexity, probably its simplicity, generality and open specification caused that this format no doubt became one of the most important and widespread ways how data on the Internet are represented.

And not only it permits us to model, structure and represent data, it is also widely used for temporary or even long-term data storing, as well as it enables knowledge sharing, communication and data interchange in general. Probably a few hundred of particular but widely used and accepted languages based on XML were successfully proposed.

They contribute not only to a basic infrastructure of the contemporary World Wide Web, or for the benefit of just a small group of specialists from the area of computer science and information technologies, but they also became involved in a wide range of practical and everyday applications – applications spanning from applied sciences to everyday activities of common people, though they might not necessarily be aware of it directly, nor they do not need to be.

To name at least a few representatives, let us begin with a protocol for exchanging XML-based messages SOAP [58], an XML serialization [39] of RDF triples [30], or OWL [101] as a language for ontologies on the Web. Then we should definitely mention XHTML [100] as a reformulation of HTML [69] used for building web pages, or RSS [70] as a web content syndication format. But there are even more applications like a language describing mathematical notations MathML [22], then SVG [31] as a representation of scalable vector graphics, or even, for example, library information systems are mostly based on traditional MARCXML [51] or METS [52] for metadata of digitized objects.

However, XML is not just a universal markup language with all its publicly shared as well as other proprietary language instances – it is a whole family of various standards and technologies.

There exist different ways of enriching the data representation as such (for example XLink [32] or XPointer [40]), different means of restricting the allowed content of documents expressed using their schemata (DTD [20] and XSD [35]), well established XML-enabled database systems allowing to store and maintain XML data (Oracle [67] or Microsoft SQL Server [56] to name at least some of them), but also native database systems (such as MarkLogic [55], Virtuoso [65] or Sedna [46]), and, last but not least, languages allowing us process, transform or query the content of XML documents (XPath [10], XQuery [15] or XSLT [26]).

Real-World Data Analyses

Our work began with a software project Analyzer [90] – an extensible framework for managing and executing statistical analyses of real-world data. It helps users to gather and automatically download documents from the Web, discover links between them, configure, compose and schedule particular analyses using the available universal and specific plugin implementations, and to browse all the

analysis results, reports and summaries in a well-arranged and unified way.

Although our main focus was initially put on XML documents, the framework itself is not bound to the family of XML technologies in the least, nor its design or architecture build on their specific aspects.

The main motivation behind *Analyzer* is to provide a user-friendly environment for performing systematic and especially repeatable analyses, so that their results could then be used to detect characteristics of the real-world data. Having identified and understood them, we may hopefully be able to optimize algorithms we use for their processing in general. Without the knowledge of such features, distribution of their values, or extent of occurrences as such, we might tend to put only too high and unnecessary attention to constructs or situations that are not important or emerge only scarcely.

In other words, it seems beneficial to propose algorithm or model optimizations with respect to the truly observed nature of the real-world data, and not just our unfounded expectations or general intentions.

Building on the existing analyses by Bex et al. [13], Mlynkova et al. [59], and Mignet et al. [57], our initial work then led us to an analysis of query statements in XPath and XQuery languages. We primarily used queries from the publicly available XML Query Use Cases [23] and parsed them in terms of the Normalized XQuery Core Grammar [34]. In particular and for example, we thoroughly studied the usage of `for` or `if` constructs, path expressions, or other concepts like individual XPath axes as well.

The entire framework, including all the particular analyses we performed, as well as a detailed description of all its components, was then summarized in [75].

Linked Data Querying

The second part of our research focused on the efficient processing of Linked Data [14], i.e. data on the Web published in a way that they follow certain recommended techniques and rules enabling their easier automated processing. Broadly speaking, these basic principles could be summarized as follows. First, real-world entities should by assigned with unique URI identifiers [11]. When trying to dereference such URI identifiers via HTTP [36], we should be provided with a usable information about these entities. And, finally, entities should be linked together, so that they can contribute to the idea of the Web of Data as a global cloud of open and interlinked data interpretable by machines.

Though other particular approaches can be used as well, the most fundamental data representation format seems to be RDF [30, 44] (*Resource Description Framework*). RDF data are based on triples $(subject, predicate, object)$, forming statements about real-world entities. Triples within a particular dataset can alternatively be viewed in a form of a graph, where vertices correspond to subjects and objects, whereas edges labeled with predicates encode the triples themselves. From the technical point of view, RDF triples can be serialized into XML [39], N-Triples [8], or Turtle [9]. And to restrain, describe, or even enrich RDF triples, schemata RDFS [21] and ontologies OWL [101] are used.

In our research we paid attention to the area of RDF data querying using SPARQL [41], certainly the most important and still evolving query language. First, in our early work [87] we identified a set of open questions that need to be

tackled in order to perform query evaluation efficiently enough. In principle, the identified challenges are related to and caused by data distribution, dynamics, and scaling. From the practical point of view, these unfold to challenges in dynamic indexing structures, management of links, data quality, as well as appropriate query processing architectures.

Having set our goals, we published a survey [88], where we described and mutually compared the existing indexing systems, all that with respect to a set of characteristics dealing with the intended deployment scopes, considered data and indexing models, as well as supported query languages and achieved pruning or other query evaluation optimizations.

To name at least some of these approaches: a quad indexing approach build on top of B^+-trees proposed by Harth and Decker [42], RDF-X stream processor for querying local RDF data by Neumann and Weikum [61], HexaStore approach with ordered nested lists by Weiss et al. [102], or even more challenging proposals like BitMat by Atre et al. [6] utilizing three-dimensional matrices and their slices, GRIN index by Udrea et al. [99] based on subgraph patterns, or other like [1, 53, 98, 79, 68, 43] to make our list complete.

Understanding the internal principles and assumptions of all these approaches, and taking into account an existing study by Ding and Finin [33], we conducted yet another analysis of real-world data. This time focusing on characteristics of RDF datasets and triples as such [74], all that according to the outlined general optimization strategy.

In particular, we inspected several characteristics of terms in RDF triples, cardinalities of subject, predicate and object projections, as well as projections based on pairs of these triple components. We also inspected more complex structures like occurrences of paths of different lengths, or occurrences of subgraph patterns in a shape of ingoing or outgoing stars.

Considering all the obtained observations we finally outlined an architecture of a query system [89] that could efficiently deal with the distribution and data volatility aspects of RDF datasets, especially within a context and needs of an ongoing initiative aiming at publishing open data about procurement [48], we partly got involved too.

Processing of Graph Data

Though it might not be apparent from the first point of view, our entire research was directly or indirectly motivated and related to the general challenge of efficient processing of graph data.

XML documents themselves are basically tree structures, and when constructs like identifier references or keys are considered too, they can be modeled as even more complex graph structures than just ordinary trees. On the other hand, RDF triples represent a perfect example of a graph data model, as well as RDF datasets and links among them form graphs too.

Beside an experimental comparison of graph databases we presented in [49], or newly initiated but not yet published contributions focusing on the information propagation models and algorithms within the scope of social networks, processing of incorrect XML documents is the last topic to which we devoted our main research interest.

And right to this topic we would like to dedicate the content of this entire thesis. In particular, to the problem of the correction of invalid XML documents.

Consistency Issues of XML Documents

The variety of available XML technologies and standards is wide, as is the usage of XML documents on the Web in practice. Unfortunately, a high number of these documents and a distributed nature of the Web itself, as well as changes in time are only some of the reasons why we can often come across with XML documents that suffer from various types of inconsistencies or errors, as not only Mlynkova et al. [59] observed.

Probably the most essential category of these errors is represented by documents that are not well-formed at all. This basically means that a given document contains syntactic issues that prevent us from parsing and representing it using a tree structure comprising of mutually nested elements. It means representing its content in a way natural to the data model XML documents are based on.

Having a well-formed document, we can focus on its structure, the allowed nesting of elements, as well as on several other basic restrictions – constraints the given document must abide by in order to be declared as valid. Although we have already mentioned two most common schema languages – DTD and XSD (XML Schema) – that both allow us to describe such restrictions, there are other approaches available as well. And they all offer us with slightly or even very different constructs we can exploit for this purpose.

Whereas Schematron [71] is based on rules that allow us to validate assertions about the presence or absence of particular patterns in XML documents, Relax NG [27] (*REgular LAnguage for XML Next Generation*) permits us to specify patterns of the allowed structure the given documents should conform to.

And even when XML documents are valid, we can still inspect further constraints posed on the data values they contain. In particular, we can study functional dependencies, keys or complex multivalued dependencies as well.

To sum up, all these inconsistencies may generally cause that processing of XML documents becomes more difficult, requires special attention, or is not possible at all. In order to tackle this problem, we could decide to make the processing algorithms more robust, so that they become able to get over such obstacles whenever possible.

On the other hand, a more general, reusable and promising solution seems to be to find corrections of the documents themselves.

Correction of XML Documents in General

First of all, let us have a look on the existing approaches that aim at correcting XML documents, that provide us with necessary formal concepts, or that at least discuss ideas related to our goals.

Context-free grammars [45], or their alternatives such as balanced context-free grammars [12] or extended context-free grammars [47], are basic formal concepts that can help us when parsing and processing XML documents. They view them in terms of ordinary sequences of symbols, i.e. words, and so they are when focusing on low-level aspects such as the well-formedness. Utilizing the corresponding

push-down automata [4], approaches by Staworko et al. [78] or Thomo et al. [97] can then start to tackle the correction problem at this level.

On the other hand, when dealing with the structural validity of XML documents, it seems to be a better idea to view such documents directly as trees, and so rather to use formal concepts of regular tree grammars [60, 62] or regular hedge grammars [91], together with their corresponding tree automata.

How the algorithms allowing us to verify such validity should be designed, is the first question that naturally arises now. Though the basic approaches are straightforward [60], validation under specific conditions such as, for example, in case of streaming XML documents [73, 25, 50, 72], or incremental validation [7, 17, 2] were studied as well.

The validation itself, unfortunately, cannot provide us with direct answers how invalid documents should be corrected in order they become valid, nor it can even suggest or estimate the extent of such corrections. In other words, one thing is to identify a particular inconsistent location in a document that is breaching the validity requirements, or even to provide technical reasons why this happens, but finding the suitable correction is yet another challenge on its own.

The correction itself is tightly related and motivated by the problems of tree-to-tree and tree-to-language edit distances. While the former case is discussed, for example, by Nierman and Jagadish [64] or Cobena et al. [28], the latter one is studied by Xing et al. [103], Tekli et al. [95] or Ng and Ng [63].

Furthermore, one thing is to detect such similarity distances, another one is to find a suitable corrected XML document, and yet another to find more such corrections at a time – especially when edit scripts with particularly expressed differences to the original document should be provided as well.

Since there exists a very recent and thorough comparison presented by Amavi et al. [5], we believe that only a brief overview of the existing correction approaches will suffice for our purposes. Therefore, while Boobna and de Rougemont [16] use testers from the theory of program verification, Suzuki [80, 81] guarantees finding a set of minimal edit scripts, and Bouchou et al. [19, 18] finding a set of all the corrections within a given similarity threshold.

Though the variety of the mentioned approaches is wide, only some of them are really able to provide the required corrections. And if they do, they mutually differ in the supported edit operations, considered schemata, characteristics of discovered corrections and their number, as well as the availability of their implementation or source codes. And as a consequence, having different assumptions and objectives, these approaches then cannot be directly compared with each other from the experimental point of view.

The approach by Bouchou et al. [19, 18] we named as the last one, based on an earlier paper by Cheriat et al. [24] from the same group of authors, served us as a main inspiration for our entire work. In this approach, the corrected XML documents – sequences of child nodes in particular – are generated dynamically by exploring and traversing state spaces of finite automata recognizing the allowed content models of elements.

Despite the efficiency issues related to the repeated computations, the main problem of this approach is that the user must provide a threshold parameter in order to prune correction directions that could potentially lead to infinite

documents. More specifically, this threshold must be chosen prior to the entire correction – and when its value is set only too low, the algorithm is not able to find any solution at all. Later on, these authors extended and finalized their work in the already mentioned overview [5].

Staworko and Chomicky [77] inspired us as well by their work on querying of XML documents that are not valid. They came with an idea of restoration graphs that are able to compactly represent all the suitable correction scenarios, but focused on the querying aspect and not the correction itself.

Last but not least, to make our insight into the existing correction approaches complete even from the perspective of the last group of consistency issues of XML documents, let us also briefly mention approaches by Flesca et al. [38, 37] or Tan et al. [94, 93, 92], both dealing with integrity constraints such as functional dependencies, keys and foreign keys.

Overview of Our Correction Model

To summarize the correction problem we decided to focus on, having one potentially invalid XML document modeled as a data tree with nodes corresponding to elements, and its schema in DTD [20] or XML Schema [35] expressed using the concept of regular tree grammars, our goal is to detect whether this document abides by all the requirements the given schema poses on elements and their nesting. And if not, then to find its suitable structural corrections. In particular, all the minimal corrections, i.e. valid data trees that are as close as possible to the original data tree to be corrected.

The correction model we proposed is based on edit operations via which we are able to insert new subtrees, delete existing ones or repair them with an option of changing labels of their nodes. Given a data tree to be corrected, we process it from its root node towards leaves, always attempting to find new and allowed sequences of child nodes whenever required.

For this purpose we use an idea of recursively nested correction multigraphs which allow us to transform the problem of finding corrections to the well established problem of finding shortest paths. This means that we do not need to generate the allowed node sequences dynamically one by one, but we are able to statically represent all of them right within these correction multigraph structures. As a consequence, not only that they permit us to significantly improve the overall correction efficiency, they also permit us to store all the found corrections in a compact way, so that the users can then directly choose right one from the discovered possible corrections, without the need of explicitly enumerating all of them as sequences of the mentioned edit operations.

Whereas the correction model as such only follows the recursive structure and nesting of data tree nodes on one hand, and nesting of grammar production rules on the other, making the required correction algorithm itself efficient enough is a nontrivial issue.

Beside certain practical requirements, we also have to, for example, deal with nodes of the same labels but different contents (which refers to the problem of competing nonterminal symbols), avoid generation of potentially infinite data

trees (because of recursive production rules or content models based on regular expressions with iterations), as well as we have to appropriately deal with the exponential worst-case time complexity that would emerge in case the naive correction strategy would be blindly followed.

At the very beginning, our motivation for the correction of XML documents came from the already discussed *Analyzer* framework [90]. We first proposed a correction model [83] that took into account not only elements of XML documents, but also their attributes. We also considered a wider set of edit operations, through which we were able to handle insertions and removals of inner data tree nodes too. However, we left these operations later on in order to increase the model efficiency [76].

In our next paper [86] we then presented the caching correction algorithm, i.e. an extension of our previous algorithms in which we successfully integrated both a dynamic programming method to deal with the top-down recursive processing paradigm, as well as a horizontal pruning strategy to improve the algorithm efficiency even more.

Summarizing the entire model and the so far proposed correction algorithms in [75], we then moved toward the last of our former correction algorithms, the incremental one [85]. In this algorithm we adapted the searching for the shortest correction paths in a way that we could rely only on partially evaluated subproblems and estimations of their correction costs. Unfortunately, this algorithm did not perform well enough to fulfill our expectations – not because of this vertical pruning optimization strategy as such, but because of the particular implementation constructs we used.

Putting all our previous experience, observations and ideas together, we finally managed to completely split three mutually orthogonal aspects of our correction algorithms: *correction strategies* dealing with pruning optimizations, *execution approaches* focusing on implementation aspects, and *signature modes* allowing to avoid repeated computations.

Hence we obtained a whole set of new and improved correction algorithms, particular configurations, among which we identified the most efficient one in the very end – and the most efficient one in terms of not only our theoretical expectations, but with respect to the results of the conducted experimental evaluation as well. Moreover and beside other optimizations, we also extended the correction model itself to support not only the subclass of single-type tree grammars, but the class of all regular tree grammars as such [84].

In this thesis we are going to discuss all the details, definitions and important aspects and features of our correction model and all the correction algorithms we proposed, right in this final and most generalized form.

And though we originally started with the correction of XML documents, the resulting model and algorithms actually solve a more general one – the problem of the correction of trees with respect to regular tree grammars.

Summary of Contributions

When we put our correction model and algorithms into the context of the other existing correction approaches, we can identify the following characteristics, advantages and contributions of our solution.

- We support the full expressive power of the entire class of regular tree grammars, and not just single-type tree grammars (mostly corresponding to XSD) or even local tree grammars (corresponding to DTD) where the presence of competing nonterminal symbols is restricted.
- Regardless the particular correction strategy, execution approach or signature mode we choose, we are always able to find all the minimal corrections, specifically regardless the extent of invalidity of a document to be corrected.
- We do not require to be provided with any parameter in order to initiate the correction. In particular, we do not need any similarity threshold parameter that is not easy to determine, especially because it is not related to the extent of invalidity, and so cannot be reliably determined before the correction is commenced.
- All the corrections we discover for a provided data tree are decoded in a compact and recursively nested structure of intent repairs, which allows users to interactively choose right one from the found corrections even without the need of explicitly enumerating all the sequences of edit operations.
- Though the worst-case time complexity is polynomial with respect to the size of data trees measured in a number of nodes, when the maximal observed fan-out is sharply lower than the overall number of nodes, the algorithm tends to be nearly linear in practice.
- We implemented all the newly proposed correction algorithms and made this implementation including all the source files publicly available [82].
- Considering a wide set of characteristics, we conducted thorough experiments dealing with documents of sizes of even 100 000 nodes, i.e. sizes up to 2 orders of magnitude higher than were considered so far by the existing approaches, yet with execution times in the order of just a few seconds.

Now we shortly compare our model and algorithms to the other existing correction approaches. First, the approach by Boobna and de Rougemont [16] assumes only a restricted version of DTD, was experimentally evaluated on data trees up to 800 nodes, and though it aims at the minimal corrections, it is not able to provide them in all situations.

Next, the approach by Suzuki [81] assumes single-type tree grammars, guarantee that edit sequences for k-closest corrections will always be found, but the approach was not implemented, nor experimentally evaluated.

Finally, the most important comparison with respect to the original approach by Bouchou et al. [18]. They consider local tree grammars, find all the corrections within a given threshold, but if this threshold is set too low, no correction is found

at all. Their experiments were based on data trees of 450 nodes with execution times in the order of minutes.

Although our correction model can generally find its usage at the very practical level (for example, it could be integrated into XML-enabled editors in order to support users with correction suggestions when editing XML documents), its impact lies in the theoretical level as well.

The correction problem itself, as already noted, is tightly related to the more general problem of similarity among trees. And according to Tekli et al. [96], not only the following areas can be considered as its direct applications: exchange and integration of XML data, searching and composition of web services, querying over inconsistent data [77], classification [103] or ranking [95] of XML documents, as well as document and schema evolution [19, 18].

Thesis Outline

In Chapter 2 we first discuss all the basic concepts which we need to formally introduce our correction model and algorithms. We describe how the XML documents we are working with are modeled, how schemata are represented using regular tree grammars, as well as we recall basic notions from the areas of regular expressions and finite automata.

The purpose of Chapter 3 is to provide a thorough description of the whole correction model we proposed, with all the involved definitions and examples as well. In particular, we introduce edit operations allowing us to transform invalid data trees into valid ones, correction intents as descriptions of the recursive correction problem to be solved, and correction multigraphs with their shortest correction paths as a means to fulfill this goal. Finally, we describe how the intent repair structures are encapsulated and how they can then be translated into particular sequences of edit operations.

However, the question how the shortest correction paths should actually be found, and found efficiently enough, is the subject of Chapter 4. Starting with several basic observations and explaining the role and importance of intent signatures, we then go through all the introduced correction strategies and execution approaches, and provide their formal algorithms and details too.

Having described the entire correction model and all the correction algorithms, we move toward the experimental evaluation presented in Chapter 5. Exploiting a wide set of characteristics and execution times as one of them, we thoroughly study all the proposed algorithm configurations in order not only to verify several expectations justifying the correction model and proposed optimizations, but to identify the most efficient algorithm configuration in particular.

To conclude, in Chapter 6 we provide an overview of the most important aspects of the whole model and all the algorithms, as well as we recall their features and contributions.

The content of this thesis – it means a description of the correction model and algorithms, their features, as well their experimental evaluation – directly follows our publications [83, 76, 86, 75, 85, 84].

2. Preliminaries

The main purpose of this section is to provide and discuss an essential theoretical background and knowledge from the areas of regular expressions, finite automata and regular tree grammars – it means notions on which our entire correction model and algorithms are built.

First of all, we describe a means how we view and model XML documents that we want to work with. The core idea of this model is based on a notion of an underlying tree – it is a structure that forms a rooted tree. Next, we define XML documents themselves by enriching underlying trees by all other information we want to capture.

Finally, XML schemata are used to restrain the allowed content of XML documents. Although there are several very different ways of achieving this general aim, we only consider schemata that can be represented as regular tree grammars. In brief, they describe the only permitted way how elements in XML documents can be mutually nested into each other. For this purpose, we also need to recall some basic facts related to regular expressions and finite automata in order to describe this nesting rules and restrictions formally.

2.1 Data Trees

We start with a detailed description of our representation of XML documents. As we already outlined, we view these documents as *data trees* that are based on *underlying trees*.

2.1.1 Strings

Before we introduce them, we first recall notions of some basic operations over strings, and sets of strings. In particular, we are interested in the concatenation . and iteration * (Kleene star) operations.

Assuming that ϵ is a special symbol for the empty word such that $\epsilon \notin \Sigma$ for a particular alphabet Σ, let $u = u_1.u_2 \dots u_m$ and $v = v_1.v_2 \dots v_n$ be two words over Σ for some $m, n \in \mathbb{N}_0$, i.e. $u_i \in \Sigma$ for $\forall\, i \in \mathbb{N}$, $1 \leq i \leq m$ and $v_j \in \Sigma$ for $\forall\, j \in \mathbb{N}$, $1 \leq j \leq n$. Then we define the *concatenation* of words u and v as a word $u.v = u_1 \dots u_m.v_1 \dots v_n$.

If L_1 and L_2 are two sets of words over Σ, then $L_1.L_2 = \{v_1.v_2 \mid v_1 \in L_1 \text{ and } v_2 \in L_2\}$.

Let us now focus on the iteration operation, first over symbols, then over words. Given $S_0 = \{\epsilon\}$ and inductively $S_{i+1} = \{v.s \mid v \in S_i \text{ and } s \in \Sigma\}$ for all $i \in \mathbb{N}_0$ and for some alphabet Σ, we define $\Sigma^* = \bigcup_{i \in \mathbb{N}_0} S_i$ as the set of all finite words over Σ, i.e. the set of all possible finite words using symbols from Σ.

Let finally $L \subseteq \Sigma^*$ be a set of some words over Σ. Given $L_0 = \{\epsilon\}$ and inductively $L_{i+1} = \{v.s \mid v \in L_i \text{ and } s \in L\}$ for all $i \in \mathbb{N}_0$, 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 concatenation operation over L.

2.1.2 Underlying Trees

So, what are the already mentioned underlying trees? Assume now that $\mathbb{N}_0^*$ is a set of all finite words over the set of all non-negative integers $\mathbb{N}_0$. Having an ordinary rooted tree with ordered sibling nodes (which actually each XML document straightforwardly is), we can simply utilize the idea of prefix numbering of nodes to capture the structure of such tree. This means we can use a suitable subset of $\mathbb{N}_0^*$ to represent such tree structure formally.

Definition 2.1 (Underlying Tree)**.** *We say that a set $D \subset \mathbb{N}_0^*$ is an* underlying tree*, if both the following conditions hold:*

- *D is closed under prefixes, i.e. having a reflexive, antisymmetric and transitive binary* prefix relation $\preceq$ *(where $\forall\, u, v \in \mathbb{N}_0^*$ we define $u \preceq v$ if $u.w = v$ for some $w \in \mathbb{N}_0^*$) we require that $\forall\, u, v \in \mathbb{N}_0^*$, $u \preceq v$: $v \in D$ implies $u \in D$.*
- *$\forall\, u \in \mathbb{N}_0^*$, $\forall\, j \in \mathbb{N}_0$: if $u.j \in D$ then $\forall\, i \in \mathbb{N}_0$, $0 \leq i \leq j$, $u.i \in D$.*

The first condition enforces the expected hierarchical structure of an underlying tree (whenever a particular node is in a given underlying tree, its parent node is in this tree as well), while the second one ensures the expected continuous arrangement of sibling nodes (there are no skipped positions in a sequence of all child nodes of a given node).

We say that D is an *empty tree*, if $D = \emptyset$. Items of D are called *nodes*, ϵ is a *root* node and a set of *leaf* nodes 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 $fanOut(u)$ 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 $n = 0$. Furthermore, for the root node $\epsilon \in D$ we put $depth(\epsilon)$ $= 1$, and for $u \in D$, $u \neq \epsilon$, $u = v.i$ for some $v \in \mathbb{N}_0^*$ and $i \in \mathbb{N}_0$ we put $depth(u)$ $= depth(v) + 1$.

In other words, $fanOut(u)$ of a node $u \in D$ denotes the number of its child nodes, whereas $depth(u)$ denotes the depth of u in D. We also sometimes use a term *position* to talk about nodes when we want to emphasize their addressing effect rather than to view them as objects of underlying trees as themselves.

Finally, we define $D^{\Delta p} = \{s \mid s \in \mathbb{N}_0^*, p.s \in D\}$ for some $p \in D$ as a *subtree* of D at position p. It is worth noting that the notion of a subtree is not based only on a subset of nodes of the original tree D, but we also truncate node prefixes accordingly. As a consequence, each subtree is always a tree.

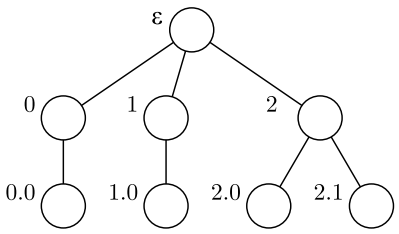


Figure 2.1: Sample underlying tree D

Example 2.1. *Assume that* $D = \{\epsilon, 0, 0.0, 1, 1.0, 2, 2.0, 2.1\}$. *It is a correctly defined underlying tree with 8 nodes from which* $LeafNodes(D) = \{0.0, 1.0, 2.0, 2.1\}$ *are its leaf nodes and the remaining ones (including the root node) are its internal nodes. For example, the root node has 3 child nodes, i.e.* $fanOut(\epsilon) = 3$, *whereas* $fanOut(2.1) = 0$ *holds for the leaf node* 2.1. *This underlying tree* D *is depicted in Figure 2.1.*

Finally, to illustrate how subtrees are formed, $D^{\Delta 2} = \{\epsilon, 0, 1\}$ *is a subtree of* D *at position* 2, *as we can see in Figure 2.2. Its nodes correspond one after another to nodes* 2, 2.0, 2.1 *of the original underlying tree* D.

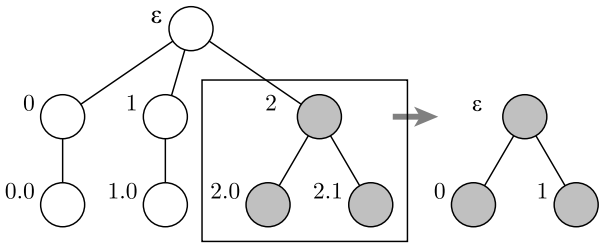


Figure 2.2: Subtree $D^{\Delta 2}$ of a sample underlying tree D

To make our insight into underlying trees complete, let us furthermore describe how their nodes (viewed as positions) can be ordered or mutually compared.

Hence, assume that $u = u_1.u_2 \dots u_m$ and $v = v_1.v_2 \dots v_n$ are two words over the alphabet $\mathbb{N}_0$ for some $m, n \in \mathbb{N}_0$, i.e. u and v are two underlying nodes. Without loss of generality, let us also suppose that $m \leq n$. Then we can define the *partial order relation* $\leq$ for words from $\mathbb{N}_0^*$ as follows. Let $c \in \mathbb{N}_0$, $0 \leq c \leq m$ be the maximal possible number such that $\forall\, k \in \mathbb{N}$, $1 \leq k \leq c$: $u_k = v_k$ (i.e. the first c symbols of both the words mutually equal to each other). Then we put $u \leq v$ if $c = m$ (which means that $u \preceq v$) or otherwise (in case of $c < m$) if $u_{c+1} \leq v_{c+1}$.

2.1.3 Data Trees

Once we have introduced the underlying trees, we can move forward and discuss the representation of XML documents themselves. First of all, we must emphasize that our correction model only focuses on elements, their allowed content and nesting. Although we have also experimented with attributes, this thesis only covers structural corrections of elements. Therefore, our model of XML documents only needs to capture these elements and (original) data values (to preserve them when presented). On the other hand, attributes, processing instructions and other constructs are fully ignored.

Before we provide a definition of *data trees*, we introduce $\mathbb{V}$ as a domain of data values and, analogously, $\mathbb{E}$ as a domain of element labels (i.e. a set of distinct element names). These domains are not bound to particular XML documents (and nor their schemata) – they are defined universally and contain really all the possible data values and element names respectively.

Definition 2.2 (Data Tree)**.** *Let D be an underlying tree, $\mathbb{V}$ the domain of data values and $\mathbb{E}$ the domain of element labels. A tuple $\mathcal{T} = (D, lab, val)$ is a* data tree*, if both the following conditions are satisfied:*

- lab *is a* labeling function $D \to \mathbb{E} \cup \{\texttt{data}\}$*, where* $\texttt{data} \notin \mathbb{E}$ *and:*
  - $DataNodes(\mathcal{T}) = \{p \in D \mid lab(p) = \texttt{data}\}$ *is a set of all* data nodes*,*
  - *if $p \in DataNodes(\mathcal{T})$ then we also require that $p \in LeafNodes(D)$ necessarily holds;*
- val *is a* value function $DataNodes(\mathcal{T}) \to \mathbb{V} \cup \{\perp\}$ *assigning values to data nodes, supposing that $\perp \notin \mathbb{V}$ represents* undefined values*.*

The idea behind data trees is simple – we use an underlying tree to form a required tree structure on one hand, and two partial functions on nodes of this underlying tree to store data values and element labels on the other. In particular, for the purpose of data values (textual contents of elements), we use data nodes, i.e. nodes having assigned a special reserved name `data`.

Analogously to underlying trees, having a particular data tree $\mathcal{T} = (D, lab, val)$ and a node $p \in D$, we define $\mathcal{T}^{\Delta p} = (D', lab', val')$ to be a data *subtree* of $\mathcal{T}$ at position p. In this case, $D' = D^{\Delta p}$ and for each function $\phi \in \{lab, val\}$: if $\phi(p.s)$ for any $s \in \mathbb{N}_0^*$ and $p.s \in D$ is defined, then $\phi'(s) = \phi(p.s)$, otherwise $\phi'(s)$ is undefined.

For the sake of clarity and throughout this entire thesis, we say that some function or item is *defined* if and only if it is assigned some meaningful value, i.e. value other than $\perp$. Otherwise it is *undefined*, i.e. it is explicitly assigned $\perp$ or it is not assigned anything at all.

Example 2.2. *Suppose we have the following XML document fragment:*

```
<?xml version="1.0"?>
<a>
   <x><c/></x>
   <d><c/></d>
   <d><c/><a/></d>
</a>
```

It corresponds to a data tree $\mathcal{T} = (D, lab, val)$ which is depicted in Figure 2.3. Its underlying tree D is exactly the same as the underlying tree in Example 2.1; element names are inscribed in nodes. To be precise, $lab = \{(\epsilon, a), (0, x), (0.0, c), (1, d), (1.0, c), (2, d), (2.0, c), (2.1, a)\}$ in case of the labeling function, and $val = \emptyset$ for the value function.

Finally, a data subtree rooted at position 2 of the data tree $\mathcal{T}$ then equals to $\mathcal{T}^{\Delta 2} = (D', lab', val')$, where $D' = D^{\Delta 2} = \{\epsilon, 0, 1\}$, $lab' = \{(\epsilon, d), (0, c), (1, a)\}$ and $val' = \emptyset$.

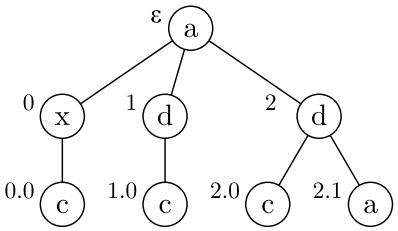


Figure 2.3: Sample data tree $\mathcal{T}$

2.2 Regular Expressions

Despite there are also other concepts of XML schema languages, the traditional ones provide constructs for restricting the allowed nesting of elements through definitions of permitted content models via regular expressions.

2.2.1 Regular Expressions

Therefore, we start with the formal definition of these regular expressions and regular languages they represent.

Definition 2.3 (Regular Expression)**.** *Let Σ be a nonempty alphabet and $S = \{\varnothing, \epsilon, |, ., ^*, (,)\}$ are special and auxiliary symbols such that $\Sigma \cap S = \emptyset$. Then we inductively define a* regular expression *r over Σ to be a word in alphabet $\Sigma \cup S$ such that:*

- *each of the following is an* atomic *regular expression:*
  - $r = \varnothing$,
  - $r = \epsilon$, *and*
  - $\forall\, x \in \Sigma$*:* $r = x$,
- *and when r_1 and r_2 are already defined regular expressions, then each of the following is a* compound *regular expression:*
  - $r = (r_1|r_2)$,
  - $r = (r_1.r_2)$, *and*
  - $r = {r_1}^*$.

The previous definition directly follows the widely accepted inductive notion of regular expressions, where $(r_1|r_2)$ corresponds to a choice, $(r_1.r_2)$ to a sequence and ${r_1}^*$ to an iteration. Having a regular expression $r = s_1 \dots s_n$ viewed as a word over $\Sigma \cup S$, we define $\mathit{symbols}(r) = \{s \mid s \in \Sigma \text{ and } \exists\, i \in \mathbb{N}_0,\ 0 \leq i \leq n,\ s_i = s\}$ to be a set of all symbols from Σ with at least one occurrence in r.

Each regular expression r over Σ represents a regular language $L(r)$ – the set of all words over Σ that conform to this regular expression. How we define it?

Definition 2.4 (Regular Language)**.** *Let r be a regular expression over an alphabet Σ. We inductively define $L(r)$ to be a* regular language *of r this way:*

- *In case r is an atomic regular expression:*
  - *if $r = \varnothing$, then $L(r) = \emptyset$,*
  - *if $r = \epsilon$, then $L(r) = \{\epsilon\}$, and*
  - *if $r = x$ for $x \in \Sigma$, then $L(r) = \{x\}$.*
- *In case r is a compound regular expression:*
  - *if $r = (r_1|r_2)$, then $L(r) = L(r_1) \cup L(r_2)$,*
  - *if $r = (r_1.r_2)$, then $L(r) = L(r_1).L(r_2)$, and*
  - *if $r = {r_1}^*$, then $L(r) = (L(r_1))^*$.*

Schemata in DTD can use two additional operators we have not mentioned yet. They are: ? (0 or 1 occurrence) and $^+$ (1 or more occurrences). Although our definition does not take them into account, obviously $r? = (r|\epsilon)$ and $r^+ = (r.r^*)$. Using an analogous idea we could also deal with constructs that are offered by XSD, like e.g. `minOccurs` and `maxOccurs`, both allowing to restrict the number of occurrences explicitly.

From now on, we also adopt a widely accepted convention of omitting parentheses to simplify regular expressions when there cannot be any confusion.

Example 2.3. *Having $\Sigma = \{A, B, C, D_A, D_B\}$ as an alphabet, we can introduce a simple regular expression $r_A = C.{D_A}^*$. It defines a language that contains all words over Σ that start with exactly one occurrence of C and continue with an arbitrary number of D_A, i.e. $L(r_A) = \{C, C.D_A, C.D_A.D_A, C.D_A.D_A.D_A, \dots\}$. Besides, $symbols(r_A) = \{C, D_A\}$.*

2.2.2 Unique Particle Attribution

We also cannot omit one important feature of regular expressions as they are assumed by both DTD and XSD languages – that is *1-unambiguity*, or, equivalently, *unique particle attribution*.

Before we can formally describe this feature, we need to introduce a notion of marked regular expressions. Having a regular expression r over an alphabet Σ, first let Σ^M be a *marked alphabet* such that $\Sigma^M = \{x_{\#i} \mid x \in \Sigma \text{ and } i \in \mathbb{N}\}$. Given a particular *marked symbol* $x_{\#i} \in \Sigma^M$, we define its *symbol projection* as $sym(x_{\#i}) = x$, i.e. *sym* is a function $\Sigma^M \to \Sigma$ that is able to fetch the original symbols from Σ.

Now, let us take the original regular expression r together with its inductive structure in mind and replace each individual occurrence of any symbol x from Σ by a marked symbol $x_{\#i}$ from Σ^M with some $i \in \mathbb{N}$, only assuring that different occurrences of the same original symbol x will be replaced by different marked symbols. For example, we can number all occurrences from left to right, starting with 1. Since this approach might be considered as one of the simplest in practice, let us choose it for the purpose of the following text.

The resulting r^M is a *marking* of r or a *marked regular expression* for r. Obviously, r^M is a regular expression over Σ^M.

Definition 2.5 (1-unambiguous Regular Expressions)**.** *Assume that r is a regular expression over an alphabet Σ and r^M is its marking. We say that regular expression r is* 1-unambiguous *if and only if for all words u, v, $w \in (\Sigma^M)^*$ and marked symbols x, $y \in \Sigma^M$, $x \neq y$: whenever $u.x.v$ and $u.y.w \in L(r^M)$, then $sym(x) \neq sym(y)$.*

The advantage of working only with 1-unambiguous regular expressions is that they enable more efficient processing, since words of their regular languages can be matched deterministically against symbols of these regular expressions while having a look ahead of just one symbol.

Example 2.4. *To illustrate how marked regular expressions can be constructed according to the left-to-right numbering approach we adopted, assume the following regular expressions: $r_1 = X.(Y|Z)$ and $r_2 = (X.Y) \mid (X.Z)$. Their markings are equal to $r_1^M = X_{\#1}.(Y_{\#2}|Z_{\#3})$ and $r_2^M = (X_{\#1}.Y_{\#2}) \mid (X_{\#3}.Z_{\#4})$.*

Despite $L(r_1) = L(r_2)$ are the same regular languages, r_1 is an example of a 1-unambiguous regular expression, but r_2 is not. The reason is that there exists two words (actually there are no words other than these two) $w_1 = X_{\#1}.Y_{\#2}$ and $w_2 = X_{\#3}.Z_{\#4}$ in $L(r_2^M)$ such that they start with different marked symbols $X_{\#1}$ and $X_{\#3}$, though they can be projected to the same original symbol of Σ, i.e. $sym(X_{\#1}) = sym(X_{\#3}) = X$.

2.3 Finite Automata

Asking whether a given word conforms to a regular expression (e.g. whether a sequence of child nodes of a given data tree node conforms to an allowed content model in our scenario) can be figured out using finite automata.

2.3.1 Deterministic Finite Automata

We first provide their general definition and then discuss a particular way how they can be constructed.

Definition 2.6 (Finite Automaton)**.** *A deterministic* finite automaton *is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where:*

- *Q is a* set of states*,*
- *Σ is an* input alphabet*,*
- *δ is a partial* transition function $Q \times \Sigma \to Q$*,*
- *$q_0 \in Q$ is an* initial state*, and*
- *$F \subseteq Q$ is a set of* accepting states*.*

If the transition function δ describes how the finite automaton works – that is in discrete steps by reading one symbol from the input word by another – we can use it to define an *extended transition function* δ^*: $Q \times \Sigma^* \to Q$ that would describe processing of the entire input words. In particular, having a word $w \in \Sigma^*$ and a state $q \in Q$ we define $\delta^*(q, w)$ as follows. When $w = \epsilon$ is an empty word,

then $\delta^*(q, w) = q$. Otherwise assuming that $w = a.v$ for some $a \in \Sigma$ and $v \in \Sigma^*$ we recursively put $\delta^*(q, w) = \delta^*(\delta(q, a), v)$ in case $\delta(q, a)$ is defined, otherwise $\delta^*(q, w) = \perp$ becomes undefined as well.

We say that a given input word $w \in \Sigma^*$ is *accepted* by a finite automaton $\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, if and only if $\delta^*(q_0, w) \in F$, i.e. when the transition function δ allows us to start at the initial automaton state q_0, follow its prescribed transitions, and terminate at one of the accepting states F while having processed the entire input word w at the same time.

Definition 2.7 (Regular Language)**.** *Given a finite automaton $\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, we define $L(\mathcal{A}) = \{w \mid w \in \Sigma^*$ for which $\delta^*(q_0, w) \in F\}$ to be a* regular language *recognized by $\mathcal{A}$, i.e. a set of all words over Σ that are accepted by $\mathcal{A}$.*

Before we move forward, we also explain the notion of reachability in the automaton state space. For each $q \in Q$ we define $reachable(q) = \{q' \mid \exists\, w \in \Sigma^*,\ q' = \delta^*(q, w)\}$ to be a set of all automaton states *reachable* from q. Furthermore, we say that a state $q' \in Q$ is *reachable* if $q' \in reachable(q_0)$, i.e. q' is reachable from the initial state q_0.

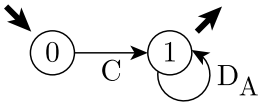


Figure 2.4: Sample finite automaton for $C.D_A{}^*$

Example 2.5. *In Figure 2.4 we can see a visualization of a sample finite automaton $\mathcal{A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$ with $Q = \{0, 1\}$ as a set of states, $\Sigma = \{C,\ D_A\}$ an input alphabet, $\delta = \{(0, C, 1),\ (1, D_A, 1)\}$ a transition function, $q_0 = 0$ the initial state, and $F = \{1\}$ a set of accepting states.*

This automaton $\mathcal{A}$ is capable of recognizing the language $L(r_A)$ of a regular expression $r_A = C.D_A{}^*$ from Example 2.3, i.e. words over Σ that start with C and end with some D_A (if any).

Though we have the general definition of finite automata, we still need to have a particular technique to be able to construct a suitable automaton $\mathcal{A}_r$ for a given regular expression r, if we want to use it for making decisions whether certain words belong to the language of such r. Formally, we need to find a way how to construct a finite automaton $\mathcal{A}_r$ such that $L(\mathcal{A}_r) = L(r)$.

Without loss of generality, we have chosen Glushkov automata [3], because they are deterministic for 1-unambiguous regular expressions, and also without ϵ-transitions (transitions that do not read a symbol from the input). Thus, they fully conform to our Definition 2.6.

On the other hand, none of these two characteristics actually affects our correction model in any way. We could easily use any other deterministic automaton as well. And we could even use any nondeterministic finite automaton (either we could slightly modify our further definitions to support them, or we could transform them to equivalent deterministic automata, which is always possible). As a consequence, our correction model is not limited just to 1-unambiguous expressions.

2.3.2 Glushkov Automata

How the Glushkov automata are then constructed? Let us first introduce a set of four auxiliary functions through which we will be able to describe such construction process precisely.

They are *empty*, $first$, $follow$ and *last*. Supposing that $RE(\Sigma)$ represents the set of all regular expressions over an alphabet Σ, they are defined as follows:

- *empty* is a function $RE(\Sigma) \to \{\texttt{false}, \texttt{true}\}$ detecting whether a regular expression $r \in RE(\Sigma)$ can lead to the empty word ϵ, i.e. $empty(r) = \texttt{true}$ if $\epsilon \in L(r)$, otherwise $empty(r) = \texttt{false}$,
- $first$ is a function $RE(\Sigma) \to \mathcal{P}(\Sigma)$ assigning to a regular expression $r \in RE(\Sigma)$ a set of all symbols that can appear as the first symbol in some word of the language $L(r)$, i.e. $first(r) = \{x \mid x \in \Sigma \text{ and } \exists\, w \in L(r) \text{ such that } w = x.v \text{ for some suitable } v \in \Sigma^*\}$,
- $follow$ is a function $RE(\Sigma) \to \mathcal{P}(\Sigma \times \Sigma)$ assigning to a regular expression $r \in RE(\Sigma)$ a set of all pairs of symbols s_1 and $s_2 \in \Sigma$ such that s_2 can appear immediately after s_1 in some word of the language $L(r)$, i.e. $follow(r) = \{(s_1, s_2) \mid s_1,\ s_2 \in \Sigma \text{ and } \exists\, w \in L(r) \text{ such that } w = u.s_1.s_2.v \text{ for some suitable } u \text{ and } v \in \Sigma^*\}$,
- *last* is a function $RE(\Sigma) \to \mathcal{P}(\Sigma)$ assigning to a regular expression $r \in RE(\Sigma)$ a set of all symbols that can appear as the last symbol in some word of the language $L(r)$, i.e. $last(r) = \{x \mid x \in \Sigma \text{ and } \exists\, w \in L(r) \text{ such that } w = u.x \text{ for some suitable } u \in \Sigma^*\}$.

All of the previously described functions refer to the existence of some appropriate words in the language $L(r)$, and so it is obvious that we actually need to find a more practical way to be really able to evaluate them when a particular regular expression $r \in RE(\Sigma)$ is provided. Luckily enough, we only need to follow the inductive definition of regular expressions:

- For an atomic regular expression r we define:
  - If $r = \varnothing$, then
    $empty(r) = \texttt{false}$, $first(r) = \emptyset$, $follow(r) = \emptyset$, and $last(r) = \emptyset$.
  - If $r = \epsilon$, then
    $empty(r) = \texttt{true}$, $first(r) = \emptyset$, $follow(r) = \emptyset$, and $last(r) = \emptyset$.
  - If $r = x$ for $x \in \Sigma$, then
    $empty(r) = \texttt{false}$, $first(r) = \{x\}$, $follow(r) = \emptyset$, and $last(r) = \{x\}$.
- And for a compound regular expression r based on r_1 and r_2:
  - If $r = (r_1|r_2)$, then
    $empty(r) = empty(r_1) \vee empty(r_2)$,
    $first(r) = first(r_1) \cup first(r_2)$,
    $follow(r) = follow(r_1) \cup follow(r_2)$, and
    $last(r) = last(r_1) \cup last(r_2)$.

– If $r = (r_1.r_2)$, then
$empty(r) = empty(r_1) \wedge empty(r_2)$,
$first(r) = first(r_1) \cup first(r_2)$ when $empty(r_1) = \texttt{true}$,
otherwise $first(r) = first(r_1)$,
$follow(r) = follow(r_1) \cup follow(r_2) \cup [last(r_1) \times first(r_2)]$, and
$last(r) = last(r_2) \cup last(r_1)$ when $empty(r_2) = \texttt{true}$,
otherwise $last(r) = last(r_2)$.

– If $r = {r_1}^*$, then
$empty(r) = \texttt{true}$,
$first(r) = first(r_1)$,
$follow(r) = follow(r_1) \cup [last(r_1) \times first(r_1)]$, and
$last(r) = last(r_1)$.

Example 2.6. *Let us now return to a regular expression $r_A = C.{D_A}^*$ from Example 2.3. Although we have not yet provided details, construction of Glushkov automata is based on marked regular expressions. So, for r we have its marking equal to $r_A^M = C_{\#1}.{D_{A\#2}}^*$.*

Now we derive values of all four introduced auxiliary functions for this regular expression r_A^M. We simply look at its inductive structure, start with its atomic subexpressions, evaluate involved operators, until we finish with r_A^M itself:

For the first atomic expression $C_{\#1}$ we derive:

$empty(C_{\#1}) = \texttt{false}$,
$first(C_{\#1}) = \{C_{\#1}\}$,
$follow(C_{\#1}) = \{\}$, *and*
$last(C_{\#1}) = \{C_{\#1}\}$.

Analogously for $D_{A\#2}$ we derive:

$empty(D_{A\#2}) = \texttt{false}$,
$first(D_{A\#2}) = \{D_{A\#2}\}$,
$follow(D_{A\#2}) = \{\}$, *and*
$last(D_{A\#2}) = \{D_{A\#2}\}$.

Now we consider the iteration operator over $D_{A\#2}$. That is:

$empty({D_{A\#2}}^*) = \texttt{true}$,
$first({D_{A\#2}}^*) = first(D_{A\#2}) = \{D_{A\#2}\}$,
$follow({D_{A\#2}}^*) = follow(D_{A\#2}) \cup [last(D_{A\#2}) \times first(D_{A\#2})] =$
$\{\} \cup [\{D_{A\#2}\} \times \{D_{A\#2}\}] = \{(D_{A\#2}, D_{A\#2})\}$, *and*
$last({D_{A\#2}}^*) = last(D_{A\#2}) = \{D_{A\#2}\}$.

Finally, for the entire $r_A^M = C_{\#1}.{D_{A\#2}}^*$ and its sequence operator we put:

$empty(r_A^M) = empty(C_{\#1}) \wedge empty({D_{A\#2}}^*) = \texttt{false} \wedge \texttt{true} = \texttt{false}$,
$first(r_A^M) = first(C_{\#1}) = \{C_{\#1}\}$ *since* $empty(C_{\#1}) = \texttt{false}$,
$follow(r_A^M) =$
$follow(C_{\#1}) \cup follow({D_{A\#2}}^*) \cup [last(C_{\#1}) \times first({D_{A\#2}}^*)] =$
$\{\} \cup \{(D_{A\#2}, D_{A\#2})\} \cup [\{C_{\#1}\} \times \{D_{A\#2}\}] =$
$\{(D_{A\#2}, D_{A\#2}), (C_{\#1}, D_{A\#2})\}$, *and*
$last(r_A^M) = last({D_{A\#2}}^*) \cup last(C_{\#1}) = \{D_{A\#2}\} \cup \{C_{\#1}\} =$
$\{D_{A\#2}, C_{\#1}\}$ *since* $empty({D_{A\#2}}^*) = \texttt{true}$.

When interpreting the previous results, we can conclude that the language $L(r_A^M)$ does not contain the empty word ϵ, its words begin with $C_{\#1}$, or that they end either with $C_{\#1}$ or $D_{A\#2}$.

And how to put all the defined auxiliary functions together to finally construct an automaton that would be able to recognize words from $L(r)$ for a given regular expression $r \in RE(\Sigma)$? The answer is given in the following definition of Glushkov automaton. As already outlined, note that the used auxiliary functions are based on r^M and not directly r.

Definition 2.8 (Glushkov Automaton)**.** *Let r be a regular expression over an alphabet Σ such that r is 1-unambiguous, and r^M its marked regular expression. Then we define a* Glushkov automaton *for r to be a deterministic finite automaton $\mathcal{A}_r = (Q, \Sigma, \delta, q_0, F)$, where:*

- *$Q = symbols(r^M) \cup \{q_0\}$ is a set of states ($symbols(r^M) \subset \Sigma^M$),*
- *Σ is the original input alphabet,*
- *δ is a transition function $Q \times \Sigma \to Q$ such that*
  - *for each $s \in first(r^M)$ we put $\delta(q_0, sym(s)) = s$, and*
  - *for each $(s_1, s_2) \in follow(r^M)$ we put $\delta(s_1, sym(s_2)) = s_2$,*
- *$q_0 \in Q$ is the artificially introduced initial state such that $q_0 \notin \Sigma^M$, and*
- *$F \subseteq Q$ is a set of accepting states such that*
  - *if $empty(r^M) =$ `true` then $F = last(r^M) \cup \{q_0\}$,*
  - *otherwise $F = last(r^M)$.*

It is important to emphasize that the previous definition only works since we assumed a regular expression r to be 1-unambiguous. Because of this feature we are sure that $\forall\, s_1,\, s_2 \in first(r^M),\, s_1 \neq s_2$: $sym(s_1) \neq sym(s_2)$. In other words, we know that $\neg\, \exists\, a \in \Sigma$ for which there would exist s_1 and $s_2 \in first(r^M)$, $s_1 \neq s_2$ such that $sym(s_1) = a = sym(s_2)$. Analogously we also know that $\forall\, (s, s_1)$ and $(s, s_2) \in follow(r^M)$, $s_1 \neq s_2$: $sym(s_1) \neq sym(s_2)$ once again. Both these observations directly imply from Definition 2.5.

As a result, the above transition function δ is defined correctly, and so the entire $\mathcal{A}_r$ is defined correctly. The practical consequence is that when r is 1-unambiguous, the corresponding Glushkov automaton $\mathcal{A}_r$ is deterministic (hence we can also talk about such r as a deterministic regular expression). On the contrary, when r would not satisfy the 1-unambiguity condition, the resulting Glushkov automaton would require to be constructed as a nondeterministic one.

Example 2.7. *Now we are ready to describe the Glushkov automaton $\mathcal{A}_{r_A}$ for our regular expression $r_A = C.D_A{}^*$. To construct it, we first need to evaluate all the empty, first, follow and last auxiliary functions on r_A^M, which we already managed to do in Example 2.6.*

Directly applying the definition, we can write that $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, where: $Q = \{q_0, C_{\#1}, D_{A\#2}\}$ is a set of states, $\delta = \{(q_0, C, C_{\#1}), (C_{\#1}, D_A, D_{A\#2}),$

$(D_{A\#2}, D_A, D_{A\#2})\}$ *is a transition function,* q_0 *is the designated initial state and* $F = \{C_{\#1},\ D_{A\#2}\}$ *is a set of accepting states.*

Because we chose the left-to-right approach for numbering occurrences when generating the marked symbols, we can leave their complicated names and simply use 0 *for the designated initial state instead of* q_0*, and* i *instead of* $x_{\#i} \in symbols(r^M)$ *in case of the remaining states. Note that this conversion is just for illustration purposes and has no effect on meaning of our definitions.*

The resulting automaton would then be $\mathcal{A}_{r_A} = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$*, where:* $Q = \{0,\ 1,\ 2\}$*,* $\delta = \{(0, C, 1),\ (1, D_A, 2),\ (2, D_A, 2)\}$*,* $q_0 = 0$ *and* $F = \{1,\ 2\}$*.*

This final Glushkov automaton $\mathcal{A}_{r_A}$ *is depicted in Figure 2.5. It is obviously a different automaton when comparing to the automaton discussed in Example 2.5 and presented in Figure 2.4, but both of them accept exactly the same* $L(r_A)$*, which only illustrates the general fact that there may exist different automata, yet serving the same objective.*

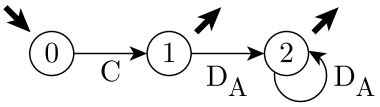


Figure 2.5: Glushkov automaton $\mathcal{A}_{r_A}$ for $r_A = C.D_A{}^*$

Just to conclude, having a 1-unambiguous regular expression r, we use $\mathcal{A}_r$ to denote its corresponding Glushkov automaton, i.e. a deterministic finite automaton that is capable of recognizing the language $L(r)$.

2.4 Regular Tree Grammars

At this moment we have already described how we model XML documents we want to work with, and we have also provided basic definitions from the theory of finite automata, regular languages and regular expressions. The question is, how to represent restrictions posed by schemata against which we actually would like to define our intended corrections.

2.4.1 Regular Tree Grammars

The natural answer to the previous question could be the notion of *regular tree grammars* [60]. Via them, we can directly describe how data trees should look like such that they conform to our expectations of the allowed nesting of elements.

Definition 2.9 (Regular Tree Grammar)**.** *A* regular tree grammar *is a tuple* $\mathcal{G} = (N,\ T,\ S,\ P)$*, where:*

- N *is a set of* nonterminal symbols*,*
- T *is a set of* terminal symbols*,*
- $S \subseteq N$ *is a nonempty set of* starting symbols*,*

- *P is a set of* production rules*, each rule of the form $[t,\ r \to n]$, where:*
  - *$t \in T$ is a terminal symbol,*
  - *r is a regular expression over N,*
  - *$n \in N$ is a nonterminal symbol.*

Without loss of generality, for each $t \in T$ and $n \in N$ there exists at most one rule $[t,\ r \to n] \in P$.

Despite these grammars do not allow us to describe all the constructs of DTD and XSD languages, they represent an appropriate way of capturing those constructs that describe the allowed nesting and content of elements, i.e. exactly those which we are interested in.

Intuitively, terminal symbols (or just *terminals*) correspond to labels of elements (their names) and nonterminal symbols (*nonterminals*) represent sort of types of these elements. It is worth highlighting that regular expressions in production rules are based on an alphabet of nonterminal and not terminal symbols. As a consequence, we are able to describe elements with the same label, but different content depending on different contexts.

Also note that the last condition in the definition enables us to reference particular production rules just by fixing values of a terminal t and nonterminal n, i.e. by $\mathcal{F}_{t,n}$ we can easily reference the only production rule that can exist in P for a pair of such t and n symbols.

Example 2.8. *Suppose we have a regular tree grammar $\mathcal{G} = (N,\ T,\ S,\ P)$, where $N = \{A,\ B,\ C,\ D_A,\ D_B\}$ are nonterminal symbols, $T = \{a,\ b,\ c,\ d\}$ are terminal symbols and $S = \{A,\ B\}$ are starting nonterminal symbols. The set P contains these production rules:*

$$\mathcal{F}_1 = \mathcal{F}_{a,A} = [\,a,\ C.{D_A}^* \to A\,],$$
$$\mathcal{F}_2 = \mathcal{F}_{b,B} = [\,b,\ {D_B}^* \to B\,],$$
$$\mathcal{F}_3 = \mathcal{F}_{c,C} = [\,c,\ \epsilon \to C\,],$$
$$\mathcal{F}_4 = \mathcal{F}_{d,D_A} = [\,d,\ C^* \to D_A\,] \text{ and}$$
$$\mathcal{F}_5 = \mathcal{F}_{d,D_B} = [\,d,\ A|B|C \to D_B\,].$$

2.4.2 Validity of Data Trees

Our situation, however, differs from the so far outlined and intuitive comprehension of the introduced tree grammars with their production rules – we do not want to use them for generating new data trees, but for validation – and later on correction as well – of the existing ones. In other words, our basic problem is that we have an XML document and we would like to decide, whether it conforms to a particular DTD or XSD schema – i.e. whether the given data tree abide by the corresponding regular tree grammar.

It also implicitly means that we need to know how to translate schemata into these grammars. However, this translation is not complicated and we will not discuss it in this thesis.

The following notion of an *interpretation tree* is the first step towards the definition of *data tree validity*.

Definition 2.10 (Interpretation Tree)**.** *Let $\mathcal{T} = (D, lab, val)$ be a data tree and $\mathcal{G} = (N, T, S, P)$ a regular tree grammar. An* interpretation tree *of data 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 nonterminal symbols such that:*
  - *for each node $p \in D$ and a sequence of its child nodes $p.0, p.1, \ldots, p.k$ for $k = fanOut(p) - 1$ there exists a production rule $[t, r \to n] \in P$ satisfying:*
    - *$int(p) = n$,*
    - *$lab(p) = t$, and*
    - *$int(p.0).int(p.1) \ldots int(p.k) \in L(r)$.*

This means that we take an underlying tree D of a given data tree $\mathcal{T}$ and attempt to resolve values of the interpretation function (i.e. nonterminal symbols) by finding appropriate production rules of the grammar $\mathcal{G}$ – those rules having a corresponding element label t as in the original data tree and, at the same time, a regular expression r matching the actual element content.

Note that our definition of the interpretation tree does not include any specific conditions posed on the root node, since this approach allows us to easily introduce concepts of both local and full validity.

Definition 2.11 (Data Tree Validity)**.** *Let $\mathcal{T} = (D, lab, val)$ be a data tree and $\mathcal{G} = (N, T, S, P)$ a regular tree grammar.*

We say that data tree $\mathcal{T}$ is locally valid *with respect to a production rule $\mathcal{F} = [t, r \to n] \in P$ of grammar $\mathcal{G}$, if and only if $\epsilon \in D$ and there exists at least one interpretation tree $\mathcal{N} = (D, int)$ of $\mathcal{T}$ against $\mathcal{G}$ such that $lab(\epsilon) = t$ and $int(\epsilon) = n$.*

Next, we say that data tree $\mathcal{T}$ is locally valid *with respect to grammar $\mathcal{G}$ in general, if and only if $\mathcal{T}$ is locally valid with respect to some (any) production rule $\mathcal{F} \in P$ of grammar $\mathcal{G}$.*

Finally, we say that data tree $\mathcal{T}$ is valid *with respect to grammar $\mathcal{G}$, if and only if $\epsilon \in D$ and there exists at least one interpretation tree $\mathcal{N} = (D, int)$ of $\mathcal{T}$ against $\mathcal{G}$ such that $int(\epsilon) \in S$.*

As a consequence, we also have a direct method how to decide whether a provided data tree $\mathcal{T}$ is valid with respect to some grammar $\mathcal{G}$, or whether it is not valid, i.e. is invalid in such case. The only thing we need to do is to attempt to construct an interpretation tree, starting with the root node and moving toward leaves, always trying to find suitable and matching production rules of the grammar.

It is also apparent that if data tree $\mathcal{T}$ is valid, it must also be locally valid.

Definition 2.12 (Regular Tree Language)**.** *Let $\mathcal{G} = (N, T, S, P)$ be a regular tree grammar. Given a production rule $\mathcal{F} \in P$ of grammar $\mathcal{G}$, we first define $L^{loc}_{\mathcal{F}}(\mathcal{G}) = \{\mathcal{T} \mid \mathcal{T}$ is locally valid with respect to $\mathcal{F}\}$. Then we also put $L^{loc}(\mathcal{G}) = \{\mathcal{T} \mid \mathcal{T}$ is locally valid with respect to $\mathcal{G}$ in general$\}$.*

And finally, we define $L(\mathcal{G}) = \{\mathcal{T} \mid \mathcal{T}$ is valid with respect to $\mathcal{G}\}$ as a regular tree language *of grammar $\mathcal{G}$.*

Example 2.9. *Now, we are ready to return to our data tree $\mathcal{T}$ from Example 2.2 (illustrated in Figure 2.3) and regular tree grammar $\mathcal{G}$ from Example 2.8.*

Data tree $\mathcal{T}$ is not valid against $\mathcal{G}$, since there cannot exist any interpretation tree $\mathcal{N}$ – this is true, for example, because of a node at position 0 (its label is equal to x that is apparently not allowed by a set of terminals T). So, despite $\mathcal{T}^{\Delta 1}$ is locally valid, both $\mathcal{T}^{\Delta 0}$ and $\mathcal{T}^{\Delta 2}$ are not, therefore, entire $\mathcal{T}$ cannot be locally valid and neither valid at all.

On the other hand, let us consider data tree $\mathcal{T}_3$ from Figure 2.6. This data tree is valid with respect to $\mathcal{G}$ and its interpretation tree $\mathcal{N}_3$ is depicted there as well. If $\mathcal{T}_3 = (D_3, lab_3, val_3)$, then we can write that $\mathcal{N}_3 = (D_3, int_3)$, where $int_3 = \{(\epsilon, A), (0, C), (1, D_A), (1.0, C), (2, D_A), (2.0, C), (2.1, C)\}$. We used production rules $\mathcal{F}_{a,A}$, $\mathcal{F}_{c,C}$, $\mathcal{F}_{d,D_A}$, $\mathcal{F}_{c,C}$, $\mathcal{F}_{d,D_A}$, $\mathcal{F}_{c,C}$ and $\mathcal{F}_{c,C}$ (in this order according to int_3) to construct this interpretation.

We will see later on that $\mathcal{T}_3$ is one of the possible corrections of the original data tree $\mathcal{T}$ that our correction algorithm is capable to find.

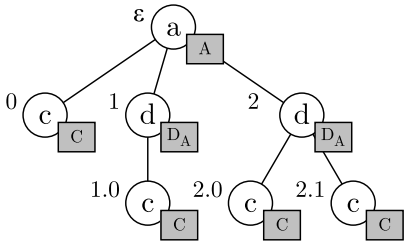


Figure 2.6: Data tree $\mathcal{T}_3$ and its interpretation tree

2.4.3 Classes of Regular Tree Grammars

The problem of regular tree grammars is that they have unnecessarily high expressive power comparing to our needs. Besides, we have also outlined that DTD does not offer as expressive possibilities as XSD does. This observation is formally described by the following notion of *competing nonterminals* [60].

Definition 2.13 (Competing Nonterminal Symbols)**.** *Let $\mathcal{G} = (N, T, S, P)$ be a regular tree grammar and n_1 and $n_2 \in N$, $n_1 \neq n_2$ are two different nonterminal symbols. We say that n_1 and n_2 are* competing *with each other, if there exist two production rules $[t, r_1 \to n_1]$ and $[t, r_2 \to n_2] \in P$ sharing the same terminal symbol t.*

The presence of such competing nonterminals in grammars makes their usage more complicated, thus, it makes sense to define grammar subclasses with less expressive power, and, therefore, hopefully with easier processing.

The explanation is simple. Assume that we have a data tree $\mathcal{T} = (D, lab, val)$ and some of its nodes $p \in D$ such that this node has a label equal to $a = lab(p)$. When we are searching for the suitable production rules that match p – those having exactly this label a as a terminal symbol on their left sides, i.e. production rules of the form $[a, r \to n]$ for some regular expressions r and nonterminal

symbols n – we might be able to find more of such production rules and, so, any of them could be considered as a potential candidate when constructing the required interpretation tree.

We will use two classes of regular tree grammars, though even other could be introduced as well. Both of them will pose different restrictions on the presence of competing nonterminals [60].

Definition 2.14 (Local and Single-Type Tree Grammars)**.** *We say that a regular tree grammar $\mathcal{G} = (N, T, S, P)$ is a* local tree grammar*, if it has no competing nonterminals, i.e. $\neg\, n_1, n_2 \in N, n_1 \neq n_2$ such that n_1 and n_2 would be competing with each other.*

We say that $\mathcal{G} = (N, T, S, P)$ is a single-type tree grammar*, if for each its production rule $[t, r \to n] \in P$ all the nonterminal symbols in $symbols(r)$ do not compete with each other, and above that the starting nonterminal symbols in S do not compete with each other too.*

Apparently, each local tree grammar is a single-type tree grammar and each single-type tree grammar is a regular tree grammar. On the other hand, some regular tree grammars are not single-type, and, analogously, some single-type tree grammars are not local ones. 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.

Without any further details, we can assume that schemata in DTD correspond to local tree grammars, and that schemata in XSD (largely) correspond to single-type tree grammars [60].

When considering DTD, content of an element always depends only on its name and never its context. This feature is in a direct correspondence with the full absence of competing nonterminals. In other words, we do not need to distinguish between terminals and nonterminals at all.

Processing of XML documents according to XSD, on the other hand, is more complicated, because there can exist more possible types (nonterminals) for elements with the same name (terminal).

Fortunately enough, when making decisions on the validity of data trees, we are still always guaranteed in both these classes that at most one appropriate production rule may exist – and, therefore, either there is no interpretation tree (and the processed data tree is not valid), or there is right one such tree (and so the data tree is valid). This is important because regular tree grammars may generally lead to more interpretation trees.

Anyway, our correction model efficiently supports even the class of regular tree grammars.

Example 2.10. *Our regular tree grammar $\mathcal{G}$ from Example 2.8 is a single-type tree grammar, but not a local tree grammar, because there are nonterminal symbols D_A and D_B that compete with each other (via production rules $\mathcal{F}_4 = \mathcal{F}_{d,D_A}$ and $\mathcal{F}_5 = \mathcal{F}_{d,D_B}$ they both lead to the same terminal d).*

2.4.4 Consistency of Grammars

Unfortunately, even a syntactically correct grammar may not be consistent. This means that there simply does not exist a single data tree that would be valid

with respect to such grammar. However, even grammars that are consistent in this way may contain production rules that might cause us difficulties in the correction model we are about to propose. Therefore we try to restrict ourselves only to grammars that are safe for us to use.

Let $\mathcal{G} = (N, T, S, P)$ be a regular tree grammar. Given a nonterminal symbol $n \in N$, we first put $R^n_{-1} = \emptyset$, $R^n_0 = \bigcup_{\mathcal{F}=[t,r\to n]\in P} symbols(r)$, and then inductively $R^n_i = R^n_{i-1} \cup \bigcup_{n'\in R^n_{i-1}\setminus R^n_{i-2}, \mathcal{F}'=[t',r'\to n']\in P} symbols(r')$ for each $i \in \mathbb{N}$. Since $\mathcal{G}$ is finite, there must exist a fixpoint $k \in \mathbb{N}$ such that $R^n_k = R^n_{k+1}$, and so we can define $ReachableNonterminals(\mathcal{G}, n) = \bigcup_{i\in\mathbb{N}_0} R^n_i = R^n_k$ as a set of all nonterminal symbols that are *reachable* from the given nonterminal n using production rules of $\mathcal{G}$.

Having the notion of reachability for individual nonterminals, let us now define $ReachableNonterminals(\mathcal{G}) = \bigcup_{n\in S}(\{n\} \cup ReachableNonterminals(\mathcal{G}, n)) = S \cup \bigcup_{n\in S} ReachableNonterminals(\mathcal{G}, n)$ to be a set of all *reachable nonterminals* of grammar $\mathcal{G}$, and then $ReachableRules(\mathcal{G}) = \{\mathcal{F} \mid \mathcal{F} \in P, \mathcal{F} = [t, r \to n]$ and $n \in ReachableNonterminals(\mathcal{G})\}$ to be a set of all *reachable rules* of $\mathcal{G}$.

Definition 2.15 (Consistency of Grammars)**.** *We say that a regular tree grammar $\mathcal{G} = (N, T, S, P)$ is an* inconsistent grammar*, if and only if $L(\mathcal{G}) = \emptyset$. Otherwise $\mathcal{G}$ is a* consistent grammar*. We say that a production rule $\mathcal{F} \in P$ is a* useless rule*, if and only if $L^{loc}_{\mathcal{F}}(\mathcal{G}) = \emptyset$.*

At this point we have all the required definitions in order to formally describe the mentioned assumptions. In particular and from now on, we only suppose that all the regular tree grammars we work with are consistent grammars, and that none of their reachable rules $\mathcal{F} \in ReachableRules(\mathcal{G})$ is useless at the same time.

Though this constraint might seem to be unnecessarily restrictive – which is true – it allows us to make our further definitions of the correction model and algorithms easier. On the other hand, at least the requirement on the grammar consistency as such does not need to trouble us at all, since if a provided grammar was not consistent, then there would be no data tree valid with respect to such grammar, and so we could hardly find any corrections for a provided data tree to be corrected.

Later on we will show that none of our current assumptions is actually required at all, i.e. that our correction model can work with any regular tree grammar without exceptions.

3. Model

In the previous chapter we have discussed the basic theoretical background of our correction model and provided the essential notions from the areas of regular expressions, finite automata and regular tree grammars used to model structural validity of XML documents.

The aim of this chapter is to formally describe the proposed correction model, its capabilities, features and principles, whereas the question how to actually design recursive evaluation algorithms and to design them efficiently enough, we postpone until yet another chapter.

We start with a brief outline of all important notions and ideas in order to provide a basic overview of the entire model at a glance. Then, we focus on particular model components – step by step, and in a detail.

3.1 Model Overview

First of all we need to define actions we are allowed to use to transform data trees into valid ones. It is obvious that such operations would be the fundamental part of any correction model – directly influencing what type of corrections can and cannot be obtained at all. We call these transformations *edit operations*.

Using them we are able to add new leaf nodes, remove existing leaf nodes, and change labels of existing nodes. When composing these edit operations into suitable sequences, we are able to get more complex operations via which we are able to insert entire new subtrees, remove existing ones or recursively repair them as well.

If our main goal is to find all the corrections for a provided data tree with respect to a given grammar, i.e. to find all data trees that are valid and that are as close as possible to the original data tree – we figure out this task by finding sequences of edit operations, via which we can then acquire such data trees.

The correction algorithm itself processes a given data tree from its root node towards leaves. Being at a particular node, our task is to find its corrections – i.e. corrections of a sequence of its child nodes – efficiently inspecting new suitable sequences that are allowed by grammar and that can be acquired right using the introduced set of edit operations. We talk about these recursive assignments as *correction intents*.

The purpose of each correction intent lies in two levels. First, we need to horizontally correct the mentioned sequence of sibling nodes. This is motivated by a traditional approach for correction of ordinary words. Unfortunately, sibling nodes are not just flat sequences of nodes, they also (usually) have their own subtrees that need to be treated as well – vertically by recursive nesting.

Broadly speaking, correction intents either describe sort of an assignment for this recursive correction, but they also exactly define, what corrections do we actually want to inspect and what not.

When finding new permitted node sequences, we could dynamically generate them one by one – simply simulating a state space traversal within a corresponding finite automaton. Instead, we decided to inspect all the suitable node sequence corrections statically – all together in a form of *correction multigraphs*.

These structures directly encode such corrections and allow us to transform the problem of their finding to the problem of finding the shortest correction paths. Once these paths are obtained, they are encapsulated into compact *intent repair* structures. They are the final product of our correction process.

To summarize the entire model: provided with a potentially invalid data tree and a regular tree grammar to which it should conform, we traverse this data tree in a top-down manner, recursively invoking nested correction intents, constructing correction multigraphs, and finally gathering computed intent repairs with encapsulated shortest correction paths on the way back – until we return to the root node once again at the very end.

Having found the repair for the root node, we have found the required corrections for the entire data tree, and so the whole correction process terminates.

To provide another perspective – correction intents, correction multigraphs as well as intent repairs are all recursively nested structures. Each correction intent represents a description of a problem to be figured out – and for this purpose it recursively invokes other correction intents that are responsible for solving the identified subproblems. A correction multigraph is a structure that provides us a means to figure out a corresponding correction intent by finding the shortest correction paths inside it. Finally, an intent repair is a structure that represents a solution for a given correction intent.

The remaining parts of this chapter focus in detail on all the notions of the correction model we have just shortly sketched. In particular, we start with a description of edit operations and sequences into which they can be composed to. Then we discuss the universal interface of recursively nested correction intents. Having them, we define correction multigraphs together with correction paths, and specifically the shortest correction paths in which only we are interested in. Finally, having wrapped the found intent repairs, we also discuss how to unfold them to actually obtain all the edit sequences on which we built our correction model from the formal point of view.

3.2 Edit Operations

As we have outlined, edit operations are elementary transformations we use for altering invalid data trees into valid ones. They behave like functions, always performing small local modifications of a provided data tree. Applying them, we are able to add a new leaf node into a data tree ($addLeaf$), remove an existing leaf node ($removeLeaf$), or change a label of an existing node of any type ($renameNode$).

3.2.1 Edit Operations

However, before we can introduce edit operations formally, we need to describe how to derive a few auxiliary sets of underlying nodes – sets that help us to identify and grasp specific groups of nodes with respect to the execution of edit operations themselves.

So, given an underlying tree D, we first define the following set of nodes:

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

Next, given an underlying tree D and a node $p \in D$, $p \neq \epsilon$, $p = u.i$, $u \in \mathbb{N}_0^*$, $i \in \mathbb{N}_0$, we define yet the following auxiliary sets of nodes:

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

Nodes in a set $InsertPositions(D)$, together with the given underlying tree D itself, represent all the positions where a new leaf node can be inserted by a corresponding *addLeaf* edit operation. To do it, we must also shift all the potentially existing sibling nodes located to the right from the position p, where the insertion is being performed. In particular, we must shift these nodes (and, of course, their subtrees too, if present) by one to the right, i.e. to replace the original affected $ExpiredNodes(D,p)$ with nodes from $ShiftedRightNodes(D,p)$.

In case of a *removeLeaf* operation, we must analogously shift all the influenced sibling nodes from $ExpiredNodes(D,p)$ (and their subtrees once again) by one to the left to form $ShiftedLeftNodes(D,p)$.

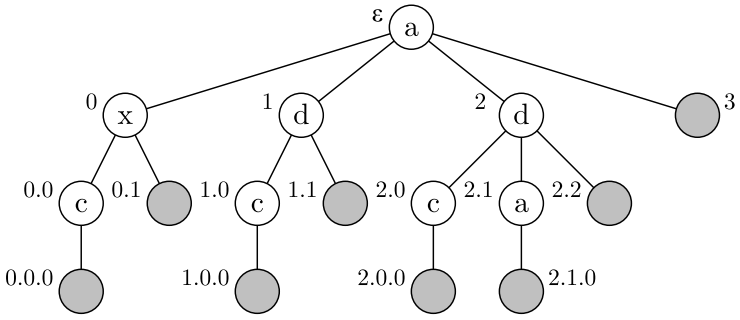


Figure 3.1: Data tree positions of $\mathcal{T}$ allowing leaf insertions

Example 3.1. *Assume that we have the same data tree $\mathcal{T}$ as in Example 2.2 and Figure 2.3, and so we have its underlying tree $D = \{\epsilon, 0, 0.0, 1, 1.0, 2, 2.0, 2.1\}$.*

First of all, what are the positions where a new leaf node can be added into D except D itself? They are $InsertPositions(D) = \{0.0.0, 0.1, 1.0.0, 1.1, 2.0.0, 2.1.0, 2.2, 3\}$. When looking at the tree structure depicted in Figure 3.1, these positions are marked as gray nodes without labels.

Now, suppose for example that we want to add a new leaf node at a position $p = 0$ *(right before the existing element with label equal to* x*, currently located at position* 0*). Then all the nodes in* $ExpiredNodes(D, 0) = \{0,\ 0.0,\ 1,\ 1.0,\ 2,\ 2.0,\ 2.1\}$ *become obsolete and need to be shifted by 1 to the right to become* $ShiftedRightNodes(D, 0) = \{1,\ 1.0,\ 2,\ 2.0,\ 3,\ 3.0,\ 3.1\}$ *in the end.*

Now, we are ready to provide the definition of all the edit operations we want to introduce. Its purpose is not only to enumerate which edit operations we actually define with respect to certain data tree assumptions, but we also describe the particular effect they have on both the underlying tree and a pair of label and value functions.

Definition 3.1 (Edit Operations)**.** *An* edit operation e *is a partial function transforming an original data tree* $\mathcal{T}_0 = (D_0,\ lab_0,\ val_0)$ *into a new data tree* $\mathcal{T}_1 = (D_1,\ lab_1,\ val_1)$*, denoted as* $\mathcal{T}_0 \xrightarrow{e} \mathcal{T}_1$*. In case the following conditions are satisfied, we define these edit operations in particular:*

- *For any* $p \in D_0 \cup InsertPositions(D_0)$ *such that* $p \neq \epsilon$*,* $p = u.i$*,* $u \in \mathbb{N}_0^*$*,* $i \in \mathbb{N}_0$*,* $u \notin DataNodes(D_0)$ *and* $a \in \mathbb{E} \cup \{\texttt{data}\}$ *we define an edit operation* $e = addLeaf(p, a)$ *such that for* $\mathcal{T}_1 = (D_1,\ lab_1,\ val_1)$ *then holds:*
  - $D_1 = [\, D_0 \setminus ExpiredNodes(D_0, p)\,] \cup \{p\} \cup ShiftedRightNodes(D_0, p)$*.*
  - *For the unaffected nodes, i.e.* $\forall\, w \in [\, D_0 \setminus ExpiredNodes(D_0, p)\,]$*:* $lab_1(w) = lab_0(w)$*, and* $val_1(w) = val_0(w)$ *in case* $val_0(w)$ *was defined.*
  - *For the newly inserted node:* $lab_1(p) = a$*, and* $val_1(p) = \perp$ *in case of* $a = \texttt{data}$*.*
  - *For the shifted nodes, i.e.* $\forall\, (u.(k{+}1).v) \in ShiftedRightNodes(D_0, p)$*:* $lab_1(u.(k+1).v) = lab_0(u.k.v)$*, and* $val_1(u.(k+1).v) = val_0(u.k.v)$ *in case* $val_0(u.k.v)$ *was defined.*
- *For an empty tree* $p = \epsilon$*,* $D_0 = \emptyset$ *and* $a \in \mathbb{E} \cup \{\texttt{data}\}$ *we define an edit operation* $e = addLeaf(p, a)$ *such that:*
  - $D_1 = \{p\}$*.*
  - $lab_1(p) = a$*, and* $val_1(p) = \perp$ *in case of* $a = \texttt{data}$*.*
- *For any* $p \in LeafNodes(D_0)$ *such that* $p \neq \epsilon$*,* $p = u.i$*,* $u \in \mathbb{N}_0^*$*,* $i \in \mathbb{N}_0$ *we define* $e = removeLeaf(p)$ *as follows:*
  - $D_1 = [\, D_0 \setminus ExpiredNodes(D_0, p)\,] \cup ShiftedLeftNodes(D_0, p)$*.*
  - *For the unaffected nodes, i.e.* $\forall\, w \in [\, D_0 \setminus ExpiredNodes(D_0, p)\,]$*:* $lab_1(w) = lab_0(w)$*, and* $val_1(w) = val_0(w)$ *in case* $val_0(w)$ *was defined.*
  - *For the shifted nodes, i.e.* $\forall\, (u.(k{-}1).v) \in ShiftedLeftNodes(D_0, p)$*:* $lab_1(u.(k-1).v) = lab_0(u.k.v)$*, and* $val_1(u.(k-1).v) = val_0(u.k.v)$ *in case* $val_0(u.k.v)$ *was defined.*

- *For the root node $p = \epsilon$, $D_0 = \{\epsilon\}$ we define $e = removeLeaf(p)$ this way:*
  - $D_1 = \emptyset$.
  - $lab_1 = \emptyset$, *and*
    $val_1 = \emptyset$.
- *For any $p \in D_0$, $a \in \mathbb{E} \cup \{\texttt{data}\}$ and $a \neq lab_0(p)$ we define an edit operation $e = renameNode(p, a)$ as follows:*
  - $D_1 = D_0$.
  - *For the unaffected nodes, i.e. $\forall\, w \in [\, D_0 \setminus \{p\}\,]$:*
    $lab_1(w) = lab_0(w)$, *and*
    $val_1(w) = val_0(w)$ *in case $val_0(w)$ was defined.*
  - *For the modified node:*
    $lab_1(p) = a$, *and*
    $val_1(p) = \perp$ *in case $a = \texttt{data}$.*

Though the intended meaning of all these edit operations is straightforward, the definition, however, talks about a partial function – this is because we define edit operations only on those data trees, where the given transformation makes sense and is defined correctly. For example, we can only apply $removeLeaf(p)$ operation, if p is present in a processed data tree and it is really a leaf node.

From now on, we will always implicitly assume that all edit operations we work with are meaningful and correctly defined in this sense.

Next, since our correction model is only interested in the structural corrections of data trees, we cannot actually generate data values for newly added data nodes, so we simply use our already introduced special undefined value $\perp$, which in this case unfolds to an empty string from the value function point of view.

Example 3.2. *Once again, let us return to our data tree $\mathcal{T} = (D, lab, val)$ that is presented in Figure 2.3. To illustrate how edit operations work, assume, for example, we want to add a new leaf node with label c to position 0, i.e. to perform an edit operation $e = addLeaf(0, c)$. First of all, note that this operation is defined on $\mathcal{T}$ correctly, since $0 \in D \cup InsertPositions(D)$.*

For we have already evaluated all the required auxiliary functions in Example 3.1, we can directly focus on a new data tree $\mathcal{T}' = (D', lab', val')$ that is a result of the transformation $\mathcal{T} \xrightarrow{e} \mathcal{T}'$.

In particular, $D' = \{\epsilon, 0, 1, 1.0, 2, 2.0, 3, 3.0, 3.1\}$, $lab' = \{(\epsilon, a), (0, c), (1, x), (1.0, c), (2, d), (2.0, c), (3, d), (3.0, c), (3.1, a)\}$, and, finally, $val' = \emptyset$. This resulting data tree $\mathcal{T}'$ is depicted in Figure 3.2.

Finally, let us also emphasize that – despite the definition of all the edit operations might seem a bit technically complicated from the perspective of auxiliary node sets and the described and required renumbering of nodes themselves – it is just a formal aspect and this renumbering happens implicitly and without any effort at the implementation level, i.e. does not need to be and is really not evaluated by the correction algorithms we proposed.

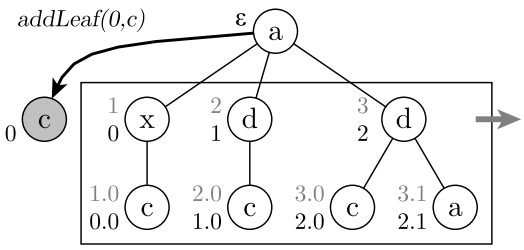


Figure 3.2: Applying edit operation $addLeaf(0, c)$ on data tree $\mathcal{T}$

3.2.2 Edit Sequences

Apparently, edit operations may be composed into more complex operations, simply when using and applying them in sequences. As a consequence, via appropriate sequences of edit operations, we are capable of inserting entire new subtrees into data trees, deleting existing ones or recursively repairing them.

Formally, let us provide the following definition.

Definition 3.2 (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 $\forall\, i \in \mathbb{N}_0,\ 0 \leq i < n$: $\mathcal{T}_i \xrightarrow{e_{i+1}} \mathcal{T}_{i+1}$ are correctly defined.*

Then we say that $S = \langle e_1, \ldots, e_n \rangle$ is a sequence of edit operations *(or just an* edit sequence*) that transforms data tree $\mathcal{T}_0$ into data tree $\mathcal{T}_n$, and instead of $\mathcal{T}_0 \xrightarrow{e_1} \mathcal{T}_1 \ldots \mathcal{T}_{n-1} \xrightarrow{e_n} \mathcal{T}_n$ we can shortly write $\mathcal{T}_0 \xrightarrow{S} \mathcal{T}_n$.*

We use angle brackets for sequences in general to make them easily differentiable, and we also assume that standard operations defined over sequences (like their concatenation) are evaluated in a natural way. Finally, a sequence is *empty*, if $n = 0$, i.e. $S = \langle\rangle$.

Example 3.3. *Suppose that we have our data tree $\mathcal{T}$ and we would like to delete its entire subtree rooted at position 2. To attain this objective, we could use a sequence of edit operations $\langle removeLeaf(2.1), removeLeaf(2.0), removeLeaf(2)\rangle$.*

On the other hand, we could also use another edit sequence where we would remove both child nodes starting with the first one and not the second one, i.e. a sequence $\langle removeLeaf(2.0), removeLeaf(2.0), removeLeaf(2)\rangle$. Notice, how the position parameters of the $removeLeaf$ edit operations have thus changed.

The previous example intentionally suggests that several issues may arise when finding suitable sequences of edit operations that lead to corrected data trees.

First, we must realize that there always exist many edit sequences having the same effect, i.e. although different sequences, still producing the same data trees when applied on the same source data tree. Sometimes we can reorder operations even without the necessity of changing their position parameters (which was not the case of the previous example), or we can add pairs of operations that mutually revoke and compensate the effect each other (e.g. to add a new leaf node and then, sooner or later, remove it once again).

To be precise, for each edit sequence we would always be able to find infinitely many other different sequences having the same result. So, our correction algo-

rithm must inspect the suitable sequences carefully enough and only focus on a finite number of the meaningful ones.

3.2.3 Costs of Edit Operations

Since we are only interested in finding valid data trees that are as close as possible to the original data tree to be corrected, we apparently have to introduce a means how to measure such proximity. The easiest way to achieve this need could be to introduce costs over the introduced edit operations. So, let us define the following notion of a *cost function*.

Definition 3.3 (Cost Function)**.** *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be data trees, and e an edit operation such that $\mathcal{T}_1 \xrightarrow{e} \mathcal{T}_2$ is correctly defined.*

Then we define $cost(e)$ to be a function assigning to e its positive cost *from $\mathbb{R}^+$, i.e. the domain of all the positive real numbers. Though it is formally a function of its explicitly listed arguments and the source data tree $\mathcal{T}_1$ as well, beside the general dependence on the edit operation type ($addLeaf$, $removeLeaf$ and $renameNode$), only the following dependencies are permitted:*

- *If $e = addLeaf(p, a)$ with some $p \in D \cup InsertPositions(D)$ and $a \in \mathbb{E} \cup \{\texttt{data}\}$, then $cost(e)$ may only depend on the node label parameter a.*
- *If $e = removeLeaf(p)$ with some $p \in LeafNodes(D)$, $cost(e)$ may only depend on the current node label $lab(p)$ and the current data value $val(p)$, but must not depend on the position p itself, i.e. value of this position as a word in $\mathbb{N}_0^*$.*
- *If $e = renameNode(p, a)$ with some $p \in D$ and $a \in \mathbb{E} \cup \{\texttt{data}\}$, then $cost(e)$ may only depend on the newly requested node label parameter a, the current node label $lab(p)$ and the current data value $val(p)$, but must not depend on the position p itself, i.e. value of this position as a word in $\mathbb{N}_0^*$.*

Intuitively, having a sequence of edit operations $E = \langle e_1, \ldots, e_k \rangle$ for some $k \in \mathbb{N}_0$, then we define $cost(E) = \sum_{i=1}^{k} cost(e_i)$.

The lower the cost is, the smaller the transformation impact on a given data tree is. And since values of the cost function may only be positive (i.e. cannot be negative, nor equal to zero specifically), the more edit operations in an edit sequence, the higher the overall cost will be.

Because of the reasons that become obvious once we introduce repairing instructions later on, we also had several dependency requirements on the cost function itself. Not only that it must behave deterministically, but its values must not be dependent especially on the position parameters of edit operations (although we have explicitly listed them as arguments), nor any other external information except predefined constants. On the other hand we can use node label parameters, or information about nodes from the original data tree (values of *lab* and *val* functions) without limitation.

From the practical perspective, we can assign each edit operation with the same cost (ignoring all the offered dependency possibilities at all), or we can assign each edit operation type predefined (and potentially different) constant values,

or we can even compute more complex costs dynamically, like, for example, when we would like to consider lexical distances between the original and new node labels in case of the *renameNode* edit operation.

Without loss of generality, in all the following examples and also for the purpose of the evaluation of the entire correction framework, we simply decided to assign each edit operation a unit cost equal to 1, since this approach does not make any differences between particular operation types, i.e. does not prefer some operation types to the other ones.

Example 3.4. *Suppose that we have three edit operations* $e_1 = renameNode(0, c)$, $e_2 = removeLeaf(0.0)$ *and* $e_3 = renameNode(2.1, c)$ *that together forms an edit sequence* $S_3 = \langle e_1, e_2, e_3 \rangle$.

Applying this sequence to our data tree $\mathcal{T}$ *from Figure 2.3 we obtain a data tree* $\mathcal{T}_3$, *i.e.* $\mathcal{T} \xrightarrow{S_3} \mathcal{T}_3$. *This whole transformation is illustrated in Figure 3.3, and the resulting data tree* $\mathcal{T}_3$ *(together with its interpretation) was already presented as well in Figure 2.6 in the previous chapter.*

If each used edit operation is assigned a unit cost, then the cost of the entire transformation is equal to 3, *i.e.* $cost(S_3) = 3$.

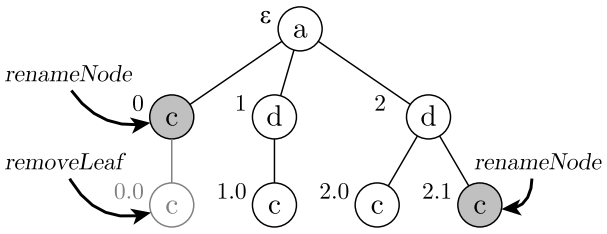


Figure 3.3: Applying an edit sequence S_3 on data tree $\mathcal{T}$

All in all, though our restrictions posed on the behavior of the cost function may seem to be restrictive, we actually believe they have no impact on the practical usability of the whole framework at all.

3.2.4 Data Tree Distances

Having defined costs of edit operations, we are able to introduce distances between data trees in general.

Definition 3.4 (Data Tree Distance)**.** *Assume that* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *are two data trees and* $\mathcal{S}$ *is a set of all sequences of edit operations that are capable of transforming* $\mathcal{T}_1$ *into* $\mathcal{T}_2$. *We define a data tree* distance *of* $\mathcal{T}_1$ *and* $\mathcal{T}_2$ *to be* $dist(\mathcal{T}_1, \mathcal{T}_2) = \min_{E \in \mathcal{S}} cost(E)$.

In other words, the distance of two data trees is equal to a minimal cost required for transforming one data tree into the other one using the set of introduced edit operations. Despite it is not important for our model, note also that this distance is not generally symmetric, since we do not require that the costs of complementary edit operations (for example, mutually revoking $addLeaf$ and $removeLeaf$ operations) are equivalent.

Definition 3.5 (Data Tree to Grammar Distance)**.** *Given a regular tree grammar $\mathcal{G}$ and the corresponding regular tree language $L(\mathcal{G})$ (it means a set of all data trees that are valid against $\mathcal{G}$), we define a distance between a data tree $\mathcal{T}_1$ and the language $L(\mathcal{G})$ as $dist(\mathcal{T}_1, L(\mathcal{G})) = \min_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}_1, \mathcal{T}_2)$.*

Provided a potentially (but not necessarily) invalid data tree $\mathcal{T}$ to be corrected using our algorithm, we would like to examine all data trees that are valid with respect to a given grammar $\mathcal{G}$, and select those ones that are as close to the original one as possible – i.e. those having exactly the minimal distance from the original data tree – i.e. right those having the data tree distance equal to the value of $dist(\mathcal{T}, L(\mathcal{G}))$.

Definition 3.6 (Correction Problem)**.** *Given a data tree $\mathcal{T}$ and a regular tree grammar $\mathcal{G}$, we define a* correction problem *of $\mathcal{T}$ with respect to $\mathcal{G}$ as a problem of evaluating $correct(\mathcal{T}, \mathcal{G}) = \text{argmin}_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}, \mathcal{T}_2) = \{\mathcal{T}_2 \mid \mathcal{T}_2 \in L(\mathcal{G})$ and $dist(\mathcal{T}, \mathcal{T}_2) = dist(\mathcal{T}, \mathcal{G})\}$.*

Example 3.5. *Let us now return to our data tree $\mathcal{T}$ and a sequence of edit operations $S_3 = \langle renameNode(0, c), removeLeaf(0.0), renameNode(2.1, c)\rangle$ discussed in the previous example.*

Assuming unit costs for all the edit operation, $cost(S_3) = 3$. This is actually the minimal cost of any edit sequence that is capable to transform the original data tree $\mathcal{T}$ to any data tree that is valid with respect to our grammar $\mathcal{G}$. In other words and as we will see later on, $dist(\mathcal{T}, L(\mathcal{G})) = 3$.

Unfortunately, it is apparently not possible to inspect all such data trees, nor all such edit sequences in practice. Thus, the correction algorithm must be able to directly inspect only the optimal ones, or at most the promising ones.

3.2.5 Repairing Instructions

Since we do not only want to find all of the corrected data trees, but to find them efficiently and represent them compactly (i.e. to encode them without the need of explicitly generating or enumerating them as data trees themselves and nor edit sequences themselves during the correction process), we have to deal with another problem. Edit operations, as we have introduced them, are explicitly bound to particular positions to which they are associated and on which they impose their local transformation impact.

Example 3.6. *We can simply illustrate our motivation using Example 3.3. There were two suitable ways how to remove a node with label a at the original position* 2.1*. When we decided to remove it before its left sibling, we used an edit operation $removeLeaf(2.1)$, but when we wanted to remove it after having already removed its left sibling at first, positions shifted accordingly and we had to use $removeLeaf(2.0)$ instead.*

It is obvious that both these edit operations are formally different operations, though they not only pose the same data tree transformation, but they represent the same correction intent as such.

The previous example suggests that it might be useful to inspect and encode edit operations universally, not with any dependencies on particular positions that may change having applied other edit operations in the vicinity. To achieve this goal, we introduce the following notion of *repairing instructions*.

Definition 3.7 (Repairing Instructions)**.** *Having edit operations* $addLeaf(p, a)$*,* $removeLeaf(p)$ *and* $renameNode(p, a)$ *with optional node label* $a \in \mathbb{E} \cup \{\texttt{data}\}$ *parameters and explicitly bound positions* $p \in \mathbb{N}_0^*$*, we define* repairing instructions *as* `addLeaf`(a)*,* `removeLeaf` *and* `renameNode`(a) *as symbolic names for the corresponding edit operations with position parameters not yet evaluated, though still associated implicitly elsewhere.*

When searching for the minimal corrections, we will use these repairing instructions and not edit operations themselves. In other words, our correction algorithm does not directly produce suitable sequences of edit operations. Instead, it uses repairing instructions to describe them compactly. All this is enabled by *correction multigraph* structures, which we have already mentioned as well, though not yet explained. Within them, we are able to associate the involved repairing instructions with particular data tree positions indirectly.

The main reason for using repairing instructions is that we are then able to inspect all the possible corrections much efficiently, as well as we are able to represent them in compact *intent repair* structures once we have found them.

Having the correction of a given data tree finished, i.e. having the intent repair for the entire data tree constructed, we can recursively unfold it, translate all the included repairing instructions to the corresponding edit operations with appropriately resolved and assigned positions, to finally obtain all the edit sequences we were searching for. However, we postpone the description of such translation until the very end of this chapter.

If the correction algorithm is about to find all the minimal corrections, it still needs to be aware of the way how this minimality should be measured. So, we need to transfer the already introduced notion of the cost function defined over the edit operations to the repairing instructions as well. And we have to do it carefully, for we must ensure that the correction algorithm still finds all the corrected data trees exactly as we have defined them before we came with these repairing instructions.

Definition 3.8 (Cost of Repairing Instructions)**.** *Given a repairing instruction* i *that is implicitly associated to a data tree position* $p \in \mathbb{N}_0^*$*, we define* $cost(i)$ *to be a function assigning to* i *its positive* cost *from* $\mathbb{R}^+$ *such that:*

- *if* $i =$ `addLeaf`(a) *for some* $a \in \mathbb{E} \cup \{\texttt{data}\}$*,*
  then necessarily $cost(i) = cost(addLeaf(p, a))$*,*
- *if* $i =$ `removeLeaf`*,*
  then necessarily $cost(i) = cost(removeLeaf(p))$*, and, finally,*
- *if* $i =$ `renameNode`(a) *for some* $a \in \mathbb{E} \cup \{\texttt{data}\}$*,*
  then necessarily $cost(i) = cost(renameNode(p, a))$*.*

In other words, the cost of a given repairing instruction i has to be identical to the cost of the corresponding edit operation e, i.e. an edit operation to which this repairing instruction will be translated later on once the position parameters can be expressed explicitly, i.e. an edit operation that this repairing instruction symbolically represents.

Now we can also see the reason of all the dependency restrictions we pose in the definition of the original cost function over the edit operations. We could not use any other dependencies, simply because they would not be available or accessible in case of the repairing instructions we plan to use.

To shortly sum it up before we move forward, the goal of the whole correction algorithm is to find all the valid data trees that have the minimal distance from the original data tree $\mathcal{T}$ to be corrected. This means we need to find suitable sequences of edit operations that permit us to transform the original data tree $\mathcal{T}$ to all its corrections, i.e. all valid data trees that are as close as possible to the original $\mathcal{T}$. And while these sequences are composed of edit operations through which we are able to perform small and local transformations of data trees in general, repairing instructions, on the other hand, allow us to search for such edit sequences more efficiently, and also represent them in a compact way.

3.3 Correction Intents

So far we have described the goal of the correction model and constructs we can use to achieve it. In other words, we have edit operations capable of transforming invalid data trees into valid ones – specifically those that are close to the original trees. Now we focus on a question, how to actually harness the possibilities the edit operations offer us to actually find the required corrections.

3.3.1 Grammar Contexts

As we have also outlined, the correction algorithm processes a provided data tree $\mathcal{T} = (D, lab, val)$ from its root node ϵ towards leaves. Suppose now that we are at a particular node $p \in D$ and our task is to correct it with respect to a grammar $\mathcal{G} = (N, T, S, P)$, i.e. to correct a sequence of its child nodes $u = \langle u_1, \ldots, u_k \rangle$ for some $k \in \mathbb{N}_0$, including their potentially existing subtrees.

From the recursive perspective this means that we must have already decided which label $t \in T$ this node p should have and to which nonterminal $n \in N$ this label should correspond, i.e. we must have already selected right one production rule $\mathcal{F}_{t,n} = [t, r \to n]$ with a regular expression r over N to which the sequence of sibling nodes u should conform.

In other words we have uniquely determined a *grammar context* $\mathcal{C}_{t,n}$ that determines the allowed content of node p.

Definition 3.9 (Grammar Contexts)**.** *Let $\mathcal{G} = (N, T, S, P)$ be a regular tree grammar, $t \in T$ a terminal symbol, $n \in N$ a nonterminal symbol, and finally $\mathcal{F}_{t,n} = [t, r \to n] \in P$ a production rule that matches both t and n (such rule must exist).*

Then we define a grammar context *of $\mathcal{G}$ for a pair of t and n to be a structure $\mathcal{C}_{t,n} = (r, N_A, P_A)$ where:*

- *r is a regular expression from $\mathcal{F}_{t,n}$ describing the allowed* content model,
- *$N_A = \{x \mid x \in symbols(r)\} \subseteq N$ is a set of the* allowed nonterminals *in r,*

- $P_A = \{\mathcal{F}' \mid \mathcal{F}' \in P,\, \mathcal{F}' = [t',\, r' \to n'],\, n' \in N_A\}$ *is a set of the* active rules.

Next, we define a starting grammar context *to be* $\mathcal{C}_\bullet = (r,\, N_A,\, P_A)$*, where:*

- $r = r_\bullet$ *is a* starting regular expression *defined as a choice over all the starting nonterminals, i.e.* $r_\bullet = n_1|\ldots|n_s$*, where* $s = |S|$ *and* $\forall\, i \in \mathbb{N}$*,* $1 \leq i \leq s$*:* $n_i \in S$ *and* $\forall\, i,\, j \in \mathbb{N}$*,* $1 \leq i < j \leq s$*:* $n_i \neq n_j$*,*
- $N_A = S$ *equals to the set of the starting nonterminals, and*
- P_A *is defined in the same way as above.*

Finally, we define an empty grammar context *to be* $\mathcal{C}_\epsilon = (r,\, N_A,\, P_A)$*, where:*

- $r = r_\epsilon = \epsilon$*,* $N_A = \emptyset$*, and* $P_A = \emptyset$*.*

Grammar contexts actually do not bring any new idea, they just allow us to make further definitions a bit easier, since they wrap all the related information together, and so they really serve as a complete description of a local grammar context to be applied when dealing with a given node sequence u.

Grammar contexts derived from particular production rules $\mathcal{F}_{t,n}$ are considered to be *standard* grammar contexts, and the remaining two the *special* ones. Also notice that if there exists a production rule for a given pair of symbols t and n, then there must be at most one such rule, and so the convention of denoting standard grammar contexts as $\mathcal{C}_{t,n}$ is therefore justified.

The purpose of the starting grammar context is to initiate the entire correction process for a provided data tree $\mathcal{T}$ (so that we process its root node ϵ with respect to all the possibilities the set of the allowed starting nonterminals offers), whilst the empty grammar context is used to remove existing subtrees.

The introduced notion of grammar contexts also enables us to process data trees in a universal way, regardless the node we are currently at, and what correction intent we have decided to follow and inspect. In other words, whatever the situation may be, grammar contexts are of the same structure, and therefore even correction intents we are about to introduce shortly can hopefully be defined in this universal way as well.

Example 3.7. *Assume our data tree* $\mathcal{T}$ *from Example 2.2 and grammar* $\mathcal{G} = (N,\, T,\, S,\, P)$ *with a set of the starting nonterminals* $S = \{A,\, B\}$ *from Example 2.8.*

Then the starting grammar context for $\mathcal{G}$ *is equal to* $\mathcal{C}_\bullet = (r,\, N_A,\, P_A)$*, where* $r = A|B$ *is the allowed content model,* $N_A = \{A,\, B\}$ *are the allowed nonterminals, and* $P_A = \{\mathcal{F}_{a,A},\, \mathcal{F}_{b,B}\}$ *are the active production rules.*

Next, suppose we chose, for example, to process the root node ϵ *of* $\mathcal{T}$ *in a way that we would like to preserve its current label* $lab(\epsilon) = a$*. This means we had assigned it the only available nonterminal symbol* A *via* $\mathcal{F}_{a,A}$ *(as this is the only active production rule from* P_A *that matches the given terminal* a*).*

Having fixed this pair of symbols a *and* A*, i.e. a rule* $\mathcal{F}_{a,A}$*, we have hence determined even the standard grammar context* $\mathcal{C}_{a,A}$ *we then use for further processing of a sequence* $\langle 0,\, 1,\, 2\rangle$*, i.e. a sequence of the child nodes of the root node* ϵ *together with their subtrees* $\mathcal{T}^{\Delta 0}$*,* $\mathcal{T}^{\Delta 1}$ *and* $\mathcal{T}^{\Delta 2}$ *respectively.*

It is not a coincidence that the way how we derive grammar contexts and then how we can recursively use them is in a direct correspondence with the way how the interpretation trees are constructed, i.e. the validity of data trees as such is defined. It is because our goal is nothing else than to find data trees that are valid, and so we must follow the definition of validity.

3.3.2 Horizontal and Vertical Correction

At this point we have introduced edit operations as a mechanism for altering data trees, and grammar contexts as a description of restrictions on the allowed content of nodes. In other words, we are provided with a means to pursue our correction goal, as well as we know what the correction results should satisfy, unfortunately we still do not know how such results should be acquired at all.

Suppose we are at a particular data tree node p and we have already decided to correct it with respect to a grammar context $\mathcal{C}_{t,n} = (r,\, N_A,\, P_A)$. This means we need to correct a sequence of its child nodes $u = \langle u_1, \ldots, u_k \rangle$ for some $k \in \mathbb{N}_0$, so that this sequence conforms to a regular expression r. Though we still need to explain a lot, this is actually and precisely a description of an *assignment* of one particular *correction intent*.

The idea of solving this correction intent goal is directly motivated by the traditional Levensthein metric and correction approach over ordinary strings (words). Its edit distance is based on a minimal number of operations (allowing to insert, delete or substitute a single symbol in a string) that are required to transform one string into another one.

We have analogously introduced the same three basic principles and ways of correction – in our scenario they are represented by the edit operations. The main difference to the situation with ordinary strings is that we are not facing just to flat sequences of nodes, but these nodes also (at least usually) have their own subtrees as well. And so we must focus either on the horizontal aspect of the whole correction, as well as on the vertical one.

Let us first discuss the horizontal correction. It is based on a traversal of the state space of the automaton $\mathcal{A}_r = (Q,\, \Sigma,\, \delta,\, q_0,\, F)$ recognizing $L(r)$. At the beginning, we have the entire sequence u unprocessed and we are at the initial automaton state q_0. The goal is to process u completely and terminate at any of the accepting states F.

Now, assume that we have already processed the first $0 \leq s \leq k$ nodes of u. This means that we have considered these nodes into the correction and that we have also potentially proposed some changes in order to pursue new and valid sequences. Anyway, we are at some automaton state q_s, and there is a node u_{s+1} ahead in the input sequence u (if there is any). All the actions we can consider now are directly determined by values of the transition function δ defined from the current state q_s.

This means that we consider all the allowed nonterminals $x \in N_A$ for which $\delta(q_s, x)$ is defined, and inspect all the possible actions we can perform at this position s and state q_s – i.e. to insert a new subtree before u_{s+1}, remove the existing subtree rooted at u_{s+1}, or recursively repair it with or without an option of changing its label $lab(u_{s+1})$. Whatever particular *action* we consider, we

simply change the current sequence position and/or automaton state depending on whether we considered u_{s+1}, and depending on whether we contributed to the corrected sequence we are pursuing.

And since we start at the initial state q_0, follow the transition function δ, and terminate at some accepting state $q_F \in F$ (i.e. we simulate the automaton $\mathcal{A}_r$ traversal), the resulting corrected sequences we are trying to build will then conform to r.

Note that all the described actions we could consider at this level (i.e. during the horizontal correction of a sequence u with respect to $\mathcal{C}_{t,n}$, i.e. during the processing of the described correction intent) actually represent other recursively nested correction intents from the vertical perspective, i.e. assignments for easier subproblems we can process exactly the same way.

An important obstacle is that the regular expression r from the selected grammar context $\mathcal{C}_{t,n}$ is based on the alphabet of nonterminals N, but labels of data tree nodes are from the alphabet of terminals T (to be more precise, from $\mathbb{E} \cup \{\texttt{data}\}$, because data trees may generally have nodes with labels that are not permitted by the grammar at all). So, we must be aware of the mutual relationship between both these sets of symbols.

For a given terminal t (label we come across in the node sequence u) there can generally exist more different active production rules in P_A, and thus different allowed nonterminals to which it can be mapped (since the class of regular tree grammars permits competing nonterminals within the regular expression r).

On the other hand and analogously, for a given nonterminal $n \in N_A$ (we come across in r) there can exist more terminals to which it can unfold (which is a possibility for all the classes of regular tree grammars anyway).

3.3.3 Correction Intents

The purpose of the following notion of *correction intents* is to encapsulate all the so far discussed ideas of the horizontal and vertical correction.

Roughly speaking, they describe an intended horizontal action at one level, and a vertical assignment and impact of this action on further recursive processing at the nested level.

Definition 3.10 (Correction Intents)**.** *Given a set of names for particular* intent types $\Omega = \{\texttt{correct}, \texttt{insert}, \texttt{delete}, \texttt{repair}, \texttt{rename}\}$*, we define a* correction intent *to be a tuple* $\mathcal{I} = (id, type, A, L)$ *with the following general structure:*

- $id \in \mathbb{N}_0^*$ *is an* intent identifier*.*
- $type \in \Omega$ *is an* intent type*.*
- $A = (p, e, v_I, v_E)$ *is an* intent action*:*
  - p *is a* base node*,*
  - e *is a repairing instruction,*

  - $v_I = (s_I,\ q_I)$*:* $s_I \in \mathbb{N}_0$ *is an* initial stratum *and* q_I *is an* initial state*, both together the* initial position*,*
  - $v_E = (s_E,\ q_E)$*:* $s_E \in \mathbb{N}_0$ *is an* ending stratum *and* q_E *is an* ending state *both together the* ending position*.*

- $L = (u,\ \mathcal{C},\ H,\ Y)$ *is an* intent assignment*:*
  - $u = \langle u_1,\ \ldots,\ u_k \rangle$*,* $k \in \mathbb{N}_0$ *is a sequence of nodes to be processed,*
  - $\mathcal{C}$ *is a grammar context to which* u *should conform,*
  - H *as a sequence of grammar contexts is an optional* context chain *that plays a special role in case of the* `insert` *intents,*
  - $Y \subseteq \Omega$ *is a set of the* allowed types *for nested correction intents.*

Intent action components describe the processing of the current sequence of sibling nodes. In particular, p is an optional reference to an existing data tree node (really as an existing node and not just a position) to which the given intent is associated, e is an optional repairing instruction implementing an intended action within the sequence (i.e. describing the intended edit operation), and, finally, v_I and v_E together represent the current progress of the node sequence processing and simulation of the automaton traversal.

Intent assignment components describe an impact and restrictions of the intent action on further recursive processing at the nested level. In particular, u is a sequence of nodes to be processed (usually child nodes of the node p, but not always), and C a grammar context to be used for this purpose. The remaining items have rather a technical meaning (but still important), so we will reveal them a bit later.

Last but not least, intent identifier *id* enables us to distinguish particular instances of correction intents among each other. Technically, these identifiers are implemented as words in $\mathbb{N}_0^*$ and characterize the recursive nesting of correction intents themselves. However, do not confuse these identifiers with positions or nodes of underlying trees, since they have nothing in common.

The definition itself, however, only provides us with a general structure that correction intents should in principle follow. In other words, we have just describe a universal interface of correction intents, which allows us to treat all the considered horizontal correction actions in a unified way, regardless we are about to insert a new subtree, remove and existing one, or repair it.

Given a particular data tree $\mathcal{T}$ to be corrected with respect to a grammar $\mathcal{G}$, we now provide a description of all the correction intents that we in particular (and not any other) derive in order to fulfill this correction goal.

A special position between all the correction intents has the *starting correction intent*, the only one of type `correct`. Its purpose is to initiate the entire correction of a provided data tree – i.e. to initiate the correction of the root node ϵ with respect to the starting grammar context $\mathcal{C}_\bullet$.

Definition 3.11 (Starting Correction Intent)**.** *Having a data tree $\mathcal{T} = (D, lab, val)$ and a regular tree grammar $\mathcal{G} = (N, T, S, P)$, we define $\mathcal{I}_\bullet = (id, type, A, L)$ to be a* starting correction intent*, where:*

- $id = \epsilon$.
- $type =$ `correct`.
- $A = \bot$, *i.e. is not defined.*
- $L = (u, \mathcal{C}, H, Y)$ *is an* intent assignment*:*
  - $u = \langle\epsilon\rangle$ *if D is not empty, else $u = \langle\rangle$,*
  - $\mathcal{C} = \mathcal{C}_\bullet = (r, N_A, P_A)$ *is the starting grammar context for $\mathcal{G}$ with a content model $r = r_\bullet$ and automaton $\mathcal{A}_{r_\bullet}$,*
  - $H = \bot$,
  - $Y = Y_{\texttt{correct}} = \Omega \setminus \{\texttt{correct}\}$.

Example 3.8. *Let us continue with our sample data tree $\mathcal{T}$, grammar $\mathcal{G}$ and already resolved starting grammar context $\mathcal{C}_\bullet$ from Example 3.7.*

Then the starting correction intent is equal to $\mathcal{I}_\bullet = (\epsilon,$ `correct`$, \bot, L)$, where $L = (\langle\epsilon\rangle, \mathcal{C}_\bullet, \bot, \Omega \setminus \{\texttt{correct}\})$.

Now we are able to continue and introduce all the remaining types of correction intents. For a given and already defined correction intent $\mathcal{I}$, the following definition describes all the recursively nested correction intents which we require to invoke from $\mathcal{I}$ in order to be able to evaluate $\mathcal{I}$ itself.

Definition 3.12 (Nesting of Correction Intents)**.** *Let $\mathcal{T} = (D, lab, val)$ be a data tree and $\mathcal{G} = (N, T, S, P)$ a regular tree grammar. Next, assume that $\mathcal{I} = (id, type, A, L)$ is an already defined correction intent with an assignment $L = (u, \mathcal{C}, H, Y)$, node sequence $u = \langle u_1, \ldots\ u_k\rangle$ for some $k \in \mathbb{N}_0$, grammar context $\mathcal{C} = (r, N_A, P_A)$, finite automaton $\mathcal{A}_r = (Q, N_A, \delta, q_0, F)$ for r, and, finally, context chain $H = \langle\mathcal{C}_{t^1,n^1}, \ldots, \mathcal{C}_{t^j,n^j}\rangle$ for some $j \in \mathbb{N}_0$.*

Given any traversal position *$v'_I = (s'_I, q'_I)$ with $s'_I \in \mathbb{N}_0$, $s'_I \leq k$, $q'_I \in Q$ (i.e. we are now at an automaton state q'_I of $\mathcal{A}_r$ and at a position number s'_I inside the processed sequence u), we now define right all the following* recursive correction intents *$\mathcal{I}' = (id', type', A', L')$ having an intent identifier id' described at the very end of this definition, and an intent type $type'$, action $A' = (p', e', v'_I, v'_E)$ and assignment $L' = (u', \mathcal{C}', H', Y')$ defined as follows:*

- *If* `insert` $\in Y$ *is allowed, then we define a new nested $\mathcal{I}'$ for each individual nonterminal $x \in N_A$ such that x is permitted right here (i.e. $\delta(q'_I, x)$ is defined) and for each active production rule $\mathcal{F}'$ matching x (i.e. $\mathcal{F}' = [t', r' \to n'] \in P_A$ where $n' = x$) and its terminal t', but only if the context chain H does not already contain $\mathcal{C}_{t',n'}$ (i.e. $\neg\,\exists\, i \in \mathbb{N}, 1 \leq i \leq j$ such that $\mathcal{C}_{t^i,n^i} = \mathcal{C}_{t',n'}$) as follows:*
  - $type' =$ `insert`.
  - $p' = \bot$,

  - $e' = \texttt{addLeaf}(t')$, *and*
  - $v'_E = (s'_I,\ \delta(q'_I, x))$.
  - $u' = \langle\rangle$,
  - $\mathcal{C}' = \mathcal{C}_{t',n'}$,
  - $H' = \langle \mathcal{C}_{t^1,n^1},\ \ldots,\ \mathcal{C}_{t^j,n^j},\ \mathcal{C}_{t',n'} \rangle = H.\langle \mathcal{C}_{t',n'} \rangle$, *i.e.* H *is appended by* $\mathcal{C}_{t',n'}$,
  - $Y' = Y_{\texttt{insert}} = \{\texttt{insert}\}$.

- *If* $\texttt{delete} \in Y$ *is allowed and we have not already processed the entire sequence* u *(i.e.* $s'_I < k$*), then we define right one new* $\mathcal{I}'$ *as follows:*
  - $type' = \texttt{delete}$.
  - $p' = u_{s'_I+1}$,
  - $e' = \texttt{removeLeaf}$, *and*
  - $v'_E = (s'_I + 1,\ q'_I)$.
  - $u' = \langle u_{s'_I+1}.0,\ \ldots,\ u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1) \rangle$,
  - $\mathcal{C}' = \mathcal{C}_\epsilon$,
  - $H = \bot$, *and*
  - $Y' = Y_{\texttt{delete}} = \{\texttt{delete}\}$.

- *If* $\texttt{repair} \in Y$ *is allowed and we have not already processed the entire sequence* u *(i.e.* $s'_I < k$*), then we define a new nested* $\mathcal{I}'$ *for each individual nonterminal* $x \in N_A$ *such that* x *is permitted right here (i.e.* $\delta(q'_I, x)$ *is defined) and for each active production rule* $\mathcal{F}'$ *matching* x *(i.e.* $\mathcal{F}' = [t',\ r' \to n'] \in P_A$ *where* $n' = x$*) and its terminal* t'*, but only if the label of the following node* $u_{s'_I+1}$ *is to be preserved (i.e.* $t' = lab(u_{s'_I+1})$*) as follows:*
  - $type' = \texttt{repair}$.
  - $p' = u_{s'_I+1}$,
  - $e' = \bot$, *and*
  - $v'_E = (s'_I + 1,\ \delta(q'_I, x))$.
  - $u' = \langle u_{s'_I+1}.0,\ \ldots,\ u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1) \rangle$,
  - $\mathcal{C}' = \mathcal{C}_{t',n'}$,
  - $H = \bot$, *and*
  - $Y' = Y_{\texttt{repair}} = \Omega \setminus \{\texttt{correct}\}$.

- *If* $\texttt{rename} \in Y$ *is allowed and we have not already processed the entire sequence* u *(i.e.* $s'_I < k$*), then we define a new nested* $\mathcal{I}'$ *for each individual nonterminal* $x \in N_A$ *such that* x *is permitted right here (i.e.* $\delta(q'_I, x)$ *is defined) and for each active production rule* $\mathcal{F}'$ *matching* x *(i.e.* $\mathcal{F}' = [t',\ r' \to n'] \in P_A$ *where* $n' = x$*) and its terminal* t'*, but only if a new intended label* t' *for the following node* $u_{s'_I+1}$ *is not the same as its current one (i.e.* $t' \neq lab(u_{s'_I+1})$*) as follows:*
  - $type' = \texttt{rename}$.

- $p' = u_{s'_I+1}$,
- $e' =$ `renameNode`(t'), *and*
- $v'_E = (s'_I + 1,\ \delta(q'_I, x))$.
- $u' = \langle u_{s'_I+1}.0,\ \ldots,\ u_{s'_I+1}.(fanOut(u_{s'_I+1}) - 1)\rangle$,
- $\mathcal{C}' = \mathcal{C}_{t',n'}$,
- $H = \bot$, *and*
- $Y' = Y_{\texttt{rename}} = \Omega \setminus \{\texttt{correct}\}$.

Next, for each of the above considered traversal positions $v'_I = (s'_I,\ q'_I)$, $s'_I \in \mathbb{N}_0$, $s'_I \leq k$, $q'_I \in Q$ *we define* $NestedIntents(\mathcal{I}, v'_I)$ *as a set of all the* nested *correction intents* invoked *by* $\mathcal{I}$ *from* v'_I, *i.e. all* $\mathcal{I}'$ *introduced above for* v'_I.

Then we define $NestedIntents(\mathcal{I})$ *as a set of all the nested correction intents invoked by* $\mathcal{I}$ *regardless a particular position* v'_I.

Finally, intent identifier id' *of each of the previously introduced nested correction intent* $\mathcal{I}'$ *is defined as* $id' = id.a$, *where* $a \in \mathbb{N}_0$ *is a uniquely assigned number capable of distinguishing all the intents in* $NestedIntents(\mathcal{I})$ *between each other.*

Though having described the allowed nesting of correction intents formally in the just finished definition, it may be useful to look at correction intents of all the individual intent types a bit closer and recall or discuss their main features and consequences right now.

In particular, we first try to describe the objective of such correction intent (i.e. describe corrections it should produce), then we discuss features and restrictions of the assignment on the further recursive processing, as well as we discuss changes of the traversal position the given action causes.

So, suppose we are processing a node sequence u with respect to a grammar context $\mathcal{C}$, all that within a correction intent $\mathcal{I}$. Being at a traversal position $v'_I = (s'_I,\ q'_I)$, i.e. having already processed s'_I nodes of u and being at an automaton state q'_I, we can derive the following nested correction intents.

We start with intents of the type `delete`, since their explanation is probably the most straightforward one. First of all, for a given node $u_{s'_I+1}$, there exists right one such `delete` correction intent. Its purpose is to completely remove $\mathcal{T}^{\Delta u_{s'_I+1}}$, i.e. the whole subtree rooted at $u_{s'_I+1}$. Since we have processed node $u_{s'_I+1}$ from the sequence u, but we have not decided to use it in the corrected node sequence we are pursuing to build, the sequence position increases from s'_I by 1, whilst the automaton state remains untouched and equal to q'_I.

The final question is, how we actually attain the outlined removal of the whole subtree $\mathcal{T}^{\Delta u_{s'_I+1}}$. We simply request processing of a sequence u' of the child nodes of $u_{s'_I+1}$ with respect to the empty grammar context $\mathcal{C}_\epsilon$. This ensures that all these child nodes will be removed recursively, since only other nested `delete` correction intents can be invoked according to $Y' = \{\texttt{delete}\}$. Having removed them, the last thing to do is to remove $u_{s'_I+1}$ itself. This action can now be performed safely, for $u_{s'_I+1}$ certainly must be a leaf node at this moment. For this purpose, the repairing instruction `removeLeaf` is prepared, being implicitly bound to $u_{s'_I+1}$ as a particular node from the original data tree $\mathcal{T}$.

Now we focus on correction intents of types `repair` and `rename`. They are very similar and analogous to each other, so we can discuss both of them together. Their purpose is to take the existing node $u_{s'_I+1}$, preserve it, and recursively process its subtree $\mathcal{T}^{\Delta u_{s'_I+1}}$ using any intent types except the `create` one.

Being at the traversal position $v'_I = (s'_I, q'_I)$, for each possible pair of a nonterminal x (such that $x \in N_A$ is allowed here and $\delta(q'_I, x)$ is defined as well) and terminal t' (such that there is an active production rule $\mathcal{F}' = [t', r' \to x] \in P_A$ that matches the given nonterminal x) we create right one nested correction intent $\mathcal{I}'$. If t' equals to the current label of $u_{s'_I+1}$, we choose the `repair` type and we simply preserve this label. Otherwise we choose the `rename` type and via a prepared repairing instruction `renameNode`(t') we change the label of $u_{s'_I+1}$ to t'.

For we have considered the node $u_{s'_I+1}$ from the processed sequence u and we have decided to preserve it, the sequence position increases from s'_I by 1, and the automaton state is changed accordingly to $\delta(q'_I, x)$.

Also note that each of the nested intents $\mathcal{I}'$ may generally represent more corrected data trees, so there is not just one possible solution as there was in case of the `delete` intents.

Finally, let us discuss the `insert` intent type. Its purpose is to insert a new subtree right at the current sequence position. Similarly to the both previous intent types, we once again create one nested correction intent $\mathcal{I}'$ for each of the allowed pairs of a nonterminal x and terminal t' symbols.

Before the insertion of such data tree can happen, we first must insert its root node, i.e. to apply the prepared `addLeaf`(t') repairing instruction. Only then its child nodes (if any) can recursively be added as well. For this purpose we request processing of an empty sequence u' with respect to a given grammar context $\mathcal{C}_{t',x}$, and allow the usage of additional nested intents only of the `insert` type. As a consequence, this approach ensures that all the minimal data trees are found.

At the base level, since we have not processed any node from u, but we have considered a newly inserted node into a corrected sequence we are building, the sequence u traversal position does not change at all, whereas the automaton state is changed accordingly to $\delta(q'_I, x)$.

The only difficulty with `insert` correction intents is that they generate entirely new nodes, while all the other types of correction intents only work with the existing ones (either by deleting them, renaming or leaving them untouched). So, we must further discuss how the recursive processing is terminated at the bottom in order to avoid generation of potentially infinite data trees.

Although nonterminal symbols (or more complicated subexpressions) that are wrapped by the iteration operator * in content models r can unfold to any number of repetitions, they do not cause any difficulties at this moment. On the other hand, recursive grammars (i.e. grammars with recursive production rules like the one in Example 2.8 because of an allowed nesting via a nonterminal B) calls for an attention now.

Fortunately enough we consider only grammars that are consistent, and so the recursive nesting of rules has to always be just an optional alternative (via the mentioned iteration operator *, or via the choice operator $|$ as well), not something that is forced. Otherwise we would not be able to stop and generate a finite data tree at all. And since our cost functions require costs to be positive,

it is always cheaper to avoid such optional recursion at all.

Precisely in this sense we have introduced the last component of correction intents, that is a *context chain*. Once we decide to insert a new node with a given grammar context, we forbid this context to be used once again anywhere within the further processing of the nested correction intents. As a consequence, we ensure that the algorithm can never fall into an infinite loop, though we do not affect which corrected data trees we are able to find at all.

Finishing the description of all the recursive types of correction intents, we shortly explain the purpose of intent identifiers as well.

Having a particular correction intent $\mathcal{I}$, then all its nested correction intents are different structures – and they are different even when we ignore values of their intent identifiers. It means that $NestedIntents(\mathcal{I})$ as a set really contains all the nested intents we introduced, and it would also contain them even if the intent structure itself did not involve such intent identifiers. Unfortunately, this natural distinguishing ability works only locally – not among correction intents that do not share the same parental intent.

Assume now we are given a data tree $\mathcal{T}$ to be corrected with respect to a grammar $\mathcal{G}$. Then let $I_0 = \{\mathcal{I}_\bullet\}$ be a set containing the starting correction intent only, and inductively for each $i \in \mathbb{N}$ let $I_i = \bigcup_{\mathcal{I} \in I_{i-1}} NestedIntents(\mathcal{I})$ be a set of all the correction intents invoked at the recursion depth i.

Then we can define $CreatedIntents(\mathcal{T}, \mathcal{G}) = \bigcup_{i \in \mathbb{N}_0} I_i$ to be a set of all the correction intents created during the whole process of the correction of data tree $\mathcal{T}$ according to grammar $\mathcal{G}$.

Note that without the artificially added intent identifiers (or another similar mechanism), such definition would not be possible at all. And so this is the reason, why we had to use them.

Just to illustrate at least a bit how the nested correction intents are derived in practice, let us continue with the following example.

Example 3.9. *Assume we want to process child nodes of the root node from our data tree $\mathcal{T}$. Suppose also that we have previously decided to preserve its label a, so the associated nonterminal is equal to A – and so we now want to process a sequence $u = \langle 0, 1, 2 \rangle$ under the grammar context $C_{a,A}$, i.e. we want this sequence to match a regular expression $r = C.D_A{}^*$ recognized by $\mathcal{A}_r$.*

Let $\mathcal{I}$ denote this correction intent. Then, the assignment of $\mathcal{I}$ is illustrated in Figure 3.4.

Now, assume we want to inspect all the possibilities at a traversal position $(0, 0)$ when processing $\mathcal{I}$. There we have the following three options to consider. We can attempt to insert a new node with label c via $\mathcal{I}_1$, delete node 0 with label x via $\mathcal{I}_2$ or rename it to c via $\mathcal{I}_3$:

$\mathcal{I}_1 = (0, \texttt{insert}, A_1, L_1)$ *where*
$A_1 = (\bot, \texttt{addLeaf}(c), (0,0), (0,1))$ *and*
$L_1 = (\langle\rangle, \mathcal{C}_{c,C}, \langle\mathcal{C}_{c,C}\rangle, \{\texttt{insert}\})$.

$\mathcal{I}_2 = (1, \texttt{delete}, A_2, L_2)$ *where*
$A_2 = (0, \texttt{removeLeaf}, (0,0), (1,0))$ *and*
$L_2 = (\langle 0.0\rangle, \mathcal{C}_\epsilon, \bot, \{\texttt{delete}\})$.

$\mathcal{I}_3 = (2,\ \texttt{rename},\ A_3,\ L_3)$ *where*
$A_3 = (0,\ \texttt{renameNode}(c),\ (0,0),\ (1,1))$ *and*
$L_3 = (\langle 0.0 \rangle,\ \mathcal{C}_{c,C},\ \bot,\ \Omega \setminus \{\texttt{correct}\})$.

All these three correction intents form a set $NestedIntents(\mathcal{I},\ (0,\ 0))$.

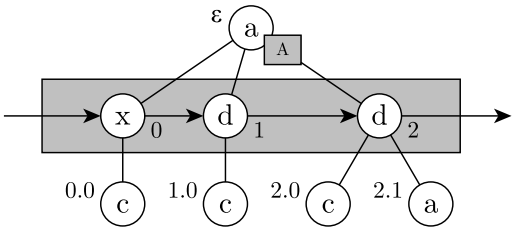


Figure 3.4: Assignment of correction intent $\mathcal{I}$ for $\mathcal{C}_{a,A}$ and ϵ from $\mathcal{T}$

Before we finally end this section and move toward other components of our correction model, it might be useful to summarize several features and important observations about correction intents as such.

From the logical point of view, correction intents allows us to split the problem of a data tree $\mathcal{T}$ correction into a set of subproblems, i.e. recursively nested correction intents dealing with problems that are easier to figure out. It means that when we are about to evaluate a particular correction intent $\mathcal{I}$, we only need to invoke all $NestedIntents(\mathcal{I})$ and appropriately use results we acquire from their evaluation in order to figure out solution for the intent $\mathcal{I}$ itself.

The first important feature of correction intents, as we have designed them, is that they allow us to conduct the correction of an entire data tree in a unified way, i.e. regardless we are processing the existing data tree nodes or generating new ones, the interface, behavior and internal principles of correction intents remain exactly the same.

When we recall the definition of correction intents, being at a particular traversal position $v'_I = (s'_I,\ q'_I)$, we always derive the nested correction intents $NestedIntents(\mathcal{I}, v'_I)$ exactly according to the transition function δ of the corresponding automaton $\mathcal{A}_r$ for a regular expression r – it means $\delta(q'_I, x)$ in particular for any of the allowed nonterminal symbols $x \in N_A$. Note that we do not use any specific features that would be guaranteed only by the Glushkov automata and the way they are constructed. As a consequence, we could use any other finite automaton capable of recognizing r, and the correction model would still be working without any modification required.

Moreover, we could also extend the model in a way that it would support nondeterministic finite automata as well. If the transition function δ of the deterministic automata is defined as a partial function $Q \times \Sigma \to Q$, in case of the nondeterministic ones it would be defined as $Q \times \Sigma \to \mathcal{P}(Q)$. This means that for a given traversal position $v'_I = (s'_I,\ q'_I)$ and an allowed nonterminal symbol $x \in N_A$, the transition $\delta(q'_I, x)$ can lead to an arbitrary number of states – none in case $\delta(q'_I, x) = \emptyset$, right one when $|\delta(q'_I, x)| = 1$, but even more than just 1 on the contrary to deterministic automata.

When we take these states into account and consider creating the nested intents for each and every one of them separately, the model immediately allows us to work even with nondeterministic finite automata. And as a consequence, with all regular expressions, not just those that are 1-unambiguous.

It is also worth of noting that we handle all nodes of the original data tree in the same way, regardless they are locally valid or not. In other words, the correction algorithm does not check the validity at first – and only if it was violated, it would invoke the correction procedures – instead, we traverse and correct the data tree directly and uniformly in all circumstances.

Next, we also do not modify original data trees in any way during the phase of processing of the correction intents – the algorithm just reads them, and so all the implicit bindings of repairing instructions to the existing data tree nodes remain untouched as well. Once the entire correction is completed, the corrected data trees can be obtained by unfolding the intent repair structure resulting from the starting correction intent $\mathcal{I}_\bullet$, as we have also already sketched.

Last but not least, though the definition of the allowed nesting of correction intents intrinsically determines how the provided edit operations are harnessed, and so how the resulting corrected data trees will look like, it is another question how the nested correction intents are actually explored, invoked and combined together. And yet another question would be, how to conduct this intent evaluation process efficiently enough.

3.4 Correction Multigraphs

The correction of a provided data tree $\mathcal{T}$ starts with the starting correction intent $\mathcal{I}_\bullet$ and recursively continues towards leaves by invoking other nested correction intents. However, we have still not yet described how to actually turn all these correction intents into life.

3.4.1 Correction Multigraphs

Assume that we now want to evaluate a particular correction intent $\mathcal{I}$, i.e. to create its intent repair structure $\mathcal{R}_\mathcal{I}$ that should represent all the minimal corrections of a node sequence $u = \langle u_1, \ldots, u_k \rangle$ from the assignment of $\mathcal{I}$ with respect to a grammar context $\mathcal{C}$.

Since the recursive nesting of intents must always reach its bottom, we can also suppose that all intents $\mathcal{I}'$ invoked both directly and indirectly from $\mathcal{I}$ are at this moment already evaluated and we are provided with the corresponding intent repair structures $\mathcal{R}_{\mathcal{I}'}$ for each such nested $\mathcal{I}'$, despite we still cannot fully reveal their definition. In other words, we can assume that we are provided with solutions of all the subproblems we identified in order to figure out the problem itself, i.e. to evaluate $\mathcal{I}$.

The only thing we need to know about these nested repairs $\mathcal{R}_{\mathcal{I}'}$ is that each of them is associated with an overall *correction cost* – the cost of acquiring all the corrected data trees it represents, i.e. the cost of all the edit sequences it encodes, even though expressed by repairing instructions.

The basic way of finding corrections of sequence u would be to dynamically traverse and inspect the state space of the corresponding automaton $\mathcal{A}_r$ – dynamically generating new suitable corrected sequences one by one, step by step, both to the width and depth.

In terms of our correction intents, we would only need to compose and mutually connect the nested correction intents into sequences via the corresponding traversal positions – starting at the automaton initial state and before u, then considering all the possibilities at each particular traversal position – and simply moving there and back again, while node by node generating and revealing all the possible corrected sequences one by one.

However, this approach would not be efficient enough. Therefore, we decided to implement another idea instead. That is to put all the nested correction intents together into a structure we call a *correction multigraph*, bind them to each other by the corresponding traversal positions once again, but represent all the sequence u correction possibilities statically this way.

So, let us provide a formal definition of this correction multigraph structure.

Definition 3.13 (Correction Multigraph)**.** *Let $\mathcal{T}$ be a data tree, $\mathcal{G}$ a grammar, and $\mathcal{I} = (id,\ type,\ A,\ L)$ a correction intent with an assignment $L = (u,\ \mathcal{C},\ H,\ Y)$, node sequence $u = \langle u_1,\ \ldots,\ u_k \rangle$ and finite automaton $\mathcal{A}_r = (Q,\ N_A,\ \delta,\ q_0,\ F)$ for r from grammar context $\mathcal{C}$.*

We define a correction multigraph *to be a tuple $\mathcal{M}_\mathcal{I} = (V,\ E,\ v_S,\ V_T)$, where:*

- *$(V,\ E)$ is a directed multigraph:*
  - *$V = \{(s, q) \mid s \in \mathbb{N}_0,\ 0 \leq s \leq k$ and $q \in Q\}$ is a set of* vertices*, where:*
    - *s is a* stratum number*, and*
    - *q is an automaton state within stratum s.*
  - *$E = \{(v_1,\ v_2,\ \mathcal{I}',\ \mathcal{R}_{\mathcal{I}'}) \mid$ both v_1 and $v_2 \in V$, $\mathcal{I}' \in NestedIntents(\mathcal{I})$, $\mathcal{I}' = (id',\ type',\ A',\ L')$, $A' = (p',\ e',\ v'_I,\ v'_E)$ such that $v_1 = v'_I$, $v_2 = v'_E$ and $\mathcal{R}_{\mathcal{I}'}$ is an intent repair for $\mathcal{I}'\}$ is a set of* edges*, where:*
    - *v_1 is an* initial vertex*,*
    - *v_2 is a* ending vertex*,*
    - *$\mathcal{I}'$ is the associated nested correction intent, and*
    - *$\mathcal{R}_{\mathcal{I}'}$ is the intent repair for $\mathcal{I}'$.*
- *$v_S = (0, q_0)$ is the* source vertex*, $v_S \in V$.*
- *$V_T = \{v_T \mid v_T = (k, q_T),\ q_T \in F\}$ is a set of the* target vertices*.*

Finally, for each vertex $v \in V$ we define $E_v^{in} = \{e \mid e \in E,\ e = (v_1,\ v_2,\ \mathcal{I}',\ \mathcal{R}_{\mathcal{I}'})$ and $v_2 = v\}$ as a set of ingoing edges *to v, and analogously $E_v^{out} = \{e \mid e \in E,\ e = (v_1,\ v_2,\ \mathcal{I}',\ \mathcal{R}_{\mathcal{I}'})$ and $v_1 = v\}$ as a set of* outgoing edges *from v.*

We first shortly describe the structure of a correction multigraph. It is worth noting that it is really a multigraph and not just an ordinary graph – because there can be loops (edges that are starting and ending at the same vertex) as well as multiple edges between the same pair of vertices.

Multigraph vertices as such directly correspond to traversal positions. We can easily divide them into *strata* – disjoint sets of vertices according to a number of already considered nodes from u, i.e. a sequence u traversing position. Vertices inside each stratum cover all the automaton $\mathcal{A}_r$ states. Edges, on the other hand, directly correspond to all the invoked nested correction intents.

Suppose now that we have a particular multigraph edge $(v_1, v_2, \mathcal{I}', \mathcal{R}_{\mathcal{I}'})$ with $v_1 = (s_1, q_1)$, $v_2 = (s_2, q_2)$, and that $\mathcal{I}' = (id', type', A', L')$ is its associated nested correction intent with an assignment $L' = (u', \mathcal{C}', H', Y')$. Then we can reference this particular edge in a more convenient way by adopting the following naming convention.

In case $type' = \texttt{delete}$ we can write $e^{\texttt{delete}}_{s_1,q_1 \to s_2,q_2}$, and in case of all the remaining recursive intent types $e^{type':t,n}_{s_1,q_1 \to s_2,q_2}$ where t and n correspond to $\mathcal{C}' = \mathcal{C}_{t,n}$. Note that we do not need to treat the `correct` intent type, as it can never appear as an edge in any correction multigraph. Also note that this naming convention is defined correctly, i.e. these names are capable of identifying all the edges within $\mathcal{M}_{\mathcal{I}}$ without conflicts.

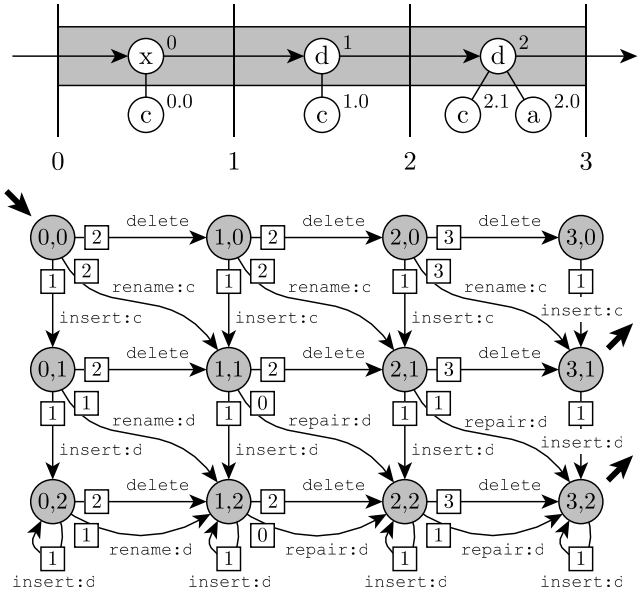


Figure 3.5: Correction multigraph $\mathcal{M}_{\mathcal{I}}$

Example 3.10. *Let us continue with Example 3.9, i.e. we want to correct a sequence $u = \langle 0, 1, 2 \rangle$ of our data tree $\mathcal{T}$ with respect to $\mathcal{C}_{a,A}$ with $r = C.D_A{}^*$. In that example we have also described all the nested correction intents $\mathcal{I}_1$, $\mathcal{I}_2$ and $\mathcal{I}_3$ invoked at a traversal position $(0, 0)$. These and all the other remaining intents invoked by $\mathcal{I}$ are the subject of this example, though we are not going to enumerate all of them.*

The corresponding correction multigraph $\mathcal{M}_{\mathcal{I}}$ for $\mathcal{I}$ is depicted in Figure 3.5. It has 4 strata (numbered from 0 to 3, exactly according to the length of u), each of which is formed by all the states of automaton $\mathcal{A}_r$, i.e. $\{0, 1, 2\}$.

To reference edges, we could afford to use even a more simplified convention in this case, since our grammar $\mathcal{G}$ is just a single type tree grammar, and so using of only a terminal symbol is sufficient for the identification too. Edges themselves are accompanied by their costs in square tags, i.e. correction costs of the intent repairs they are associated with.

Finally, the correction multigraph $\mathcal{M}_{\mathcal{I}} = (V,\ E,\ v_S,\ V_T)$ as such:

- $V = \{(0,0),\ (0,1),\ (0,2),\ (1,0),\ (1,1),\ \ldots,\ (3,1),\ (3,2)\}$ *is a set of vertices,*
- $E = \{((0,0),\ (0,1),\ \mathcal{I}_1,\ \mathcal{R}_{\mathcal{I}_1}),\ ((0,0),\ (1,0),\ \mathcal{I}_2,\ \mathcal{R}_{\mathcal{I}_2}),\ ((0,0),\ (1,1),\ \mathcal{I}_3,\ \mathcal{R}_{\mathcal{I}_3}),\ \ldots\} = \{e^{\texttt{insert}:c,C}_{0,0\to 0,1},\ e^{\texttt{delete}}_{0,0\to 1,0},\ e^{\texttt{rename}:c,C}_{0,0\to 1,1},\ \ldots\}$ *is a set of edges,*
- $v_S = (0,0)$ *is the source vertex, and*
- $V_T = \{(3,1),\ (3,2)\}$ *a set of the target vertices.*

Apparently, a correction multigraph $\mathcal{M}_{\mathcal{I}}$ for a correction intent $\mathcal{I}$ is just another representation of all the nested intents in $NestedIntents(\mathcal{I})$.

In the following example, we shortly outline how the situation looks like in case of the sample data tree root node correction.

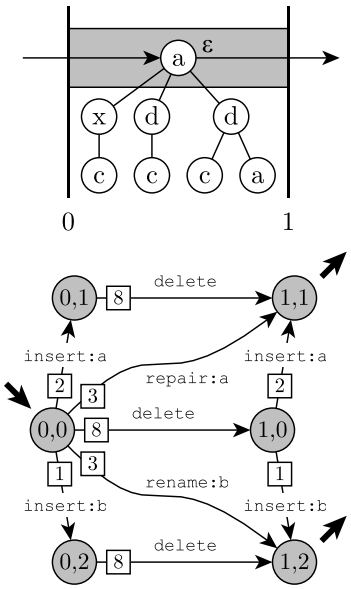


Figure 3.6: Correction multigraph $\mathcal{M}_{\mathcal{I}_\bullet}$

Example 3.11. *Analogously to the previous example, correction multigraph $\mathcal{M}_{\mathcal{I}_\bullet}$ for the starting correction intent $\mathcal{I}_\bullet$ is depicted in Figure 3.6.*

3.4.2 Correction Paths

The purpose of the correction multigraph is to represent all the possible corrections for a given sequence u of sibling nodes. However, how can we obtain these corrections? The answer is simple – we just need to focus on paths in this

multigraph – paths starting at the source vertex v_S and ending at any of the target vertices V_T – i.e. paths starting at stratum 0 (before the entire u) at the automaton initial state q_0 and ending at the last stratum (having the entire u processed) at any of the automaton accepting states F (thus producing a sequence conforming to a given regular expression r).

So, let us now describe these paths formally. We start with paths in correction multigraphs in general. Then we define, which of these paths are the correction paths. Finally, which correction paths are the shortest correction paths – for we are interested in finding right the minimal corrections only.

Definition 3.14 (Correction Paths)**.** *Let $\mathcal{M}_\mathcal{I} = (V, E, v_S, V_T)$ be a correction multigraph. Given $x, y \in V$, we define a* path *from x to y to be a sequence $p_{x,y} = \langle e_1, \ldots, e_n \rangle$ of edges with* length *$n \in \mathbb{N}_0$ such that:*

- *Let first $\forall\, k \in \mathbb{N}$, $1 \leq k \leq n$: $e_k = (v_1^k, v_2^k, \mathcal{I}^k, \mathcal{R}_{\mathcal{I}^k}^k)$, $e_k \in E$ and c^k is an overall cost of $\mathcal{R}_{\mathcal{I}^k}^k$.*
- *If $n > 0$, then we require that $v_1^1 = x$ and $v_2^n = y$; if $n = 0$, then $x = y$. Next, $\forall\, k \in \mathbb{N}$, $1 \leq k < n$: $v_2^k = v_1^{k+1}$, i.e. edges are incident to each other.*
- *$\neg\, \exists\, j,\ k \in \mathbb{N}$, $1 \leq j < k \leq n$: $v_1^j = v_1^k$ or $v_2^j = v_2^k$ or $v_1^j = v_2^j$, i.e. vertices do not repeat within a path.*

Given a vertex $v \in V$, we say that $v \in p_{x,y}$, if $\exists\, k \in \mathbb{N}$, $1 \leq k \leq n$ such that $v = v_1^k$ or $v = v_2^k$. If $p_{x,y} = \langle\rangle$, then we put both $x, y \in p_{x,y}$. Analogously, given an edge $e \in E$, we say that $e \in p_{x,y}$, if $\exists\, k \in \mathbb{N}$, $1 \leq k \leq n$ such that $e = e_k$.

We say that path $p_{x,y}$ is a correction path*, if is starts at the source vertex (i.e. $x = v_S$) and ends at any of the target vertices (i.e. $y \in V_T$).*

Finally, the cost *of path $p_{x,y}$ is defined as $cost(p_{x,y}) = \sum_{k=1}^{n} c^k$, i.e. as a sum of the costs of associated intent repairs on all the edges involved in the path.*

Note that the requirement on multigraph paths to be simple (i.e. that their vertices and, therefore, neither edges) might seem to be too strong. After all, certain multigraph *walks* from the source vertex to the target vertices could also potentially represent suitable corrections. However, we are searching for the minimal corrections only. And since we assumed that costs of edit operations are positive, we can focus on simple paths only without loss of any potential corrections.

The purpose of the following definition is to introduce the shortest paths within our correction multigraphs.

Definition 3.15 (Shortest Paths)**.** *Let $\mathcal{M}_\mathcal{I} = (V, E, v_S, V_T)$ be a correction multigraph, and $p_{x,y} = \langle e_1, \ldots, e_n \rangle$ a path of length $n \in \mathbb{N}_0$ from a vertex $x \in V$ to a vertex $y \in V$.*

We say that $p_{x,y}$ is the shortest path*, if and only if $\neg\, \exists\, p'_{x,y}$ such that it would have a lower cost, i.e. such that $cost(p'_{x,y}) < cost(p_{x,y})$.*

Next, by $P_{x,y}$ we denote a set of all the paths from x to y, by $P_{x,y}^{min}$ a set of all the shortest paths from x to y, and by $cost(P_{x,y}^{min})$ a cost of some (any) path in $P_{x,y}^{min}$.

Finally, given a nonempty set of vertices $Z \subseteq V$, let $m = \min_{z \in Z} cost(P_{x,z}^{min})$ be a minimal cost over all the shortest paths to all vertices from Z. Then we define

$P^{min}_{x,Z} = \{p \mid \exists\, z \in Z,\ p \in P^{min}_{x,z}$ *and* $cost(p) = m\}$ *as a set of all the shortest paths from* x *ending somewhere in* Z *and having this minimal cost. Without any surprise,* $cost(P^{min}_{x,Z}) = m$.

As an apparent consequence of our definition, there may generally exist more shortest paths between the given pair of vertices and not just one, yet we decided to use a term *shortest* and not *minimal*.

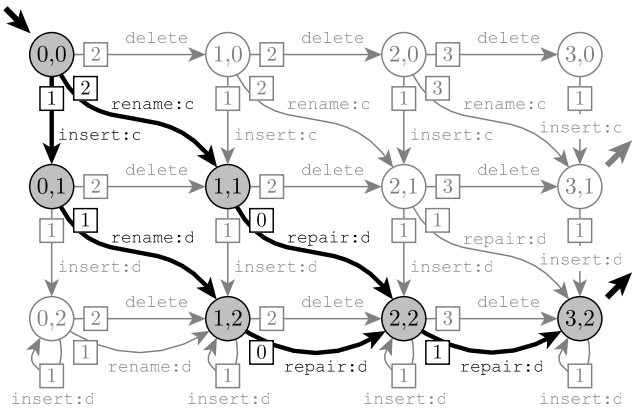


Figure 3.7: Shortest correction paths $P^{min}_{v_S,V_T}$ in multigraph $\mathcal{M}_{\mathcal{I}}$

Example 3.12. *The shortest paths in correction multigraph* $\mathcal{M}_{\mathcal{I}} = (V,\ E,\ v_S,\ V_T)$ *from Example 3.10 are highlighted in Figure 3.7. They are:*

$$p_1 = \langle e^{\texttt{insert}:c,C}_{0,0\to0,1},\ e^{\texttt{rename}:d,D_A}_{0,1\to1,2},\ e^{\texttt{repair}:d,D_A}_{1,2\to2,2},\ e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\rangle, \text{ and}$$
$$p_2 = \langle e^{\texttt{rename}:c,C}_{0,0\to1,1},\ e^{\texttt{repair}:d,D_A}_{1,1\to2,2},\ e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\rangle.$$

In other words, $P^{min}_{v_S,V_T} = \{p_1,\ p_2\}$. *The cost of each of these paths is equal to* 3. *Last but not least, notice that both these paths end at the accepting state* 2 *at the last stratum, none of them at state* 1.

In the previous example we have just highlighted that we did not use all the target vertices V_T. Notice that it is correct, for we are really interested only in the minimal cost $\min_{v_T \in V_T} cost(P^{min}_{v_S,v_T})$ over all the target vertices, and so $P^{min}_{v_S,V_T}$, not the shortest paths to each and every one of them separately.

Example 3.13. *Analogously, the shortest paths in the correction multigraph* $\mathcal{M}_{\mathcal{I}_\bullet}$ *for the starting correction intent* $\mathcal{I}_\bullet$ *are depicted in Figure 3.8.*

On the contrary to the previous example and the multigraph $\mathcal{M}_{\mathcal{I}}$, *here, all the target vertices of* $\mathcal{M}_{\mathcal{I}_\bullet}$ *are involved in the shortest correction paths.*

One of the fundamental questions we could ask is whether there always exists at least one shortest correction path in $P^{min}_{v_S,V_T}$. The answer is yes because we assumed only consistent grammars where all the reachable production rules are not useless. In case of grammars that are inconsistent or contain useless reachable rules, there may exist correction multigraphs with no shortest correction paths. And though it is generally possible to detect such types of grammars as explained

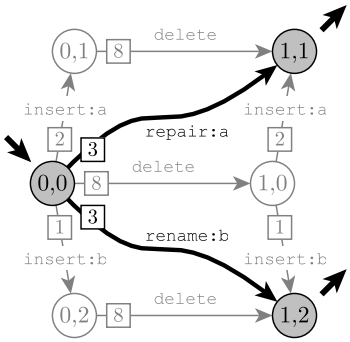


Figure 3.8: Shortest correction paths $P^{min}_{v_S,V_T}$ in multigraph $\mathcal{M}_{\mathcal{I}_\bullet}$

by Lu et al. [54], it is not necessary to do so explicitly, because our model can be extended in a way that it becomes capable of detecting and handling such grammars inherently as well.

First, whenever none of the target vertices is reachable in a correction multigraph $\mathcal{M}_{\mathcal{I}'}$ for a particular correction intent $\mathcal{I}'$, it means whenever $P^{min}_{v_S,V_T} = \emptyset$, then the intent repair for such $\mathcal{I}'$ becomes undefined, i.e. $\mathcal{R}_{\mathcal{I}'} = \bot$. Second, whenever a correction intent $\mathcal{I}$ requests a nested intent $\mathcal{I}' \in NestedIntents(\mathcal{I})$ such that $\mathcal{R}_{\mathcal{I}'} = \bot$, then the corresponding edge for $\mathcal{I}'$ in a correction multigraph $\mathcal{M}_{\mathcal{I}}$ is not considered when searching for the shortest correction paths.

When these two measures are adopted, i.e. when the nonexistence of the shortest correction paths is detected on one hand and edges for nested intents with undefined repairs are ignored, the model starts to support even inconsistent grammars or grammars with reachable useless production rules, i.e. all the regular tree grammars without any additional assumptions.

However, in order to make the definitions and algorithms easier to describe, we decided to preserve our original assumption in the further text, yet only for technical reasons as explained.

Now we focus on another, more practical problem. First of all, having a given correction multigraph constructed and its nested intents evaluated, we need to be able to find such shortest correction paths. And once we have them, we would also appreciate if we were able to represent them efficiently as well. It means on top of a correction multigraph structure, and not as a set of enumerated paths with unfolded edges as we have defined $P^{min}_{v_S,V_T}$.

Fortunately enough, the following pair of tracing functions enables us to deal with both these tasks.

Definition 3.16 (Path Tracing Functions)**.** *Let $\mathcal{M}_{\mathcal{I}} = (V,\ E,\ v_S,\ V_T)$ be a correction multigraph for a correction intent $\mathcal{I}$. We define the following two* tracing functions *for $\mathcal{M}_{\mathcal{I}}$:*

- *pDistance is a function $V \to \mathbb{R}_0^+ \cup \{\bot\}$ that assigns to each $v \in V$ its* distance *from the source vertex v_S, i.e. $pDistance(v) = cost(P^{min}_{v_S,v})$. In case there is no path between the source vertex v_S and v, i.e. $P^{min}_{v_S,v} = \emptyset$, we put $pDistance(v) = \bot$.*

- *pPredecessors is a function* $V \to \mathcal{P}(E)$ *that assigns to each* $v \in V$ *a set of all edges ingoing to* v *that are involved in the shortest paths* $P^{min}_{v_S,v}$ *to this* v*, i.e.* $pPredecessors(v) = \bigcup_{p \in P^{min}_{v_S,v},\ p=\langle e_1,\ldots,e_n\rangle,\ n\in\mathbb{N}_0,\ n>0} \{e_n\}$.

How to use these two tracing functions to actually obtain $P^{min}_{v_S,V_T}$? First, let $unfold(v_S) = \{\langle\rangle\}$ for the source vertex v_S. Then for any other $v \in V$, $v \neq v_S$ inductively $unfold(v) = \bigcup_{e \in pPredecessors(v),\ e=(v_1,v,\mathcal{I},\mathcal{R}_\mathcal{I})} \{p.\langle e\rangle \mid p \in unfold(v_1)\}$, for which we introduce a shorter notation $\bigcup_{e \in pPredecessors(v),\ e=(v_1,v,\mathcal{I},\mathcal{R}_\mathcal{I})} unfold(v_1).\langle e\rangle$ with the equivalent meaning. As a consequence, $unfold(v) = P^{min}_{v_S,v}$.

Assuming that $V^A_T = \{v_T \mid v_T \in V_T$ such that $cost(P^{min}_{v_S,v_T}) = cost(P^{min}_{v_S,V_T})\}$ is a set of all the target vertices that are involved in the shortest correction paths to V_T, and $paths = (pDistance, pPredecessors)$ encapsulates both the tracing functions, we define $unfold(paths) = \bigcup_{v_T \in V^A_T} unfold(v_T)$. As a consequence, $unfold(paths) = P^{min}_{v_S,V_T}$, which was exactly our goal.

Example 3.14. *To illustrate how unfolding of the tracing functions pDistance and pPredecessors work in practice, let us return to the shortest correction paths* $P^{min}_{v_S,V_T} = \{p_1, p_2\}$ *for* $\mathcal{M}_\mathcal{I}$ *we identified in the previous example.*

However, to make this example shorter, we only focus on those vertices of $\mathcal{M}_\mathcal{I}$*, about which we already know that they are involved in* $P^{min}_{v_S,V_T}$*. So, we have:*

$pDistance((0,0)) = 0$, $pPredecessors((0,0)) = \{\}$.
$pDistance((0,1)) = 1$, $pPredecessors((0,1)) = \{e^{\texttt{insert}:c,C}_{0,0\to0,1}\}$.
$pDistance((1,1)) = 2$, $pPredecessors((1,1)) = \{e^{\texttt{rename}:c,C}_{0,0\to1,1}\}$.
$pDistance((1,2)) = 2$, $pPredecessors((1,2)) = \{e^{\texttt{rename}:d,D_A}_{0,1\to1,2}\}$.
$pDistance((2,2)) = 2$, $pPredecessors((2,2)) = \{e^{\texttt{repair}:d,D_A}_{1,2\to2,2}, e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\}$.
$pDistance((3,2)) = 3$, $pPredecessors((3,2)) = \{e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\}$.

Then for all the considered vertices we evaluate the $unfold$ *function as follows:*

$unfold((0,0)) = \{\langle\rangle\}$.

$unfold((0,1)) = unfold((0,0)).\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}\rangle = \{\langle\rangle\}.\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}\rangle = \{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}\rangle\}$.

$unfold((1,1)) = unfold((0,0)).\langle e^{\texttt{rename}:c,C}_{0,0\to1,1}\rangle = \{\langle\rangle\}.\langle e^{\texttt{rename}:c,C}_{0,0\to1,1}\rangle = \{\langle e^{\texttt{rename}:c,C}_{0,0\to1,1}\rangle\}$.

$unfold((1,2)) = unfold((0,1)).\langle e^{\texttt{rename}:d,D_A}_{0,1\to1,2}\rangle = \{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}\rangle\}.\langle e^{\texttt{rename}:d,D_A}_{0,1\to1,2}\rangle = \{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}\rangle\}$.

$unfold((2,2)) = unfold((1,2)).\langle e^{\texttt{repair}:d,D_A}_{1,2\to2,2}\rangle \cup unfold((1,1)).\langle e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\rangle =$
$\{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}\rangle\}.\langle e^{\texttt{repair}:d,D_A}_{1,2\to2,2}\rangle \cup \{\langle e^{\texttt{rename}:c,C}_{0,0\to1,1}\rangle\}.\langle e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\rangle =$
$\{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}, e^{\texttt{repair}:d,D_A}_{1,2\to2,2}\rangle\} \cup \{\langle e^{\texttt{rename}:c,C}_{0,0\to1,1}, e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\rangle\} =$
$\{\langle e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}, e^{\texttt{repair}:d,D_A}_{1,2\to2,2}\rangle, \langle e^{\texttt{rename}:c,C}_{0,0\to1,1}, e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\rangle\}$.

$unfold((3,2)) = unfold((2,2)).\langle e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\rangle = \ldots = \{p_1, p_2\}$ *for:*
$p_1 = \langle e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}, e^{\texttt{repair}:d,D_A}_{1,2\to2,2}, e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\rangle$, *and*
$p_2 = \langle e^{\texttt{rename}:c,C}_{0,0\to1,1}, e^{\texttt{repair}:d,D_A}_{1,1\to2,2}, e^{\texttt{repair}:d,D_A}_{2,2\to3,2}\rangle$.

Finally, since we know that $V_T^A = \{(3, 2)\}$ *is a set of all the involved target vertices, we can finally write that:*

$unfold(paths) = unfold((3, 2)) = \{p_1, p_2\} = P^{min}_{v_S,V_T}$.

At this moment we have successfully introduced correction multigraphs as well as shortest correction paths within them, hence we are provided with all the notions we need to be able to transform the problem of evaluating correction intents into the problem of finding shortest paths.

In particular, given a correction intent $\mathcal{I}$ together with its correction multigraph $\mathcal{M}_\mathcal{I} = (V, E, v_S, V_T)$, we only need to find all the shortest correction paths from the source vertex v_S to any of the target vertices V_T, i.e. to acquire $P^{min}_{v_S,V_T}$.

And though this problem of searching for the shortest paths might seem to be a simple one, we actually need to dedicate the whole following chapter to discuss, how the traditional path searching algorithms can in particular be utilized in our correction model to really be able to search for the correction paths efficiently enough.

Therefore, we first finish the description of the model itself, and return to the correction algorithms themselves later on.

3.5 Intent Repairs

In this section we describe the last fundamental notion of the correction model we proposed. In particular, we first provide a definition of an intent repair structure and components it comprises of, and then we demonstrate how this intent repair structure can be unfolded to obtain the sequences of edit operations we are searching for.

3.5.1 Repair Structures

After the correction multigraph $\mathcal{M}_\mathcal{I}$ for intent $\mathcal{I}$ is constructed, its nested correction intents evaluated and all the shortest correction paths $P^{min}_{v_S,V_T}$ explored, we can compactly store them in a structure we call a *sequence repair*.

Definition 3.17 (Sequence Repair)**.** *Given an intent* $\mathcal{I}$ *and its correction multigraph* $\mathcal{M}_\mathcal{I} = (V, E, v_S, V_T)$ *with a set* $P^{min}_{v_S,V_T}$ *of all the shortest correction paths represented by pDistance and pPredecessors functions, we define a* sequence repair *for* $\mathcal{I}$ *to be a tuple* $\mathcal{N}_\mathcal{I} = (V', E', v_S, V_T, paths, c)$*, where:*

- (V', E') *is a* subgraph *of* (V, E) *such that:*
  - $V' = \{v \mid v \in V, \exists p \in P^{min}_{v_S,V_T}, v \in p\}$,
  - $E' = \{e \mid e \in E, \exists p \in P^{min}_{v_S,V_T}, e \in p\}$;
- $paths = (pDistance, pPredecessors)$ *such that* $unfold(paths) = P^{min}_{v_S,V_T}$;
- $c = cost(P^{min}_{v_S,V_T})$.

The idea behind a sequence repair $\mathcal{N}_\mathcal{I}$ is simple. We need to preserve relevant parts of the correction multigraph $\mathcal{M}_\mathcal{I}$, i.e. those parts (vertices and edges) that

are directly involved in the shortest correction paths $P^{min}_{v_S,V_T}$. Other parts of the multigraph can be (but may not be) pruned and thrown away.

Note that we actually only need to have a reasonably efficient access to the shortest correction paths – as sequences of edges as they were defined – and not because of the intent evaluation itself, but because of the intent repair translation that does not need to happen at all.

Example 3.15. *Continuing with Example 3.12 and the correction mutligraph $\mathcal{M}_\mathcal{I}$ for $\mathcal{I}$ with the already resolved shortest correction paths $P^{min}_{v_S,V_T} = \{p_1, p_2\}$, we can derive the sequence repair $\mathcal{N}_\mathcal{I} = (V', E', v_S, V_T, paths, c)$ as follows:*

- $V' = \{(0,0), (0,1), (1,1), (1,2), (2,2), (3,2)\}$ *is a set of involved vertices,*
- $E' = \{e^{\texttt{insert}:c,C}_{0,0\to0,1}, e^{\texttt{rename}:d,D_A}_{0,1\to1,2}, e^{\texttt{repair}:d,D_A}_{1,2\to2,2}, e^{\texttt{repair}:d,D_A}_{2,2\to3,2}, e^{\texttt{rename}:c,C}_{0,0\to1,1}, e^{\texttt{repair}:d,D_A}_{1,1\to2,2}\}$ *a set of involved edges, and*
- $c = 3$ *as the correction cost.*

Both the tracing functions pDistance and pPredecessors were discussed as well in Example 3.14 – even though just partly right there, actually right in the extent covered by the shortest correction paths themselves.

Having constructed a sequence repair $\mathcal{N}_\mathcal{I}$ to represent all the possible corrections of a node sequence u from $\mathcal{I}$, we are finally able to define an *intent repair* structure.

Its purpose is to encapsulate the discussed sequence repair (that results from the evaluation of the intent assignment of $\mathcal{I}$) together with an optional repairing instruction (that is prepared in the intent action of $\mathcal{I}$ from the very beginning) – to represent all the possible corrections of the entire correction intent $\mathcal{I}$.

Definition 3.18 (Intent Repair)**.** *Let $\mathcal{N}_\mathcal{I} = (V, E, v_S, V_T, paths, c)$ be a sequence repair for a correction intent $\mathcal{I} = (id, type, A, L)$ with an intent action $A = (p, e, v_I, v_E)$. An* intent repair *for $\mathcal{I}$ is a tuple $\mathcal{R}_\mathcal{I} = (e, \mathcal{N}_\mathcal{I}, cost)$, where:*

- *e is an optional repairing instruction,*
- *$\mathcal{N}_\mathcal{I}$ is the sequence repair for $\mathcal{I}$,*
- *$cost = cost(e) + c$ is the overall* correction cost*; in case the repairing instruction e is not defined, we assume that $cost(e) = 0$.*

A correction multigraph at the bottom of the recursive nesting of correction intents has no edges outgoing from its source vertex and this source vertex is also one of the target vertices at the same time, so there exists right one shortest path, i.e. a path with no edges and of a cost equal to 0.

Such empty path can easily be encapsulated to a given sequence repair and then also to an intent repair. Therefore, the recursive nesting of intent repair structures as such is defined correctly.

Example 3.16. *Having the correction intent $\mathcal{I}$ from Example 3.9 and its sequence repair $\mathcal{N}_\mathcal{I}$ from Example 3.15, we can now derive the intent repair $\mathcal{R}_\mathcal{I}$ for $\mathcal{I}$ and obtain $\mathcal{R}_\mathcal{I} = (\perp, \mathcal{N}_\mathcal{I}, 3)$.*

Now we are finally ready to clarify the mutual relationship of the most essential notions we have so far discussed in this chapter.

Whereas correction intents describe problems to be figured out using other nested correction intents as easier subproblems. Correction multigraphs, on the other hand, are structures that allow us to evaluate assignments of such correction intents effectively by transforming the problem of finding all the minimal corrected data trees to the problem of finding the shortest correction paths. Intent repairs, finally, represent recursively nested structures that allow us to store results of the entire intent evaluation process in a compact way.

However, we still have not yet answered one final question. That is, how to actually obtain the required sequences of edit operations from the evaluated intent repairs.

3.5.2 Translation of Repairs

The correction model is based on edit operations through which we are able to transform invalid data trees into valid ones. On the other hand, intent repairs are complex recursively nested structures that are based on pruned correction multigraphs and repairing instructions with not yet explicitly resolved bindings to data tree nodes.

The aim of the following text is to describe, how these intent repairs can be translated into particular sequences of edit operations, so that we obtain all the valid corrected data trees we are searching for from the very beginning.

The idea of this translation process is straightforward. And if we realize that correction intents, correction multigraphs and intent repairs are all recursively nested, there should be nothing strange on introducing this translation inductively as well.

So, we first describe how to translate a repairing instruction into a standalone edit operation. Then we discuss how to unfold a sequence repair into sequences of edit operations. Finally we need to discuss the translation of a whole intent repair itself. For this purpose we only need to learn how to combine the translation of a repairing instruction from the intent action together with a translation of a sequence repair resulting from the evaluation of the intent assignment.

We start with the translation of repairing instructions.

Definition 3.19 (Repairing Instruction Translation)**.** *Given a repairing instruction e, we define its* translation *to an edit operation $fix(e)$ as follows:*

- *If* $e = \texttt{addLeaf}(a)$ *for some* $a \in \mathbb{E} \cup \{\texttt{data}\}$*, then* $fix(e) = addLeaf(0, a)$.
- *If* $e = \texttt{removeLeaf}$*, then* $fix(e) = removeLeaf(0)$.
- *If* $e = \texttt{renameNode}(a)$*, then* $fix(e) = renameNode(0, a)$.

Note that the position parameters assigned during the translation of repairing instructions are just tentative – their final values will be fully resolved and adjusted later on.

For this purpose we need three auxiliary functions using which we can achieve such modifications. So, let us now introduce them.

Given a node $u \in \mathbb{N}_0^*$ and a constant $c \in \mathbb{N}_0$, we define $modPrepend(u, c) = c.u$ as a function that prepends a new first symbol c to u. If $u \neq \epsilon$, $u = i.v$ for some $i \in \mathbb{N}_0$ and $v \in \mathbb{N}_0^*$, then we define $modShiftRight(i.v, c) = (i + c).v$ as a function that shifts u at its first symbol by c to the right. Finally we define $modCut(i.v) = v$ as a function that truncates the first symbol from u.

All these functions have just been defined over node positions, but we can easily extend their definition on edit operations (we apply the given function on a position parameter of a given edit operation), sequences of edit operations (we apply the given function on all edit operations contained in a given sequence), and sets of sequences of edit operations as well (we apply the given function on all sequences in a given set).

Example 3.17. *First of all, let us look at a few simple examples of behavior of these auxiliary functions on node positions:*

$$modPrepend(2.1, 0) = 0.2.1,$$
$$modShiftRight(2.1, 1) = 3.1 \text{ and}$$
$$modCut(2.1) = 1.$$

Next, using the introduced extensions of these basic functions, we can, for example, write that:

$$\begin{aligned} &modPrepend(\{\langle removeLeaf(3.1)\rangle\}, 0) \\ &\quad = \{modPrepend(\langle removeLeaf(3.1)\rangle, 0)\} \\ &\quad = \{\langle modPrepend(removeLeaf(3.1), 0)\rangle\} \\ &\quad = \{\langle removeLeaf(modPrepend(3.1, 0))\rangle\} \\ &\quad = \{\langle removeLeaf(0.3.1)\rangle\}. \end{aligned}$$

Now, we focus on the translation of sequence repairs. For this purpose we have to process each shortest path separately by fetching, adjusting and combining edit sequences representing corrections of the nested intent repairs on their edges (though we do not know how to construct them yet).

Definition 3.20 (Correction Path Translation)**.** *Let $p = \langle e_1, \ldots, e_m\rangle$ for some $m \in \mathbb{N}_0$ be a path in a correction multigraph $\mathcal{M}_\mathcal{I} = (V, E, v_S, V_T)$ or a sequence repair $\mathcal{N}_\mathcal{I} = (V', E', v_S, V_T, paths, c)$ for a correction intent $\mathcal{I}$.*

Let furthermore $\forall\, i \in \mathbb{N}$, $1 \leq i \leq m$, $e_i = (v_1^i, v_2^i, \mathcal{I}^i, \mathcal{R}_{\mathcal{I}^i})$, c^i be the correction cost of $\mathcal{R}_{\mathcal{I}^i}$, $t^i \in \Omega$ be a type of $\mathcal{I}^i$, and last but not least $fix(\mathcal{R}_{\mathcal{I}^i})$ be the already resolved translation of the intent repair $\mathcal{R}_{\mathcal{I}^i}$.

Then starting with $a_p^0 = 0$, we successively process all the edges of p and put for each corresponding $i \in \{1, \ldots, m\}$:

- $X_p^i = modShiftRight(fix(\mathcal{R}_{\mathcal{I}^i}), a_{i-1})$,
- $a_p^i = a_p^{i-1} + 1$ *for* $t^i \in \{$`insert`, `repair`, `rename`$\}$, *and*
  $a_p^i = a_p^{i-1} + 0$ *for* $t^i \in \{$`delete`$\}$.

Having evaluated all X_p^i, we define a correction path translation *for p as* $fix(p) = \{\langle x_p^1, x_p^2, \ldots, x_p^m\rangle \mid \forall\, i \in \mathbb{N}, 1 \leq i \leq m, x_p^i \in X_p^i\}$.

The idea of the translation of correction paths is straightforward. We iterate over its edges, fetch the translations $fix(\mathcal{R}_{\mathcal{I}^i})$ of the nested intent repairs $\mathcal{R}_{\mathcal{I}^i}$, and adjust the position parameters of all the edit operations they contain accordingly, while putting them into X_p^i. To perform these adjustments correctly, we only need to be aware of the overall number of nodes we involved in the corrected node sequence we are producing at this level, i.e. to increment a_p^i appropriately depending on types t^i of the nested intents.

Once all X_p^i are resolved, we just consider all the possible combinations of edit sequences they contain to finally form the required path translation $fix(p)$. This is necessary, since each edge may generally represent more possible corrections, i.e. each $fix(\mathcal{R}_{\mathcal{I}^i})$ may contain more possible edit sequences, and we must consider all of them independently with respect to all the remaining edges.

Note that the path translation process is the very place where the involved repairing instructions become associated to particular node positions, i.e. right here the implicit bindings hidden within the correction paths themselves are turned into the explicit bindings (though final only from the perspective of this level of recursive nesting, i.e. relatively to sibling nodes only).

Now we finish the translation of the whole sequence repair structure. To fulfill this objective, we look at all the involved shortest correction paths ant put their translations together.

Definition 3.21 (Sequence Repair Translation)**.** *Let $\mathcal{N}_{\mathcal{I}} = (V, E, v_S, V_T, paths, c)$ be a sequence repair for a correction intent $\mathcal{I}$, and $unfold(paths) = P^{min}_{v_S,V_T}$ all the shortest correction paths. Then we define $fix(\mathcal{N}_{\mathcal{I}}) = \bigcup_{p \in P^{min}_{v_S,V_T}} fix(p)$ to be a* sequence repair translation *for $\mathcal{N}_{\mathcal{I}}$.*

Example 3.18. *Suppose we want to translate our sequence repair $\mathcal{N}_{\mathcal{I}}$ from Example 3.15. This means that we first need to translate both the shortest correction paths p_1 and p_2 from Example 3.12.*

For the first path p_1 we can successively derive:

$$\begin{aligned} X^1_{p_1} &= \{\langle addLeaf(0,c)\rangle\},\ a^1_{p_1} = 1,\\ X^2_{p_1} &= \{\langle renameNode(1,d)\rangle\},\ a^2_{p_1} = 2,\\ X^3_{p_1} &= \{\langle\rangle\},\ a^3_{p_1} = 3,\ \text{and}\\ X^4_{p_1} &= \{\langle renameNode(3.1,c)\rangle, \langle removeLeaf(3.1)\rangle\}\ \text{and}\ a^4_{p_1} = 4. \end{aligned}$$

So, path p_1 translates to $fix(p_1) = \{S_1, S_2\}$, where:

$$\begin{aligned} S_1 &= \langle addLeaf(0,c),\ renameNode(1,d),\ renameNode(3.1,c)\rangle,\ \text{and}\\ S_2 &= \langle addLeaf(0,c),\ renameNode(1,d),\ removeLeaf(3.1)\rangle. \end{aligned}$$

Analogously for the second path p_2 we can write:

$$\begin{aligned} X^1_{p_2} &= \{\langle renameNode(0,c),\ removeLeaf(0.0)\rangle\},\ a^1_{p_2} = 1,\\ X^2_{p_2} &= \{\langle\rangle\},\ a^2_{p_2} = 2,\ \text{and}\\ X^3_{p_2} &= \{\langle renameNode(2.1,c)\rangle, \langle removeLeaf(2.1)\rangle\}\ \text{and}\ a^3_{p_2} = 3. \end{aligned}$$

Hence, path p_2 translates to $fix(p_2) = \{S_3, S_4\}$, where:

$$\begin{aligned} S_3 &= \langle renameNode(0,c),\ removeLeaf(0.0),\ renameNode(2.1,c)\rangle\ \text{and}\\ S_4 &= \langle renameNode(0,c),\ removeLeaf(0.0),\ removeLeaf(2.1)\rangle. \end{aligned}$$

Note that both $X^4_{p_1}$ *and* $X^3_{p_2}$ *were derived from the same edge* $e^{\texttt{repair}:d,D_A}_{2,2\to3,2}$*, and, so, the same translation of the corresponding nested intent repair, but their position shift was not the same. Also note that* S_3 *equals to an edit sequence we already studied in Example 3.4.*

Finally, having resolved translations of both the shortest correction paths p_1 *and* p_2*, we can wrap them into the translation of the entire sequence repair* $\mathcal{N}_\mathcal{I}$ *and formally write that* $fix(\mathcal{N}_\mathcal{I}) = fix(p_1) \cup fix(p_2) = \{S_1, S_2, S_3, S_4\}$*.*

Now, only the translation of intent repairs themselves remains to be discussed.

Definition 3.22 (Intent Repair Translation)**.** *Having an intent repair* $\mathcal{R}_\mathcal{I} = (e, \mathcal{N}_\mathcal{I}, cost)$ *for a correction intent* $\mathcal{I}$ *of type* t*, we define an* intent repair translation *of* $\mathcal{R}_\mathcal{I}$ *as* $fix(\mathcal{R}_\mathcal{I})$ *this way:*

- *If* $t =$ `correct`*, then* $fix(\mathcal{R}_\mathcal{I}) = \{modCut(r) \mid r \in fix(\mathcal{N}_\mathcal{I})\}$*.*
- *If* $t =$ `insert`*, then* $fix(\mathcal{R}_\mathcal{I}) = \{\langle fix(e)\rangle.modPrepend(r, 0) \mid r \in fix(\mathcal{N}_\mathcal{I})\}$*.*
- *If* $t =$ `delete`*, then* $fix(\mathcal{R}_\mathcal{I}) = \{modPrepend(r, 0).\langle fix(e)\rangle \mid r \in fix(\mathcal{N}_\mathcal{I})\}$*.*
- *If* $t =$ `repair`*, then* $fix(\mathcal{R}_\mathcal{I}) = \{modPrepend(r, 0) \mid r \in fix(\mathcal{N}_\mathcal{I})\}$*.*
- *If* $t =$ `rename`*, then* $fix(\mathcal{R}_\mathcal{I}) = \{\langle fix(e)\rangle.modPrepend(r, 0) \mid r \in fix(\mathcal{N}_\mathcal{I})\}$*.*

Beside the proper adjustment of position parameters, the only other thing we had to consider was the proper order in which the translations of optional repairing instructions and sequence repairs should be mutually combined. This order depends on a type of a correction intent for which the translation is being constructed.

In case of the `insert` correction intent type, we always need to first use the repairing instruction (to add a new leaf node as a root node for the new subtree), and only then we can use the sequence repair itself (to generate this subtree, if any is to be generated). Similarly, we need to remove an entire existing subtree (if any) before its root node can be removed in case of the `delete` intents. Note that the wrong order for both these types of intents would lead to sequences that would not be correctly defined, and so the appropriate order is essential.

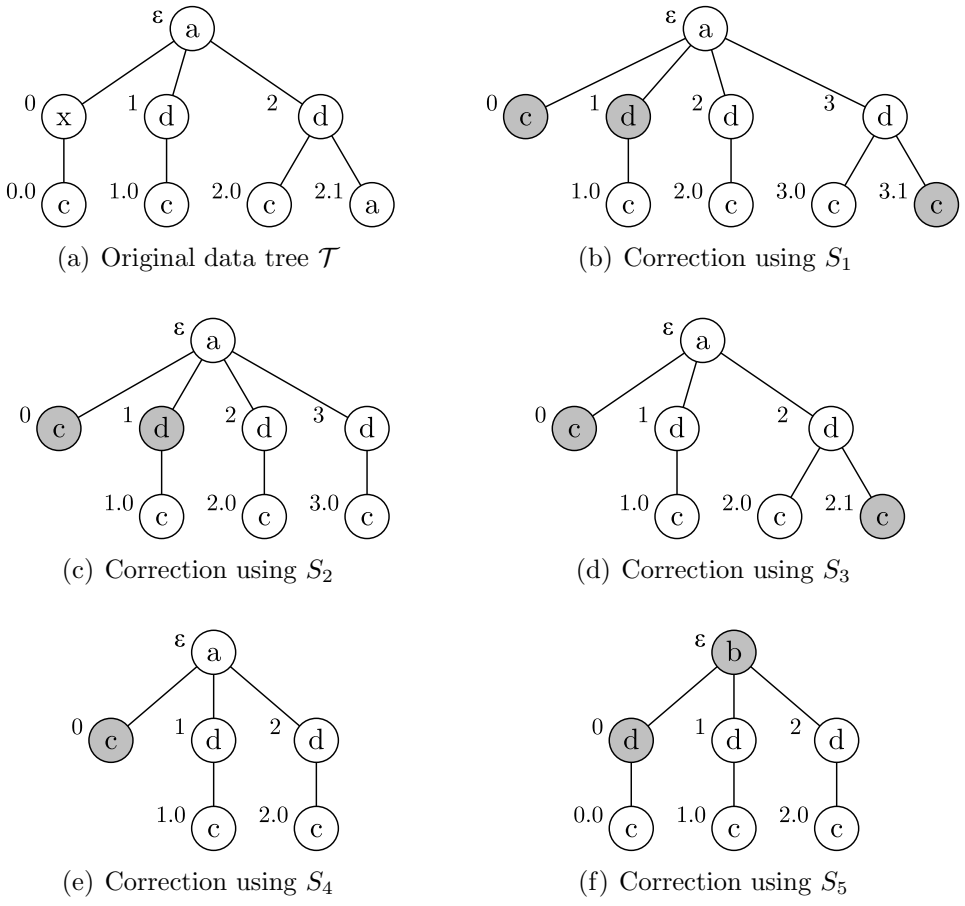
Correction intents of types `correct` and `repair` are not assigned any repairing instruction, so, we just alter the involved position parameters accordingly. Finally, in case of the `rename` intent type, we are free to choose the order – we decided to use the repairing instruction first, for we think this option is probably a more natural one.

Example 3.19. *Let us now construct the translation of the entire intent repair* $\mathcal{R}_\mathcal{I} = (\bot, \mathcal{N}_\mathcal{I}, 3)$ *from Example 3.16 for the correction intent* $\mathcal{I}$ *of type* `repair` *from Example 3.9.*

According to the previous definition, we can simply write that $fix(\mathcal{R}_\mathcal{I}) = modPrepend(fix(\mathcal{N}_\mathcal{I}), 0) = \{S'_1, S'_2, S'_3, S'_4\}$ *such that:*

$S'_1 = \langle addLeaf(0.0, c), renameNode(0.1, d), renameNode(0.3.1, c)\rangle,$
$S'_2 = \langle addLeaf(0.0, c), renameNode(0.1, d), removeLeaf(0.3.1)\rangle,$
$S'_3 = \langle renameNode(0.0, c), removeLeaf(0.0.0), renameNode(0.2.1, c)\rangle,$ *and*
$S'_4 = \langle renameNode(0.0, c), removeLeaf(0.0.0), removeLeaf(0.2.1)\rangle.$

It is also interesting to focus on intent repairs and their translation from the practical point of view. One thing is to obtain all the sequences of edit operations and then all the corresponding valid data trees – another thing is to choose the right one such offered corrected data tree according to our decision – one data tree we think really should represent the most appropriate correction from our point of view. It can be shown that we actually can let the user to choose such one correction directly, without the need of having the starting intent repair structure translated completely at all.



(a) Original data tree $\mathcal{T}$ (b) Correction using S_1

(c) Correction using S_2 (d) Correction using S_3

(e) Correction using S_4 (f) Correction using S_5

Figure 3.9: Data tree $\mathcal{T}$ and all its corrections with respect to grammar $\mathcal{G}$

Example 3.20. *Finally, we can conclude the entire correction process of the invalid data tree $\mathcal{T}$ from Example 2.2 with respect to the single type tree grammar $\mathcal{G}$ from Example 2.8.*

There exist five possible minimal corrections with the cost equal to 3*. All of them are depicted in Figure 3.9. They can be acquired from the original data $\mathcal{T}$ tree by applying the following final sequences of edit operations.*

$$S_1 = \langle addLeaf(0, c),\ renameNode(1, d),\ renameNode(3.1, c)\rangle,$$
$$S_2 = \langle addLeaf(0, c),\ renameNode(1, d),\ removeLeaf(3.1)\rangle,$$
$$S_3 = \langle renameNode(0, c),\ removeLeaf(0.0),\ renameNode(2.1, c)\rangle,$$
$$S_4 = \langle renameNode(0, c),\ removeLeaf(0.0),\ removeLeaf(2.1)\rangle,\ \textit{and}$$
$$S_5 = \langle renameNode(\epsilon, b),\ renameNode(0, d),\ removeLeaf(2.1)\rangle.$$

Formally, $dist(\mathcal{T}, L(\mathcal{G})) = 3$ *and the translation of the starting correction intent* $\mathcal{I}_\bullet$ *is equal to* $fix(\mathcal{R}_{\mathcal{I}_\bullet}) = \{S_1, S_2, S_3, S_4, S_5\}$.

Having described the intent repair structures and their translation into required sequences of edit operations, we have also finished the description of our entire correction model.

4. Algorithms

To recall the main idea of the correction algorithm, having a potentially (but not necessarily) invalid data tree $\mathcal{T}$ to be corrected with respect to a regular tree grammar $\mathcal{G}$, we process such data tree in a top-down manner by recursively nesting and evaluating correction intents.

In particular, we begin with the starting correction intent $\mathcal{I}_\bullet$ that represents the whole correction of the root node ϵ of $\mathcal{T}$, including its subtree. To evaluate a particular correction intent $\mathcal{I}$ with an action $A = (p, e, v_I, v_E)$ and an assignment $L = (u, \mathcal{C}, H, Y)$, we first split such $\mathcal{I}$ into easier subproblems, i.e. we create all the nested correction intents $\mathcal{I}' \in NestedIntents(\mathcal{I})$.

Then we compose and mutually bind all of them within a correction multigraph $\mathcal{M}_\mathcal{I}$, request their evaluation, and so we obtain their intent repairs $\mathcal{R}'_\mathcal{I}$ together with their overall correction costs as well. Therefore all the multigraph edges are now assigned with costs, which in turn enables us to search for the shortest correction paths $P^{min}_{v_S,V_T}$ that directly represent all the possible minimal corrections of a sequence of nodes u from the assignment L of $\mathcal{I}$.

Once we have found them, we simply encapsulate them into a sequence repair $\mathcal{N}_\mathcal{I}$, and then we consider the optional repairing instruction e from $\mathcal{I}$ as well to finally acquire the intent repair $\mathcal{R}_\mathcal{I}$ for $\mathcal{I}$ itself.

Having backtracked to the very beginning, i.e. having found the intent repair $\mathcal{R}_{\mathcal{I}_\bullet}$ for the starting correction intent $\mathcal{I}_\bullet$, we have successfully corrected the entire data tree $\mathcal{T}$. In case we are really interested in all the corrected data trees, i.e. valid data trees $\operatorname{argmin}_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}, \mathcal{T}_2)$ that have the minimal possible distance from $\mathcal{T}$, we only need to obtain all the sequences of edit operations resulting from the translation $fix(\mathcal{R}_{\mathcal{I}_\bullet})$ and apply them separately on $\mathcal{T}$.

Thus we have figured out the entire correction problem $correct(\mathcal{T}, \mathcal{G})$ of a data tree $\mathcal{T}$ with respect to a regular tree grammar $\mathcal{G}$ as we have defined it.

This chapter is dedicated to the postponed question how the search for the shortest correction paths should actually be conducted. Though the first glimpse might suggest otherwise, it is actually not straightforward to perform this search with desired efficiency.

This means that we are going to describe how the correction multigraphs should be constructed, whether all their vertices and edges really have to be explored and considered, how to search for the shortest paths themselves, as well as how to request processing of the nested correction intents, and whether to request such evaluation at all.

Beginning with *correction tasks* and other basic notions we will need throughout this chapter to describe our correction algorithms formally, we especially pay attention to *intent signatures* at first, since they allow us to significantly prune the space of all the created correction intents $CreatedIntents(\mathcal{T}, \mathcal{G})$, and so to reduce the amount of work that is required to be done.

Next, we introduce three *correction strategies* representing different ways and extents of correction multigraph construction and exploration – hopefully finding optimizations that can help us to improve the overall efficiency once again. Last but not least, we describe several *execution approaches* – different ways of implementation of correction strategies from the technical point of view.

4.1 Algorithm Essentials

Before we are able to present and discuss details of all the correction strategies and execution approaches, we first introduce correction tasks as encapsulations of correction intents, and then explain how the correction of a provided data tree is initiated as a whole from the algorithmic perspective.

4.1.1 Correction Tasks

Although correction intents, as we have introduced them, fully define the correction problem and subproblems we are dealing with, and so could be used as the only argument for the recursive correction routine (except the data tree and grammar, of course), we decided to wrap them into a structure we call a *correction task*, and to use this structure instead.

This means that whenever there is a correction intent $\mathcal{I}$ to be evaluated, we first create its correction task $\mathcal{K}_\mathcal{I}$, and then we use this task and its internal components during the whole further recursive processing instead of $\mathcal{I}$ itself.

Definition 4.1 (Correction Task)**.** *Given a correction intent $\mathcal{I}$, a* correction task *$\mathcal{K}_\mathcal{I} = (I, R, phase, vars, quota, deps)$ for $\mathcal{I}$ is a structure, where:*

- *$\mathcal{R}$ is a reference to the intent repair for $\mathcal{I}$,*
- *$phase \in \{$*`initialization`*,* `exploration`*,* `termination`*,* `evaluated`*$\}$ is a current* evaluation phase*,*
- *$vars$ is a container of* working variables*:*
  - *$\mathcal{M}$ is a reference to the correction multigraph for $\mathcal{I}$,*
  - *$V_{reached}$ is a set of* reached vertices*,*
  - *$c_{reached}$ is a currently* reached cost*,*
  - *c_{fixed} is a* fixed cost*,*
  - *$pDistance$ and $pPredecessors$ are tracing functions,*
  - *$E_{delayed}$ is a set of* delayed edges*,*
- *quota is an* assigned quota *limiting the extent of evaluation, and*
- *deps is a container of* request dependencies*:*
  - *$K_{requesting}$ is a set of references to* requesting tasks*,*
  - *$K_{requested}$ is a set of references to* requested tasks*.*

Technically, correction tasks comprise of several components that enable us to preserve all the required internal evaluation variables altogether within only one structure. In case of the default correction strategy and a straightforward implementation, they would not be needed at all. However, in case of more extended correction strategies and execution approaches we are going to introduce, they become necessary. Therefore we have decided to introduce them right now at the very beginning, so we can explain all the following algorithms in a unified

way, building on the same basis, and so highlighting their mutual differences in a more convenient and comprehensive way.

The purpose of the provided definition of correction tasks was just to describe their general structure and internal components. When a new correction task for a particular correction intent is created, nearly all its components are undefined at the beginning. Once its evaluation is requested and commenced, all its components are arbitrarily utilized, accessed and modified to fulfill the intent evaluation goal. Once this goal is attained and the intent repair produced, a reference to this repair is finally stored within the task itself, and all the remaining components can be cleared, since they will no longer be needed.

Now we at least shortly describe the intended meaning of some of the task components. First, a container of working variables $vars$ is used during the exploration and construction of the corresponding correction multigraph $\mathcal{M}$ in order to find all the shortest correction paths. An assigned quota is used to restrain the extent of requested and allowed evaluation in case of the most optimized correction strategy, the refinement one. Finally, a container of request dependencies only plays its role in case of the forwarding execution approaches, where the management of used system threads and their mutual relationships have to be recorded manually.

All in all, whether all the individual components will really be used, depends on a particular correction strategy and an execution approach – and so we will reveal their details later on. To conclude, all the components except the intent reference $\mathcal{I}$ are optional and may not be defined, i.e. may be equal to $\perp$.

Before we move forward to the main correction algorithm, we introduce yet another one notion. If we had defined $CreatedIntents(\mathcal{T}, \mathcal{G})$ as a set of all the distinct correction intents that need to be created and evaluated in order to process the correction of a data tree $\mathcal{T}$ with respect to a grammar $\mathcal{G}$, we can analogously derive $CreatedTasks(\mathcal{T}, \mathcal{G}) = \{\mathcal{K}_{\mathcal{I}} \mid \mathcal{K}_{\mathcal{I}} \text{ is a task for } \mathcal{I} \in CreatedIntents(\mathcal{T}, \mathcal{G})\}$ as a set of all the *created tasks*.

4.1.2 Correction Routine

Let us now shortly look at the main correction procedure in Algorithm 4.1, where we describe how the whole correction of a data tree $\mathcal{T}$ with respect to a grammar $\mathcal{G}$ is initiated, performed and terminated from a high-level perspective.

Algorithm 4.1: Correction algorithm: $correct(\mathcal{T}, \mathcal{G})$

Input: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$

1 $\mathcal{I}_\bullet \leftarrow$ create the starting correction intent for $\mathcal{T}$ and $\mathcal{G}$;
2 $\mathcal{K}_{\mathcal{I}_\bullet} \leftarrow$ create a task for $\mathcal{I}_\bullet$ via $createCorrectionTask(\mathcal{I}_\bullet)$;
3 initiate the recursive correction by calling $correct(\mathcal{K}_{\mathcal{I}_\bullet})$;
4 fetch a reference to the intent repair $\mathcal{R}_{\mathcal{I}_\bullet}$ for $\mathcal{I}_\bullet$ from $\mathcal{K}_{\mathcal{I}_\bullet}$;
5 **return** *set of corrected data trees* $\{\mathcal{T}' \mid \mathcal{T} \xrightarrow{S} \mathcal{T}',\ S \in fix(\mathcal{R}_{\mathcal{I}_\bullet})\}$;

The provided code directly corresponds to all the model definitions we have introduced so far, and so we believe that no further explanation is needed at this moment. We only assume that both the data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$ are already provided, i.e. have already been parsed from their source files prior this correction procedure has been invoked.

Even though the theoretical objective is to really return the set of all the corrected data trees $\mathit{correct}(\mathcal{T}, \mathcal{G})$ as we have described it in Definition 3.6, we have also explained that the final translation step can be omitted in case the user is only interested in just one particular correction (which actually would be the most common case in practice).

The remaining parts of this chapter will discuss, how to actually implement the recursive correction routine $\mathit{correct}(\mathcal{K}_\mathcal{I})$ with respect to different correction strategies we have yet only shortly listed.

Given a particular correction intent $\mathcal{I}$ wrapped by a correction task $\mathcal{K}_\mathcal{I}$, its purpose is to construct the corresponding correction multigraph $\mathcal{M}_\mathcal{I}$, find the shortest correction paths, and encapsulate them in the resulting intent repair structure $\mathcal{R}_\mathcal{I}$.

Though we want to make the code of all the following algorithms as accurate and formal as possible, we will omit some minor technical details and only focus on the main logic to make the code more readable in general. However, all the following aspects will always be described completely and correctly: how correction multigraphs are constructed, which of their vertices and edges are explored, how and when the evaluation of nested correction intents is requested (if ever), and how the searching for the shortest correction paths proceeds and is terminated once we have found them.

We will also follow several simple conventions in the code of our algorithms. First, we will use $\leftarrow$ as an assignment operator instead of the standard $=$. We will also use a symbol $^\circ$ in names of variables (like, for example, $\mathcal{M}^\circ_\mathcal{I}$) to denote that a given structure (in this case the correction multigraph for an intent $\mathcal{I}$) is currently being constructed, and so may not yet be completely defined and may not fully correspond to formal definitions we provided. Once they become, we explicitly mark them as finalized and start using the standard names (like $\mathcal{M}_\mathcal{I}$) from that moment on.

4.2 Intent Signatures

Since we have intentionally split the internal structure of a correction intent $\mathcal{I} = (\mathit{id}, \mathit{type}, A, L)$ into its action A and assignment L components, it should not be surprising that the action component A has absolutely no effect on the intent repair $\mathcal{R}_\mathcal{I}$ we construct.

However, it might be interesting that we can describe the general dependency of intent repairs on correction intents even more thoroughly and in finer terms. And even more challenging could be the finding that without such discussion the correction algorithm would probably not be efficient enough to be successfully used in practice.

So, let us have a look at a short example. Assume that we are processing a particular correction intent $\mathcal{I}$ and we are about to consider a new nested correction

intent $\mathcal{I}'$ of type `delete` – an intent which should cause that a particular existing data tree node p will be removed, including its subtree (if there is any). It is apparent that the resulting intent repair $\mathcal{R}'_{\mathcal{I}}$ not only does not depend on the action A of intent $\mathcal{I}$ – it actually only depends on the given data tree node p and nothing else. Therefore, whichever correction intent $\mathcal{I}$ decides to consider the removal of this particular node p, though distinct nested correction intents will always have to be created, the resulting intent repair structure $\mathcal{R}'_{\mathcal{I}}$ will always be the same – exactly the same.

As a consequence, it might become useful to find a way how to detect these (and similar) situations in general. Because if we are able to accomplish this call, we simply do not need to compute the same intent repairs repeatedly, over and over again. The answer lies in the following definition of *intent signatures*.

Definition 4.2 (Intent Signatures)**.** *Let $\mathcal{I} = (id,\ type,\ A,\ L)$ be a correction intent with an action $A = (p,\ e,\ v_I,\ v_E)$ and an assignment $L = (u,\ \mathcal{C},\ H,\ Y)$ such that $\mathcal{C} = \mathcal{C}_{t,n}$ in case type $\notin \{$`correct`, `delete`$\}$ in particular.*

Then we define an intent signature *for $\mathcal{I}$ as a tuple $sig(\mathcal{I})$, where:*

- *If type =* `correct`*, then $sig(\mathcal{I}) =$ (`correct`).*
- *If type =* `insert`*, then $sig(\mathcal{I}) =$ (`insert`, H).*
- *If type =* `delete`*, then $sig(\mathcal{I}) =$ (`delete`, p).*
- *If type =* `repair`*, then $sig(\mathcal{I}) =$ (`repair`, p, t, n).*
- *If type =* `rename`*, then $sig(\mathcal{I}) =$ (`rename`, p, t, n).*

We say that two correction intents $\mathcal{I}_1 = (id_1,\ type_1,\ A_1,\ L_1)$ and $\mathcal{I}_2 = (id_2,\ type_2,\ A_2,\ L_2)$ have identical signatures*, i.e. $sig(\mathcal{I}_1) = sig(\mathcal{I}_2)$, if and only if $type_1 = type_2$ and all the other corresponding pairs of components of $sig(\mathcal{I}_1)$ and $sig(\mathcal{I}_2)$ are identical as well.*

The meaning of intent signatures is straightforward – whenever we have two correction intents $\mathcal{I}_1$ and $\mathcal{I}_2$ with identical signatures, then the resulting intent repair structures $\mathcal{R}_{\mathcal{I}_1}$ and $\mathcal{R}_{\mathcal{I}_2}$ will also be exactly the same. Hence, we only need to compute them once.

Example 4.1. *Let us now return to our sample correction intent $\mathcal{I}$ with its correction multigraph $\mathcal{M}$ that we already presented in Example 3.10 and illustrated in Figure 3.5.*

In this example we split all the intents $\mathcal{I}'$ from $NestedIntents(\mathcal{I})$ into disjoint sets, exactly according to their intent signatures $sig(\mathcal{I}')$. To make the following listing more readable, we used multigraph edges as references to these nested correction intents instead.

$sig(\mathcal{I}') =$ (`insert`, $\langle \mathcal{C}_{c,C} \rangle$) in case of
edges $e_{s,0\to s,1}^{\texttt{insert}:c,C}$ for all strata $s \in \{0,1,2,3\}$.
$sig(\mathcal{I}') =$ (`insert`, $\langle \mathcal{C}_{d,D_A} \rangle$) in case of
edges $e_{s,1\to s,2}^{\texttt{insert}:d,D_A}$ and loops $e_{s,2\to s,2}^{\texttt{insert}:d,D_A}$ for all strata $s \in \{0,1,2,3\}$.

$sig(\mathcal{I}') = (\texttt{delete}, 0)$ *in case of edges* $e^{\texttt{delete}}_{0,q\to 1,q}$ *for all states* $q \in \{0,1,2\}$.
$sig(\mathcal{I}') = (\texttt{delete}, 1)$ *in case of edges* $e^{\texttt{delete}}_{1,q\to 2,q}$ *for all states* $q \in \{0,1,2\}$.
$sig(\mathcal{I}') = (\texttt{delete}, 2)$ *in case of edges* $e^{\texttt{delete}}_{2,q\to 3,q}$ *for all states* $q \in \{0,1,2\}$.

$sig(\mathcal{I}') = (\texttt{rename}, 0, c, C)$ *in case of edge* $e^{\texttt{rename}:c,C}_{0,0\to 1,1}$.
$sig(\mathcal{I}') = (\texttt{rename}, 1, c, C)$ *in case of edge* $e^{\texttt{rename}:c,C}_{1,0\to 2,1}$.
$sig(\mathcal{I}') = (\texttt{rename}, 2, c, C)$ *in case of edge* $e^{\texttt{rename}:c,C}_{2,0\to 3,1}$.

$sig(\mathcal{I}') = (\texttt{rename}, 0, d, D_A)$ *in case of edges* $e^{\texttt{rename}:d,D_A}_{0,1\to 1,2}$ *and* $e^{\texttt{rename}:d,D_A}_{0,2\to 1,2}$.
$sig(\mathcal{I}') = (\texttt{repair}, 1, d, D_A)$ *in case of edges* $e^{\texttt{repair}:d,D_A}_{1,1\to 2,2}$ *and* $e^{\texttt{repair}:d,D_A}_{1,2\to 2,2}$.
$sig(\mathcal{I}') = (\texttt{repair}, 2, d, D_A)$ *in case of edges* $e^{\texttt{repair}:d,D_A}_{2,1\to 3,2}$ *and* $e^{\texttt{repair}:d,D_A}_{2,2\to 3,2}$.

Apparently, the most notable impact of intent signatures in this example is on nested intents of types `insert` *and* `delete` *(which is not a coincidence).*

Each time we have fully evaluated a particular correction intent $\mathcal{I}$, we simply put the resulting intent repair structure $\mathcal{R}_\mathcal{I}$ into a store we call a *task cache* using its signature $sig(\mathcal{I})$ as an indexing key. Then, whenever we are about to request evaluation of a nested correction intent $\mathcal{I}'$ – we first try to fetch the equivalent intent repair from the tasks cache using $sig(\mathcal{I}')$ as a key – and only in case we are not successful we actually proceed to the evaluation request and really commence its execution.

Notice, however, that although we may not need to request the evaluation of the given nested correction intent $\mathcal{I}'$, we still have to create it as a structure $\mathcal{I}'$ itself, and integrate it within a correction multigraph $\mathcal{M}_\mathcal{I}$ for $\mathcal{I}$. Therefore, one thing is to create a nested correction intent and a corresponding multigraph edge to which it is associated, another thing is whether we have already computed the intent repair of a given signature, or not yet.

The task cache is formally modeled as an ordinary set of pairs $(key, \mathcal{K})$, allowing us to store references to correction tasks (and so the associated intent repairs as a consequence) under values of provided indexing *keys*. In our case, these indexing key are simply the values of intent signatures, exactly as we have introduced them. However, we will need to use the task cache even for other purposes, and so we based it on tasks (and not directly on intent repairs), as well as we allowed a more robust indexing schema (and talk about indexing keys in general, not restraining them to intent signatures in particular).

It is also worth of noting that if the recursive nesting of correction intents as such could so far be modeled within boundaries of a tree structure, now we need to switch to a directed acyclic graph structure. This observation does not pose any notable complications except that execution approaches that are based on multiple threads must be aware of this fact and so must ensure necessary synchronization whenever needed.

From another point of view, both the sets $CreatedIntents(\mathcal{T}, \mathcal{G})$ of all the created correction intents and $CreatedTasks(\mathcal{T}, \mathcal{G})$ of all the created tasks now significantly reduce their sizes – simply because all the created tasks now have intents of distinct signatures. As a consequence, both these sets no longer must be of identical sizes. Instead, the number of the created tasks will always be at most as high as the number of created intents. In practice, the number of tasks will actually tend to be sharply lower.

In general, there are several situations when intent signatures can play their role – more different situations than we might conclude just from the first glimpse. First, we can find identical signatures between different nested correction intents within one particular correction multigraph (as we have shown in the previous example). Next, we can also find them between different correction intents from different multigraphs (i.e. when evaluating different correction intents). And, last but not least, there can be identical signatures even between the same or at least nearly the same correction intents from different multigraphs – it means intents that are equivalent in their assignment components, or even both action and assignment components – though they will always differ in their identifiers from apparent reasons.

Anyway, since the evaluation of correction intents is now no longer bound to particular intents, as it is bound to intent signatures, we should also change our naming convention accordingly for correction multigraphs $\mathcal{M}_{\mathcal{I}}$, sequence repairs $\mathcal{N}_{\mathcal{I}}$ and intent repairs $\mathcal{R}_{\mathcal{I}}$ to $\mathcal{M}_{sig(\mathcal{I})}$, $\mathcal{N}_{sig(\mathcal{I})}$ and $\mathcal{R}_{sig(\mathcal{I})}$ respectively. However, to make the following text easier to read, we will simply use just $\mathcal{M}$, $\mathcal{N}$ and $\mathcal{R}$ respectively when it is not clear from the context whether intent signatures are applied, or they are not.

Putting all the previous observations together, despite both the idea and implementation of intent signatures is easy, the impact on the correction efficiency is crucial. And although we believe that our correction algorithms could not be usable in practice without having this handling of signatures *enabled*, all the algorithms we are going to discuss will work even when these signatures would be *disabled* – exactly in a form they will be presented.

4.3 Correction Strategies

The purpose of this section is to introduce three different correction strategies we proposed. It means particular techniques how correction multigraphs can be constructed, their vertices and edges explored, and the shortest correction paths they contain revealed in the end.

We start with a straightforward means we call a *default strategy*, and then we discuss its two improvements – optimizations that allow us to reduce the overall number of multigraph edges that need to be explored in order to find all the shortest correction paths we are searching for, and so to reduce the number of nested correction intents that need to be evaluated, and so improving the overall algorithm efficiency. And whereas an *exploring strategy* as the first optimization is a step in the right direction, a *refinement strategy* enables us to achieve pruning even vertically to the depth of the recursive nesting, and not just horizontally within correction multigraphs themselves.

4.3.1 Default Strategy

The default strategy exactly follows the correction model as we have introduced it – whenever a particular correction intent $\mathcal{I}$ is about to be evaluated, we always and without exceptions construct its entire correction multigraph $\mathcal{M}$, with all its vertices and edges being explored. Having acquired intent repairs for all the

nested correction intents, we straightforwardly find the shortest correction paths and produce the intent repair structure $\mathcal{R}$ for $\mathcal{I}$ in the very end.

The formal and overall description for this first correction strategy is presented in Algorithm 4.2.

Algorithm 4.2: Default strategy: $correct_{\texttt{DEF}}(\mathcal{K})$

Input: Correction task $\mathcal{K} = (\mathcal{I}, \mathcal{R} = \bot, phase = \bot, vars = \bot, quota = \bot, deps = \bot)$ for a correction intent $\mathcal{I} = (id, type, A, L)$ with an action $A = (p, e, v_I, v_E)$ and an assignment $L = (u, \mathcal{C}, H, Y)$

Global: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$

`// Initialization phase`

1 $vars \leftarrow$ initialize working variables ($\mathcal{M}^\circ = \bot$, $V_{reached} = \emptyset$, $c_{reached} = \bot$, $c_{fixed} = \bot$, $pDistance = \emptyset$, $pPredecessors = \emptyset$, $E_{delayed} = \bot$) for task $\mathcal{K}$;

2 $\mathcal{M}^\circ \leftarrow$ create a new empty correction multigraph ($V^\circ = \emptyset$, $E^\circ = \emptyset$, v_S, V_T) with standard v_S and V_T according to intent assignment L;

`// Construction phase`

3 **foreach** $\mathcal{I}' \in NestedIntents(\mathcal{I})$ **do**

4 Let $\mathcal{I}' = (id', type', A', L')$ and $A' = (p', e', v'_I, v'_E)$;

5 $\mathcal{R}' \leftarrow requestIntentRepair_{\texttt{DEF}}(\mathcal{I}')$;

6 **if** $(v'_I \notin V^\circ)$ **then** $V^\circ \leftarrow V^\circ \cup \{v'_I\}$;

7 **if** $(v'_E \notin V^\circ)$ **then** $V^\circ \leftarrow V^\circ \cup \{v'_E\}$;

8 $E^\circ \leftarrow E^\circ \cup \{(v'_I, v'_E, \mathcal{I}', \mathcal{R}')\}$;

9 mark multigraph $\mathcal{M}^\circ$ as final and use $\mathcal{M} = (V, E, v_S, V_T)$ instead;

`// Exploration phase`

10 $pDistance(v_S) \leftarrow 0$; $pPredecessors(v_S) \leftarrow \emptyset$;

11 $V_{reached} \leftarrow \{v_S\}$;

12 $performExplorationLoop_{\texttt{DEF}}(\mathcal{K})$;

`// Termination phase`

13 $\mathcal{N} \leftarrow$ create a sequence repair for $\mathcal{I}$ based on multigraph $\mathcal{M}$, tracing functions ($pDistance$, $pPredecessors$) and $cost(P^{min}_{v_S,V_T}) = c_{fixed}$;

14 $\mathcal{R} \leftarrow$ create an intent repair using $\mathcal{N}$ and a repairing instruction e;

15 finally put $phase \leftarrow$ `evaluated` and clear working variables $vars \leftarrow \bot$;

Having prepared the container of working variables (line 1) with all its components except for the set of delayed edges (since it will not be required by this strategy), and also initialized the correction multigraph $\mathcal{M}^\circ$ itself (line 2), we simply iterate over all the nested correction intents $\mathcal{I}'$ in a multigraph construction loop (lines 3 – 8), where we insert into the multigraph all the vertices and edges for such $\mathcal{I}'$ we come across.

Once the multigraph $\mathcal{M}^\circ$ is constructed completely (line 9) and marked as $\mathcal{M}$, we initialize both tracing functions (line 10) and the set of reached vertices

(line 11), so we are able to perform the entire search for the shortest correction paths within $\mathcal{M}$ (line 12). Before we describe how this shortest paths exploration loop actually works, we first finish the overview of the whole strategy.

So, having found all the paths, the only thing to be accomplished is to encapsulate them into a corresponding sequence repair $\mathcal{N}$ at first (line 13), and to the final intent repair $\mathcal{R}$ in the very end (line 14), while disposing of all the no longer needed working variables and marking the intent repair as fully evaluated for technical reasons (line 15).

Thus we have attained the goal of the entire evaluation of the correction intent $\mathcal{I}$ with respect to the default correction strategy.

Now we proceed to Algorithm 4.3, where we describe how the evaluation of nested correction intents is really requested – all that in accordance to intent signatures and caching of evaluated intent repairs.

In case the handling of signatures is enabled (lines 1 – 7), we first always try to fetch the equivalent intent repair $\mathcal{R}'$ according to the signature of $\mathcal{I}'$ (line 2), and only when such repair has not yet been evaluated, we prepare a new empty task $\mathcal{K}'$ to wrap $\mathcal{I}'$ (line 4), put it into the tasks cache (line 5), and then proceed to its recursive evaluation (line 7).

Just note that we technically do not directly store references to intent repairs at the moment they really become fully evaluated. Instead we preserve references to all the created correction tasks within the tasks cache – from the very moment they were created and requested to be evaluated. This means that the required intent repairs themselves are then available indirectly right through correction tasks – whenever we need to access them later on, once they are evaluated.

In case intent signatures are not considered (lines 8 – 10), a new task is created, as well as its recursive evaluation is invoked in all circumstances.

Algorithm 4.3: Default strategy: $requestIntentRepair_{\texttt{DEF}}(\mathcal{I}')$

Input: Nested correction intent $\mathcal{I}' = (id', type', A', L')$
Output: Intent repair $\mathcal{R}'$ for $\mathcal{I}'$
Use: $TasksCache$ if signatures are enabled

```
1  if (signatures are enabled) then
2      if (∃ t' such that (sig(I'), t') ∈ TasksCache) then K' ← t';
3      else
4          K' ← createEmptyTask_DEF(I');
5          TasksCache ← TasksCache ∪ {(sig(I'), K')};
6      Let K' = (I', R', phase', vars', quota', deps');
7      if (phase' ≠ evaluated) then correct_DEF(K');
8  else
9      K' ← createEmptyTask_DEF(I');
10     correct_DEF(K');
11 return R';
```

The creation of a new correction task is described in Algorithm 4.4, where simply a new and completely empty task structure is prepared for $\mathcal{I}'$ and returned.

Algorithm 4.4: Default strategy: $createEmptyTask_{\texttt{DEF}}(\mathcal{I}')$

Input: Nested correction intent $\mathcal{I}'$
Output: Newly created correction task $\mathcal{K}'$ for $\mathcal{I}'$

1 **return** $\mathcal{K}' \leftarrow (\mathcal{I}', \mathcal{R}' = \bot, phase' = \bot, vars' = \bot, quota' = \bot, deps' = \bot)$;

The only thing that still remains to be described from the default correction strategy is actually one of the most important parts – that is how the search for the shortest correction paths really works. And though this search is based on the traditional Dijkstra's algorithm, the proper description turns out to be useful later on as a comparison basis when introducing both the improved correction strategies.

Algorithm 4.5: Default strategy: $performExplorationLoop_{\texttt{DEF}}(\mathcal{K})$

Input: Correcton task $\mathcal{K}$ for an intent $\mathcal{I}$

1 Let $vars = (\mathcal{M} = (V, E, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed} = \bot)$ be working variables for task $\mathcal{K}$;
2 **while** $(V_{reached} \neq \emptyset)$ **do**
3 $v \leftarrow$ select any vertex from $\text{argmin}_{v' \in V_{reached}} pDistance(v')$;
4 $V_{reached} \leftarrow V_{reached} \setminus \{v\}$;
5 $c_{reached} \leftarrow pDistance(v)$;
6 **if** $(v \in V_T) \wedge (c_{fixed} = \bot)$ **then** $c_{fixed} \leftarrow c_{reached}$;
7 **if** $(c_{fixed} \neq \bot) \wedge (c_{fixed} < c_{reached})$ **then break**;
8 **foreach** $g \leftarrow (v_1 = v, v_2, \mathcal{I}', \mathcal{R}') \in E_v^{out}$ **do**
9 **if** $(pDistance(v_2) = \bot)$ **then** $V_{reached} \leftarrow V_{reached} \cup \{v_2\}$;
10 Let $\mathcal{R}' = (e', \mathcal{N}', cost')$;
11 $c \leftarrow c_{reached} + cost'$;
12 **if** $(pDistance(v_2) = \bot) \vee (pDistance(v_2) > c)$ **then**
13 $pDistance(v_2) \leftarrow c$; $pPredecessors(v_2) \leftarrow \{g\}$;
14 **else if** $(pDistance(v_2) = c)$ **then**
15 $pPredecessors(v_2) \leftarrow pPredecessors(v_2) \cup \{g\}$;

So, let us now focus on Algorithm 4.5. Its only purpose is to iterate over the set of reached vertices, while discovering the shortest correction paths step by step. Before the procedure is invoked, only the source vertex v_S of the multigraph $\mathcal{M}$ is in this set of reached vertices. Then the involved vertices are gradually removed

one by one (once they become selected and processed), and new ones added (once they become discovered on the contrary).

At the beginning of each iteration of the exploration loop (lines 2 – 15) we select one vertex v from the set of currently reached vertices that has the lowest distance $pDistance(v)$ from the source vertex v_S (line 3). If there are more such minimal vertices, we are free to choose any of them. Anyway, since its current tracing distance is minimal among all the reached and not yet processed vertices, it cannot be improved in any way later on (line 5).

Therefore we can declare this vertex v as processed, remove it from the set of the reached vertices (line 4), and inspect all its outgoing edges E_v^{out} (lines 8 – 15). When we come across a new edge ending vertex v_2 (that was not yet processed, nor is currently reached), we add it into the set of reached vertices (line 9). Then we set, replace or update values of $pDistance$ and $pPredecessors$ tracing functions accordingly for this vertex v_2 (lines 11 – 15).

When the loop over the reached vertices selects some target vertex from V_T for the first time (line 6), the cost of the shortest correction paths $cost(P^{min}_{v_S,V_T})$ has just been resolved. However, we cannot terminate the exploration loop at this moment, since there can still exist other shortest correction paths – either to this first discovered target vertex, or to any other from the remaining ones. And so the exploration loop must continue until the reached cost rises even more (line 7). Only then it is finally safe to end.

At this moment we have described the default correction strategy completely. Despite it is able to achieve the set correction goal, its main problem is that it forces the correction multigraphs to be always discovered completely. And so it might be useful to come with another strategy – an improved one – a strategy that would be able to avoid exploration of those multigraph vertices and edges that are not necessarily required for finding the shortest correction paths.

4.3.2 Exploring Strategy

And this is exactly the idea hidden behind the *exploring strategy*, where we integrated both the multigraph construction and exploration phases together. This means that vertices and edges of correction multigraphs are discovered and constructed on demand – exactly as the exploring loop over the reached vertices requires in order to search for the shortest correction paths accurately.

The main correction routine is presented in Algorithm 4.6. There are actually no big differences to the default strategy – only the correction phase has been completely removed, and the exploration phase begins with an insertion of the source vertex v_S not only into the set of the reached vertices (line 5), but also into the set of multigraph vertices as such (line 3).

And since the purpose of this strategy is to horizontally prune the correction multigraph $\mathcal{M}^\circ$, we are no longer able to declare it as final at the end. However, this has no impact on the encapsulation of a sequence repair $\mathcal{N}$ (line 7), nor the intent repair $\mathcal{R}$ (line 8), for they are both provided with everything they require to be constructed correctly.

Now the question is how the exploration loop (line 6) needs to be changed.

Algorithm 4.6: Exploring strategy: $correct_{\texttt{EXP}}(\mathcal{K})$

Input: Correction task $\mathcal{K} = (\mathcal{I}, \mathcal{R} = \bot, phase = \bot, vars = \bot, quota = \bot, deps = \bot)$ for a correction intent $\mathcal{I} = (id, type, A, L)$ with an action $A = (p, e, v_I, v_E)$ and an assignment $L = (u, \mathcal{C}, H, Y)$
Global: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$

`// Initialization phase`
1 $vars \leftarrow$ initialize working variables ($\mathcal{M}^\circ = \bot$, $V_{reached} = \emptyset$, $c_{reached} = \bot$, $c_{fixed} = \bot$, $pDistance = \emptyset$, $pPredecessors = \emptyset$, $E_{delayed} = \bot$) for task $\mathcal{K}$;
2 $\mathcal{M}^\circ \leftarrow$ create a new empty correction multigraph ($V^\circ = \emptyset$, $E^\circ = \emptyset$, v_S, V_T) with standard v_S and V_T according to intent assignment L;

`// Exploration phase`
3 $V^\circ \leftarrow V^\circ \cup \{v_S\}$;
4 $pDistance(v_S) \leftarrow 0$; $pPredecessors(v_S) \leftarrow \emptyset$;
5 $V_{reached} \leftarrow \{v_S\}$;
6 $performExplorationLoop_{\texttt{EXP}}(\mathcal{K})$;

`// Termination phase`
7 $\mathcal{N} \leftarrow$ create a sequence repair for $\mathcal{I}$ based on multigraph $\mathcal{M}^\circ$, tracing functions ($pDistance$, $pPredecessors$) and $cost(P^{min}_{v_S,V_T}) = c_{fixed}$;
8 $\mathcal{R} \leftarrow$ create an intent repair using $\mathcal{N}$ and a repairing instruction e;
9 finally put $phase \leftarrow$ `evaluated` and clear working variables $vars \leftarrow \bot$;

The formal description of the exploration loop enabling the search for all the shortest correction paths in case of the exploring strategy is provided in Algorithm 4.7.

The outer iteration loop over the set of reached vertices (lines 2 – 18) remains exactly the same as in case of the default strategy. This means we still always select one of the reached vertices currently having the minimal tracing distance from the source vertex (line 3), as well as both the termination conditions are preserved too (lines 6 and 7).

However, instead of iterating over the outgoing edges E^{out}_v from the currently processed vertex v, we have to directly iterate over all the nested correction intents $NestedIntents(\mathcal{I}, v)$ defined at v (lines 8 – 18), since these edges are not yet constructed. To do it, we first recursively acquire the corresponding intent repair $\mathcal{R}'$ for each such nested $\mathcal{I}'$ (line 10), create a newly discovered ending vertex v'_E (line 11) and the edge e itself (line 12), and finally set, replace or update values of the tracing functions accordingly once again (lines 14 – 18).

Finally, both the procedures for acquiring the nested intent repairs (either by really evaluating them, or fetching them from the tasks cache using their signatures in case they are enabled) and creation of new correction tasks remain exactly the same as for the default strategy (as presented in Algorithm 4.3 and Algorithm 4.4 respectively), and so we decided not to describe them once again. The only difference is that we switch from `DEF` to `EXP` alternatives of all the involved functions accordingly.

Algorithm 4.7: Exploring strategy: $performExplorationLoop_{\texttt{EXP}}(\mathcal{K})$

Input: Correction task $\mathcal{K}$ for an intent $\mathcal{I}$
Global: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$
Use: $TasksCache$ if signatures are enabled

1 Let $vars = (\mathcal{M}^\circ = (V^\circ, E^\circ, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed} = \bot)$ be working variables for task $\mathcal{K}$;
2 **while** $(V_{reached} \neq \emptyset)$ **do**
3 $v \leftarrow$ select any vertex from $\text{argmin}_{v' \in V_{reached}} pDistance(v')$;
4 $V_{reached} \leftarrow V_{reached} \setminus \{v\}$;
5 $c_{reached} \leftarrow pDistance(v)$;
6 **if** $(v \in V_T) \wedge (c_{fixed} = \bot)$ **then** $c_{fixed} \leftarrow c_{reached}$;
7 **if** $(c_{fixed} \neq \bot) \wedge (c_{fixed} < c_{reached})$ **then break**;
8 **foreach** $\mathcal{I}' \in NestedIntents(\mathcal{I}, v)$ **do**
9 Let $\mathcal{I}' = (id', type', A', L')$ and $A' = (p', e', v'_I = v, v'_E)$;
10 $\mathcal{R}' \leftarrow requestIntentRepair_{\texttt{EXP}}(\mathcal{I}')$;
11 **if** $(v'_E \notin V^\circ)$ **then** $V^\circ \leftarrow V^\circ \cup \{v'_E\}$; $V_{reached} \leftarrow V_{reached} \cup \{v'_E\}$;
12 $g \leftarrow (v, v'_E, \mathcal{I}', \mathcal{R}')$; $E^\circ \leftarrow E^\circ \cup \{g\}$;
13 Let $\mathcal{R}' = (e', \mathcal{N}', cost')$;
14 $c \leftarrow c_{reached} + cost'$;
15 **if** $(pDistance(v'_E) = \bot) \vee (pDistance(v'_E) > c)$ **then**
16 $pDistance(v'_E) \leftarrow c$; $pPredecessors(v'_E) \leftarrow \{g\}$;
17 **else if** $(pDistance(v'_E) = c)$ **then**
18 $pPredecessors(v'_E) \leftarrow pPredecessors(v'_E) \cup \{g\}$;

The overall impact of the exploring strategy is that there emerges a reasonably high chance that some parts of correction multigraphs (some of their edges and vertices) might not be explored at all. Hence some nested correction intents might not be requested for their evaluation at all, and so the overall correction efficiency will most likely be improved.

4.3.3 Refinement Strategy

Both the already introduced correction strategies have one important common feature – whenever we request any nested correction intent to be evaluated, we always acquire its completely evaluated intent repair structure, and so its overall correction cost – a fully resolved cost which we then in turn use to search for the shortest correction paths.

But what about considering another optimization strategy – a strategy that would not force the nested correction intents to be evaluated completely, and, at the same time and as a consequence, that would be able to rely just on estimates of their correction costs?

Our response to this idea is the *refinement strategy*. In order to explain its main principles, we try to describe the situation from the perspective of a correction intent $\mathcal{I}$ at first, and then from the perspective of nested correction intents $\mathcal{I}'$ it requests.

So, when an intent $\mathcal{I}$ is currently being processed and we are about to request some nested intent $\mathcal{I}'$ to be evaluated, we no longer want this evaluation to be complete. Instead, we only would like to achieve a small and limited progress in its evaluation in order to obtain refined estimate of its overall correction cost. This will hopefully permit us to make decisions on further processing and exploration of the multigraph $\mathcal{M}$ for $\mathcal{I}$ more consciously and efficiently.

In other words, we have just described what the intent $\mathcal{I}$ would like to expect from the refinement and partial evaluation of a particular nested intent $\mathcal{I}'$. And how to fulfill such expectation from the perspective of this nested intent?

The way how to do it lies directly in the loop over the reached vertices during the integrated phase of the correction multigraph exploration and the shortest correction paths searching (we use the same horizontal pruning idea as in the exploring strategy). Whenever the evaluation of $\mathcal{I}'$ is requested, we only perform a limited number of iterations of this loop and stop immediately after we have achieved the progress that was requested, i.e. when we are able to provide the parent intent $\mathcal{I}$ with a new and refined estimate.

The result of this estimation always describes the lower bound of the final overall correction cost of the intent repair $\mathcal{R}'$ to which we are step by step approaching. Since we begin with an empty correction multigraph $\mathcal{M}'$ and then we only successively append it by newly explored edges, the cost estimate itself is monotonic. In particular, nondecreasing. And because the parent intent $\mathcal{I}$ always wants to achieve at least some progress of the nested intent $\mathcal{I}'$ evaluation from apparent reasons, we actually implement the refinement in a way that this estimate will even be increasing.

Notice that despite each individual refinement request of a particular intent $\mathcal{I}'$ only aims at a partial evaluation progress, we are still heading towards its complete evaluation. The only difference now is that this effort might be stopped earlier – when a given nested correction intent $\mathcal{I}'$ becomes no longer perspective enough from the point of view its parent $\mathcal{I}$ has.

There are actually only two significant impacts on the correction algorithm we have to appropriately deal with. First, the evaluation of any correction task has to become interruptible and then also allow resuming – in other words, we must be able to scatter the whole evaluation into smaller pieces. Second, the usage of only estimates of the nested correction costs has to be incorporated into the exploration loop – and integrated in a way that it still guarantees to find exactly all the shortest correction paths we are searching for.

Fortunately enough, the solution of the first issue is actually simple. When a nested intent $\mathcal{I}'$ is requested for the refinement evaluation, we then have achieved the expected progress, and so we are about to interrupt our work – we only need to preserve everything we have so far computed, including even all the working and temporary variables. So, when the refinement evaluation of $\mathcal{I}'$ is requested once again sometime later on, we can resume our work exactly at the same point where we left it the last time.

There should be no surprise in realizing that the already introduced container of working variables within correction tasks was designed perfectly in a way to reflect this need. However and contrary to both the previous correction strategies, we now need to use the tasks cache to store the created tasks in all the circumstances, not just when the signatures would be enabled – though we will still need to distinguish between these two signature modes in order to use appropriate indexing keys.

And how the loop over the reached vertices needs to be changed to still guarantee that all the shortest correction paths will be unexceptionally found? We still always select one of the reached vertices v that has currently the minimal possible distance from the source vertex v_S. However, all the edges ingoing to such vertex v may have a different extent of their evaluation – and so the further processing of v will vary depending on whether the tracing distance to v can be declared as final, or whether it still needs to be refined.

In the following text we provide a description of all the individual procedures from which the whole refinement strategy comprises of, as well as we explain all the involved notions and assumptions formally.

Let us therefore begin with the main recursive correction routine that is presented in Algorithm 4.8.

The first apparent change we need to be aware of is the explicit need of splitting the entire procedure into individual evaluation phases. When a particular correction task $\mathcal{K}$ is requested for the evaluation for the first time, nearly all its components are undefined. So is undefined especially its *phase* attribute (line 1), which allow us to correctly recognize a situation when such task has to be appropriately initialized at first (lines 2 – 8).

During this initialization, standard steps are performed. In particular, the container of working variables is prepared (this time including the set of delayed edges), an empty correction multigraph $\mathcal{M}$ is prepared, the source vertex v_S is inserted into it, and both the tracing functions are initialized as well for this source vertex v_S.

The exploration phase may then immediately begin (lines 11 – 17). We first visit the exploration loop over the reached vertices (lines 12), and then we always update costs of the constructed sequence $\mathcal{N}^\circ$ and intent repair $\mathcal{R}^\circ$ structures accordingly (lines 13 and 14). Exactly this way the current value of the correction cost estimate becomes available for the requesting correction intent.

However, we must now start distinguishing between two different ways how the execution of this block of code may come to its end (lines 15 – 17). Either it can be the case of only an interrupted evaluation, or it can mean that the whole evaluation should completely be terminated.

The first means of reaching the end of the exploration block emerges in situations, when the expected and requested refinement progress has been achieved, but the exploration as such can still not be terminated yet (line 17). In this case we do not change the current evaluation phase, and so the following termination block of code is completely bypassed, and the very end of the whole procedure is immediately reached.

In other words, the evaluation has just only been interrupted – and may be

Algorithm 4.8: Refinement strategy: $correct_{\texttt{RFN}}(\mathcal{K})$

Input: Task $\mathcal{K} = (\mathcal{I}, \mathcal{R}^\circ, phase, vars, quota, deps = \bot)$ for a correction intent $\mathcal{I} = (id, type, A, L)$ with an action $A = (p, e, v_I, v_E)$ and an assignment $L = (u, \mathcal{C}, H, Y)$

Global: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$

`// Initialization phase`

if $(phase = \bot)$ **then** $phase \leftarrow$ `initialization`;

if $(phase =$ `initialization`$)$ **then**

- $vars \leftarrow$ initialize working variables ($\mathcal{M}^\circ = \bot$, $V_{reached} = \emptyset$, $c_{reached} = \bot$, $c_{fixed} = \bot$, $pDistance = \emptyset$, $pPredecessors = \emptyset$, $E_{delayed} = \emptyset$) for $\mathcal{K}$;
- $\mathcal{M}^\circ \leftarrow$ create a new empty correction multigraph ($V^\circ = \emptyset$, $E^\circ = \emptyset$, v_S, V_T) with standard v_S and V_T according to intent assignment L;
- $V^\circ \leftarrow V^\circ \cup \{v_S\}$;
- $pDistance(v_S) \leftarrow 0$; $pPredecessors(v_S) \leftarrow \emptyset$;
- $V_{reached} \leftarrow \{v_S\}$;
- $phase \leftarrow$ `exploration`;

Let $vars = (\mathcal{M}^\circ = (V^\circ, E^\circ, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed})$ be working variables for task $\mathcal{K}$;

Let $\mathcal{R}^\circ = (e, \mathcal{N}^\circ, cost^\circ)$ be the currently constructed intent repair and $\mathcal{N}^\circ = (V' = \bot, E' = \bot, v_S = \bot, v_T = \bot, paths = \bot, c^\circ)$ its sequence repair;

`// Exploration phase`

if $(phase =$ `exploration`$)$ **then**

- $performExplorationLoop_{\texttt{RFN}}(\mathcal{K})$;
- **if** $(c_{fixed} \neq \bot)$ **then** $c^\circ \leftarrow c_{fixed}$; **else** $c^\circ \leftarrow c_{reached}$;
- $cost^\circ \leftarrow cost(e) + c^\circ$;
- **if** $(c_{fixed} \neq \bot) \wedge (c_{fixed} < c_{reached})$ **then** $phase \leftarrow$ `termination`;
- **else if** $(V_{reached} = \emptyset)$ **then** $phase \leftarrow$ `termination`;
- **else if** $(quota \neq \bot) \wedge (c_{reached} \geq quota)$ **then return**;

`// Termination phase`

if $(phase =$ `termination`$)$ **then**

- **if** $(E_{delayed} \neq \emptyset)$ **then** $processDelayedEdges_{\texttt{RFN}}(\mathcal{K})$;
- update subgraph (V', E'), v_S, v_T, and $paths$ components of sequence repair $\mathcal{N}^\circ$ using $\mathcal{M}^\circ$ and tracing functions ($pDistance$, $pPredecessors$);
- mark $\mathcal{N}^\circ$ as final and use $\mathcal{N} = (V', E', v_S, v_T, paths, c)$ instead;
- mark intent repair $\mathcal{R}^\circ$ as final and use $\mathcal{R} = (e, \mathcal{N}, cost)$ instead;
- finally put $phase \leftarrow$ `evaluated` and clear working variables $vars \leftarrow \bot$;

resumed later on once again – simply jumping right into the exploration block and continuing with the exploration loop over the reached vertices, resuming it at the same state as it was abandoned the last time.

The second way of reaching the end of the exploration block happens when the searching for the shortest correction paths has successfully been finished inside the exploration loop. Unfortunately, there is not just the standard reason for this conclusion – that the reached cost exceeded the fixed cost (line 15) – but we must newly detect even situations when there is nothing else to be further explored and processed in the set of the reached vertices (line 16).

This is actually not a new circumstance – only in case of both the previous correction strategies we did not need to deal with it explicitly, because it simply happened by leaving the loop, having finished its last iteration.

Anyway, we then move to the termination phase (lines 18 – 23), where we first finalize processing of all the delayed edges as we explain at the very end of this section (line 19), so we are then able to compose the complete and final sequence repair $\mathcal{N}$ and intent repair $\mathcal{R}$ as well. Since both these structures did exist during the whole (though scattered) evaluation, only the explicitly enumerated components have to be updated at this moment (especially note that costs of both $\mathcal{N}$ and $\mathcal{R}$ were updated continuously).

Having done these updates, the entire evaluation of a given correction intent $\mathcal{I}$ finally terminates and its intent repair $\mathcal{R}$ is now complete and available.

Now we move toward Algorithm 4.9 and discuss how the loop over the reached vertices has to be adjusted in order to support the refinement idea and the scattered evaluation. However, before we can provide its description, we need to introduce a few definitions.

We say that an edge $g = (v_1, v_2, \mathcal{I}', \mathcal{R}')$ is a *closed edge*, if the associated intent repair $\mathcal{R}'$ is already completely evaluated, i.e. there exists a corresponding task $\mathcal{K}' = (\mathcal{I}', \mathcal{R}', \mathit{phase}', \mathit{vars}', \mathit{quota}', \mathit{deps}')$ for $\mathcal{I}'$ in the tasks cache such that $\mathit{phase}' = \texttt{evaluated}$. Otherwise g is an *open edge*. The difference and impact is obvious – whereas costs associated to closed edges are final, costs of open edges are only estimates of such final values.

Having a reached vertex v, its extent of refinement depends on its ingoing edges $E_v^{in,\circ}$ (actually only those that are currently explored). For each of these edges $g \in E_v^{in,\circ}$, $g = (v_1, v, \mathcal{I}', \mathcal{R}')$ with $\mathcal{R}' = (e', \mathcal{N}', \mathit{cost}')$ we define an *exploring distance* $\mathit{eDistance}(g) = \mathit{pDistance}(v_1) + \mathit{cost}'$ as a distance to v from the source vertex v_S using an edge g.

Let now $d_v^{in,\circ} = \min_{g \in E_v^{in,\circ}} \mathit{eDistance}(g)$ be the minimal exploring distance to v among all the ingoing edges $E_v^{in,\circ}$. Since v is a reached vertex, $E_v^{in,\circ}$ must contain at least one edge, and so $d_v^{in,\circ}$ may never be undefined. Next, let $E_v^{in,closed,\circ}$ be a set of all the closed ingoing edges to v, and analogously $E_v^{in,open,\circ} = \{g \mid g \in E_v^{in,\circ}$ and g is open$\}$ a set of all the open ingoing edges to v. Then we also define $d_v^{in,closed,\circ} = \min_{g \in E_v^{in,closed,\circ}} \mathit{eDistance}(g)$ as the minimal distance to v when only its closed ingoing edges are considered, and $d_v^{in,open,\circ} = \min_{g \in E_v^{in,open,\circ}} \mathit{eDistance}(g)$ in case of the open ingoing edges respectively.

Finally, we say that v is a *complete vertex*, if there exists at least one closed ingoing edge ($E_v^{in,closed,\circ} \neq \emptyset$), and, at the same time, there are no open ingoing

edges at all ($E_v^{in,open,\circ} = \emptyset$) or the distance to v using any of them would never be better than using the best closed ingoing edge ($d_v^{in,open,\circ} \geq d_v^{in,closed,\circ}$). Otherwise we say that v is an *incomplete vertex*.

When a vertex v is incomplete, it means that there is no closed ingoing edge at all ($E_v^{in,closed,\circ} = \emptyset$) or there exists at least one open ingoing edge that is still perspective and requires further refinement ($d_v^{in,open,\circ} < d_v^{in,closed,\circ}$). When v becomes complete, it means that its final tracing distance from the source vertex v_S has already been correctly detected. Also notice that a vertex may become complete, even when there still may exist open edges, though at most with the same or even worse exploring distances.

Algorithm 4.9: Refinement strategy: $performExplorationLoop_{\texttt{RFN}}(\mathcal{K})$

Input: Task $\mathcal{K} = (\mathcal{I}, \mathcal{R}^\circ, phase, vars, quota, deps = \bot)$ for an intent $\mathcal{I}$

1 Let $vars = (\mathcal{M}^\circ = (V^\circ, E^\circ, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed})$ be working variables for task $\mathcal{K}$;

2 **while** $(V_{reached} \neq \emptyset)$ **do**

3 $v \leftarrow$ select any vertex from $\operatorname{argmin}_{v' \in V_{reached}} d_{v'}^{in,\circ}$;

4 $c_{reached} \leftarrow d_v^{in,\circ}$;

5 **if** $(c_{fixed} \neq \bot) \wedge (c_{fixed} < c_{reached})$ **then break**;

6 **if** $(quota \neq \bot) \wedge (c_{reached} \geq quota)$ **then break**;

7 **if** (*vertex v is complete*) **then** $processCompleteVertex_{\texttt{RFN}}(\mathcal{K}, v)$;

8 **else** $processIncompleteVertex_{\texttt{RFN}}(\mathcal{K}, v)$;

Now we finally have all the details we need to approach the description of the exploration loop itself in Algorithm 4.9.

Although this loop (lines 2 – 8) once again selects the most promising reached vertex v to be processed (line 3), we must be aware of one important difference to the previous correction strategies. Whereas until now we could always rely on values of the *pDistance* tracing function for all the reached vertices (since we kept all its values accurate and up-to-date for all of them) – now we set values for both the tracing functions no sooner than a given vertex becomes complete and is marked processed.

In other words, we have to look at the exploring distances over all the ingoing edges to all the individual reached vertices directly. In particular, we always select a vertex v that has this distance $d_v^{in,\circ}$ minimal, regardless such distance is achieved by an open edge, or an already closed one. This exploring distance for v then becomes the currently reached cost (line 4).

The termination condition (line 5) once again ensures that when the reached cost exceeds the fixed one, all the shortest correction paths must have already been found, and so the multigraph exploration may come to its end. On the other hand, a completely exhausted quota suggests that the expected and requested refinement progress has been just attained, and so the evaluation is about to be interrupted (line 6).

The further processing of vertex v depends on whether it is already complete (line 7), or not yet (line 8) – i.e. whether it can be finalized, or not.

Algorithm 4.10: Refinement strategy: $processCompleteVertex_{\texttt{RFN}}(\mathcal{K}, v)$

Input: Task $\mathcal{K} = (\mathcal{I}, \mathcal{R}^\circ, phase, vars, quota, deps = \bot)$ for an intent $\mathcal{I}$, and a complete vertex v
Global: Data tree $\mathcal{T}$ and regular tree grammar $\mathcal{G}$

1 Let $vars = (\mathcal{M}^\circ = (V^\circ, E^\circ, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed})$ be working variables for task $\mathcal{K}$;
2 $V_{reached} \leftarrow V_{reached} \setminus \{v\}$;
3 **if** $(v \in V_T) \wedge (c_{fixed} = \bot)$ **then** $c_{fixed} \leftarrow c_{reached}$;
4 $pDistance(v) \leftarrow d_v^{in,closed,\circ}$;
5 $pPredecessors(v) \leftarrow \{g \mid g \in E_v^{in,closed,\circ}, eDistance(g) = d_v^{in,closed,\circ}\}$;
6 $E_{delayed} \leftarrow E_{delayed} \cup \{g \mid g \in E_v^{in,open,\circ}, eDistance(g) = d_v^{in,closed,\circ}\}$;
7 **foreach** $\mathcal{I}' \in NestedIntents(\mathcal{I}, v)$ **do**
8 Let $\mathcal{I}' = (id', type', A', L')$ and $A' = (p', e', v'_I = v, v'_E)$;
9 $\mathcal{R}'^\circ \leftarrow requestIntentRepair_{\texttt{RFN}}(\mathcal{I}')$;
10 **if** $(v'_E \notin V^\circ)$ **then** $V^\circ \leftarrow V^\circ \cup \{v'_E\}$; $V_{reached} \leftarrow V_{reached} \cup \{v'_E\}$;
11 $g \leftarrow (v, v'_E, \mathcal{I}', \mathcal{R}'^\circ)$; $E^\circ \leftarrow E^\circ \cup \{g\}$;
12 Let $\mathcal{R}'^\circ = (e', \mathcal{N}'^\circ, cost'^\circ)$;
13 **if** $(v'_E \notin V_{reached}) \wedge (cost'^\circ = 0)$ **then** $E_{delayed} \leftarrow E_{delayed} \cup \{g\}$;

We first discuss a situation when the given selected reached vertex v is complete. Its further processing is then depicted in Algorithm 4.10.

Since no additional refinement of this vertex v is required, we finalize it by removing it from the set of reached vertices (line 2), setting values of both the tracing functions appropriately (lines 4 – 6), and exploring all the outgoing edges from v, i.e. all the nested correction intents defined at v (lines 7 – 13).

We also verify the terminating condition on reaching the first target vertex in a standard way, i.e. by anchoring the currently reached cost (line 3).

However, before we move forward, we stay with the mentioned tracing functions for a while. Since v is complete, we know that $d_v^{in,closed,\circ} = d_v^{in,\circ}$, and so $pDistance(v)$ cannot be improved later on, and so can be finalized right at this moment (line 4). To derive the $pPredecessors(v)$, we only focus on the ingoing edges that are closed and that have this minimal exploration distance (line 5). However, even now there might exist some edges that are open, but still perspective enough (line 6).

Though they no longer can influence the tracing distances themselves in any way, they can still be involved in the shortest correction paths we are searching for, and so we cannot simply ignore them. On the other hand, they are not needed to be completely evaluated at this moment, so we decided to put all of them in

to a special container of delayed edges, i.e. edges that might still be perspective enough, but detecting whether they really are or not can be postponed until the very end of the intent $\mathcal{I}$ evaluation.

Next, let us focus on the loop over the newly discovered nested correction intents (lines 7 – 13). When comparing it to the equivalent loop of the exploring strategy, we once again request the nested intent repair $\mathcal{R}'^{\circ}$ (line 9), create a new ending vertex v'_E in case it was not yet explored (line 10), always create a new edge g for $\mathcal{I}'$ (line 11), but we do not alter the tracing functions for v'_E here as already outlined.

On the other hand, we must not forget to deal with newly discovered edges in case they lead to some of the already processed complete vertices, but still have a perspective cost (line 13). In particular, we put all such edges to the container of delayed edges once again. Notice that we must treat those edges explicitly here – simply because the already referenced delaying code (line 6) would not be reachable for them, since their ending vertex v'_E has already been processed as we assumed.

The question now is, how the nested intent repair $\mathcal{R}'^{\circ}$ is actually requested (line 9). The answer lies in Algorithm 4.11.

Algorithm 4.11: Refinement strategy: $requestIntentRepair_{\texttt{RFN}}(\mathcal{I}')$

Input: Nested correction intent $\mathcal{I}' = (id', type', A', L')$
Output: Intent repair $\mathcal{R}'^{\circ}$ for $\mathcal{I}'$
Use: $TasksCache$ regardless the mode of signatures

```
1 if (signatures are enabled) then
2     if (∃ t' such that (sig(I'), t') ∈ TasksCache) then  K' ← t';
3     else
4         K' ← createPreparedTask_RFN(I');
5         TasksCache ← TasksCache ∪ {(sig(I'), K')};
6 else
7     K' ← createPreparedTask_RFN(I');
8     TasksCache ← TasksCache ∪ {(id', K')};
9 return R'°;
```

Its basic structure brings nothing strange, because we only need to distinguish between enabled (lines 1 – 5) and disabled (lines 6 – 8) handling of signatures and caching of evaluated intent repairs. However, there are two important differences to the previous strategies.

First, we have to cache tasks even when signatures are disabled, otherwise we would not be able to preserve them and their content between individual refinement evaluation requests. In this case, an intent identifier id' is used as a key when accessing the tasks cache (line 8).

The second difference – and even more fundamental one in its consequences and advantages it brings – is that we never invoke the recursive correction routine for the nested task $\mathcal{K}'$ right here. It means that the only purpose of this procedure is to fetch the already existing task, or create a new one, but never proceed to requesting its evaluation as such.

The reason for omitting this evaluation is simple – it is because the first cost estimate of the nested intent repair $\mathcal{R}'^{\circ}$ can be based only on the associated repairing instruction e' (if any) for this nested intent $\mathcal{I}'$. In other words, we are able to acquire the first estimate even without any recursive refinement actually requested and nor executed.

Even though it might not be apparent here, this observation causes the overall correction efficiency to improve significantly.

Arm in arm, when creating a new correction task for the nested intent $\mathcal{I}'$ this time, we do not just create a completely empty task structure $\mathcal{K}'$, but we initialize all its components that allow us to publish this first and kind of a free of charge estimation, as well as permit its further refinement later on.

Algorithm 4.12: Refinement strategy: $createPreparedTask_{\texttt{RFN}}(\mathcal{I}')$

Input: Nested correction intent $\mathcal{I}' = (id', type', A', L')$ and its intent action $A' = (p', e', v'_I, v'_E)$

Output: Newly created correction task $\mathcal{K}'$ for $\mathcal{I}'$

1 $\mathcal{N}'^{\circ} \leftarrow$ create a new partly initialized sequence repair ($V' = \bot$, $E' = \bot$, $v'_S = \bot$, $v'_T = \bot$, $paths' = \bot$, $c'^{\circ} = 0$);

2 $\mathcal{R}'^{\circ} \leftarrow$ create a new partly initialized intent repair (e', $\mathcal{N}'^{\circ}$, $cost'^{\circ} = c'^{\circ} + cost(e') = cost(e')$);

3 **return** $\mathcal{K}' \leftarrow (\mathcal{I}', \mathcal{R}'^{\circ}, phase' = \bot, vars' = \bot, quota' = \bot, deps' = \bot)$;

In particular and as we can see in Algorithm 4.12, we first initialize a sequence repair $\mathcal{N}'^{\circ}$ with all its components undefined except for its cost c'° (line 1). Then we prepare an intent repair $\mathcal{R}'^{\circ}$ with the repairing instruction e' and the overall correction cost set accordingly (line 2), so that we can finally encapsulate it into a newly created task $\mathcal{K}'$ and return it in the end (line 3).

When the loop over the reached vertices selects an incomplete vertex v to be processed (line 8 of Algorithm 4.9), then we have to refine its open ingoing edges by requesting their further evaluation. So, let us have a look at Algorithm 4.13, where this processing is illustrated.

The refinement expectations posed on the nested correction intent $\mathcal{I}'$ are determined directly by the requesting parent intent $\mathcal{I}$. Since it requires its further evaluation, it certainly presumes that at least some progress in the cost estimate of $\mathcal{R}'^{\circ}$ will be achieved – otherwise no new information would be available to $\mathcal{I}$. On the other hand, the allowed partial evaluation of $\mathcal{I}'$ should be limited as much as possible in order to avoid any unnecessary work.

For the purpose of limiting the extent of such permitted and expected nested refinement we use the assigned quota, an argument forwarded via a *quota*

component within a structure of the given task $\mathcal{K}'$. The meaning of this quota is straightforward – it contains a boundary of a sequence repair $\mathcal{N}'$ cost to which the nested evaluation of $\mathcal{I}'$ should approach, but not exceed. This indirectly means that we restrain the extent of the allowed nested evaluation by a permitted growth of the cost estimate we are step by step pursuing to refine.

Just notice that for technical reasons, assigned quotas are really based on the cost of a sequence repair $\mathcal{N}'$ only, and do not involve the impact of a repairing instruction e' from $\mathcal{R}'$. As a consequence, quotas can be directly matched to the reached costs valid within the exploration loop.

Although the value of the assigned quota is a matter of a suitable heuristic, we decided to always expect the cost increase of 1, since we also assumed the cost of all the edit operations to be equal to 1. Having a different cost function in use, quotas to be assigned would need to be treated carefully. Anyway, the assigned quota always has to be sharply greater than the current sequence cost estimate. Otherwise no nested progress would be possible to achieve.

From the nested intent $\mathcal{I}'$ point of view, its exploration loop over the reached vertices (see Algorithm 4.9) iterates until the assigned quota is exhausted, as already explained. At that moment its exploration is interrupted. But in case $\mathcal{I}'$ can be evaluated completely and its final intent repair $\mathcal{R}'$ acquired within the allowed assigned quota, it is always done so.

Algorithm 4.13: Refinement strategy: $processIncompleteVertex_{\texttt{RFN}}(\mathcal{K}, v)$

Input: Task $\mathcal{K} = (\mathcal{I}, \mathcal{R}^\circ, phase, vars, quota, deps = \bot)$ for an intent $\mathcal{I}$, and an incomplete vertex v

Use: $TasksCache$ regardless the mode of signatures

1 Let $vars = (\mathcal{M}^\circ = (V^\circ, E^\circ, v_S, V_T), V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed})$ be working variables for task $\mathcal{K}$;

2 $oEdges \leftarrow \{g \mid g \in E_v^{in,open,\circ}, eDistance(g) = d_v^{in,open,\circ}\}$;

3 **foreach** $g = (v_1, v_2 = v, \mathcal{I}', \mathcal{R}'^\circ) \in oEdges$ **do**

4 Let $\mathcal{I}' = (id', type', A', L')$ and $A' = (p', e', v'_I = v_1, v'_E = v_2)$;

5 **if** (*signatures are enabled*) **then** $key' \leftarrow sig(\mathcal{I}')$; **else** $key' \leftarrow id'$;

6 Let $\mathcal{K}'$ be a task such that $(key', \mathcal{K}') \in TasksCache$;

7 Let then $\mathcal{K}' = (\mathcal{I}', \mathcal{R}'^\circ, phase', vars', quota', deps')$ and

8 $\mathcal{R}'^\circ = (e', \mathcal{N}'^\circ, cost'^\circ)$;

9 $quota' \leftarrow cost'^\circ - cost(e') + 1$;

10 $correct_{\texttt{RFN}}(\mathcal{K}')$;

To make the insight into the mechanism of assigned quotas complete, let us also shortly look at the processing of the starting correction intent $\mathcal{I}_\bullet$. Contrary to all the other correction intents, this one is assigned an *unlimited quota* (technically $quota = \bot$). More precisely, it must be assigned this unlimited quota, since this starting correction intent is responsible for conducting the entire data tree

correction, and so cannot interrupt its work somewhere in the middle – not having finished it completely.

Finally, how the assigned quotas are turned into practice during the processing of an incomplete vertex v (Algorithm 4.13)? First of all, we always consider only the most perspective open edges, i.e. those open edges g ingoing to v such that they have the exploring distance minimal (line 2). We could, of course, consider even other open edges to be refined as well, but our selection avoids any necessary work completely.

Then we process these most perspective edges in a loop (lines 3 – 10), one after another, always accessing the associated correction task $\mathcal{K}'$ from the tasks cache (line 6), assigning the allowed quota (line 9), and finally requesting the nested partial evaluation of $\mathcal{I}'$ as such (line 10).

The very last thing we need to discuss before we can conclude the whole refinement correction strategy is the way how all the delayed edges are processed before the given intent evaluation can be finally terminated. The code of this procedure is available in Algorithm 4.14.

Algorithm 4.14: Refinement strategy: $processDelayedEdges_{\texttt{RFN}}(\mathcal{K})$

Input: Task $\mathcal{K} = (\mathcal{I}, \mathcal{R}^\circ, phase, vars, quota, deps = \bot)$ for an intent $\mathcal{I}$
Use: $TasksCache$ regardless the mode of signatures

1 Let $vars = (\mathcal{M}^\circ, V_{reached}, c_{reached}, c_{fixed}, pDistance, pPredecessors, E_{delayed})$ be working variables for task $\mathcal{K}$;
2 **foreach** $g \leftarrow (v_1, v_2, \mathcal{I}', \mathcal{R}'^\circ) \in E_{delayed}$ **do**
3 Let $\mathcal{I}' = (id', type', A', L')$ and $A' = (p', e', v'_I = v_1, v'_E = v_2)$;
4 **if** (*signatures are enabled*) **then** $key' \leftarrow sig(\mathcal{I}')$;
5 $key' \leftarrow id'$; Let $\mathcal{K}'$ be a task such that $(key', \mathcal{K}') \in TasksCache$;
6 Let then $\mathcal{K}' = (\mathcal{I}', \mathcal{R}'^\circ, phase', vars', quota', deps')$ and
7 $\mathcal{R}'^\circ = (e', \mathcal{N}'^\circ, cost'^\circ)$;
8 **if** $(eDistance(g) > pDistance(v_2))$ **then break**;
9 **if** $(phase' \neq \texttt{evaluated})$ **then**
10 $quota' \leftarrow cost'^\circ - cost(e') + 1$;
11 $correct_{\texttt{RFN}}(\mathcal{K}')$;
12 **if** $(eDistance(g) = pDistance(v_2))$ **then**
13 $pPredecessors(v_2) \leftarrow pPredecessors(v_2) \cup \{g\}$;
14 clear the set of delayed edges $E_{delayed} \leftarrow \emptyset$;

We simply iterate over all the delayed edges g (lines 2 – 13), always accessing the corresponding correction task $\mathcal{K}'$ at first (line 5). Though all the edges were still open and perspective at the moment they were put into this set of delayed

edges, further flow of events could cause that they are no longer perspective right now (line 8).

If a given edge g is still perspective ($eDistance(g) = pDistance(v_2)$), and it is not yet completely evaluated at the same time (lines 9 – 11), we request its one final refinement evaluation (line 11) by assigning a quota that is high just enough to give a chance for such $\mathcal{I}'$ to terminate its evaluation completely right at this cost (and so a chance this edge g to become closed), or to disqualify itself from being perspective at all (line 10).

If this final refinement (if required) caused this edge g to become closed and its exploring distance is still equivalent to the tracing distance of its ending vertex v_2 (line 12), we can add this edge between the already found predecessors defined for such vertex v_2 (line 13).

Having explained how the postponed evaluation attempt of the delayed edges works, we have also completed the description of the whole refinement strategy.

Its core idea is actually based just on one important change – when searching for the shortest correction paths, we do no longer rely on completely evaluated nested intents and overall correction costs of their fully resolved intent repairs, but instead we work only with estimates of such nested costs.

As a consequence, we are not only able to optimize the horizontal multigraph exploration – by integrating the construction and exploration phases together once again as in case of the exploring strategy – but we are also able to achieve the vertical optimization – by pruning less perspective nesting of correction intents even to the depth of the recursion.

As a whole, there is a high chance that this strategy will be the most efficient one, yet this conclusion cannot be accepted without a thorough experimental evaluation, because although the number all the created tasks will most likely be reduced significantly, the overhead brought by shattering the evaluation into small pieces may not be suppressed sufficiently.

4.4 Execution Approaches

While correction strategies represent different answers to the question how correction multigraphs should be explored, the purpose of *execution approaches* is to find ways how the recursive evaluation requests of correction intents should actually be performed from the technical point of view. In other words, which programming constructs or system resources could be used to implement correction strategies in practice.

We proposed five particular execution approaches – *nesting single*, *invoking single*, *invoking multiple*, *forwarding single*, and, finally, *forwarding multiple*. The purpose of the following text is to discuss their basics, differences, and shared characteristics as well.

4.4.1 Nesting Single Approach

The first execution approach we focus on is the *nesting single*. Though it is probably the most simple and straightforward one, it is still the most efficient single-threaded one on the other hand.

Assume that we are evaluating a particular correction task $\mathcal{K}$ for a correction intent $\mathcal{I}$. When a task $\mathcal{K}'$ for a nested correction intent $\mathcal{I}'$ is about to be requested for the execution, we simply and directly call the recursive correction routine $correct(\mathcal{K}')$ (its particular alternative according to a chosen correction strategy), exactly as we have done it in all the algorithms we have presented so far.

The context is switched, and the nested execution of $\mathcal{K}'$ starts immediately. When the requested evaluation goal is achieved (i.e. when the intent repair $\mathcal{R}'$ is fully constructed in case of the complete evaluation of the default and exploring strategies, or when we the expected and requested refinement progress is achieved in case of the partial evaluation of the refinement strategy), the very end of the nested correction routine is simply reached, and so the context is automatically returned back to the execution of $\mathcal{K}$. As a consequence, the evaluation of $\mathcal{K}$ resumes at exactly the same point where it was abandoned before, and so it may smoothly and immediately go on.

All the particular algorithms we presented in the previous section for all the introduced correction strategies were designed perfectly in a way to comply with this nesting single approach.

4.4.2 Invoking Single and Multiple Approaches

Let us now present two additional execution approaches we call *invoking single* and *invoking multiple*. They both enable us to explicitly utilize system threads to perform the evaluation of correction tasks, but whereas the former one only allows us to request right one such evaluation at a time, the latter one can request even multiple task at once.

We begin with the invoking single approach first. So, let the correction task $\mathcal{K}$ for an intent $\mathcal{I}$ be currently executed in a thread W. When a nested task $\mathcal{K}'$ for some $\mathcal{I}'$ is requested to be executed, a new thread W' is created, associated with $\mathcal{K}'$, and its execution is commenced, i.e. W' starts executing the correction routine $correct(\mathcal{K}')$. The parent thread W becomes blocked on $\mathcal{K}'$ and starts waiting until it can be woken up once again.

When the nested thread W' achieves the requested evaluation goal posed on $\mathcal{K}'$, it notifies the parent thread W on behalf of the task $\mathcal{K}'$ in order to wake it up. Then the nested thread W' simply reaches the end of the correction routine, it terminates its execution and is released. When the parent thread W is woken up, its execution resumes exactly at the point where it was abandoned, and the evaluation of $\mathcal{K}$ may continue.

As a consequence, at each moment there exists a chain of nested threads, since we always only request right one the nested task to be executed. So, we are obviously not able to exploit the potential which multi-threaded processing can generally offer.

The purpose of the invoking multiple approach is to deal right with this drawback. In other words, we would like to be able to emit more evaluation requests at a time, each to be executed by a separate thread. However, two questions immediately arise. First, how many such requests should be considered at once. And second, how to actually identify and select requests that should be executed

together – all that with respect to the correction strategies we have introduced and their specifics.

Despite plenty of different mechanisms could generally be adopted, we decided for the following solution. In case of the default and exploring strategies, when a particular vertex v is being processed within the exploration loop over the reached vertices during the evaluation of a correction intent $\mathcal{I}$ (Algorithm 4.5 and Algorithm 4.7 respectively), we take as a basis for this purpose the whole set of the nested correction intents defined at v, i.e. $NestedIntents(\mathcal{I}, v)$ (line 8 and line 8 respectively in these algorithms).

In case of the exploration phase of the refinement strategy and processing of a reached vertex v that is incomplete (Algorithm 4.13), we take into account all the most perspective open edges ingoing to v, i.e. a set $\{g \mid g \in E_v^{in,open,\circ}, eDistance(g) = d_v^{in,open,\circ}\}$ (line 3). Finally, in case of the processing of the delayed edges (Algorithm 4.14), all those that are still perspective and not yet refined enough at the same moment are right those that are considered here (line 9).

So, assume that we have identified a set of correction intents $\{\mathcal{I}'_1, \ldots, \mathcal{I}'_n\}$ for some $n \in \mathbb{N}_0$, and requests right for all these intents are about to be invoked together. Unfortunately, we now need to start distinguishing between both the intent signature modes, since they bring different levels of complexity and issues to be figured out.

First, let intent signature handling be disabled. Then assume we are currently evaluating a correction task $\mathcal{K}$ in a thread W, and we are about to request evaluation of nested tasks $\{\mathcal{K}'_1, \ldots, \mathcal{K}'_n\}$ for the identified set of correction intents $\{\mathcal{I}'_1, \ldots, \mathcal{I}'_n\}$. So for each and every one of them, i.e. for each $i \in \{1, \ldots, n\}$, we create a new thread W'_i, associate it with $\mathcal{K}'_i$, and commence its execution. Then the parent thread W itself becomes blocked on all $\mathcal{K}'_i$ and starts waiting until it can be woken up once again.

When each particular nested thread W'_i achieves its evaluation goal posed on the given task $\mathcal{K}'_i$, and the notification to W is sent on behalf of such $\mathcal{K}'_i$, the nested thread W'_i terminates its execution and is released. Once the parent thread W receives notifications from all the nested threads W'_i, i.e. is no longer blocked on any requested $\mathcal{K}'_i$, it is woken up, its execution resumes, and the evaluation of $\mathcal{K}$ may continue.

In case intent signatures are enabled, the situation becomes a little bit more complicated because of the following reason. Not only that each particular task $\mathcal{K}$ may request the evaluation of multiple nested tasks $\mathcal{K}'$ at once, but also – and as a consequence – each particular task $\mathcal{K}'$ may then be requested for the evaluation by several different parental tasks $\mathcal{K}$ at once. This means that we must ensure that each particular task is being executed at most once at each moment, i.e. that its execution cannot be accidentally commenced for several times concurrently.

Assume therefore that we are currently evaluating a correction task $\mathcal{K}$ in a thread W, and we are about to request the evaluation of nested correction tasks $\{\mathcal{K}'_1, \ldots, \mathcal{K}'_n\}$ for some $n \in \mathbb{N}_0$, such that all these nested tasks have mutually different intent signatures.

Then for each and every one of them, i.e. for each $i \in \{1, \ldots, n\}$, we first find out whether there already exists a thread W'_i that is currently running and executing $\mathcal{K}'_i$. If and only if not, we create a new thread W'_i, associate it with

$\mathcal{K}'_i$, and commence its execution. Once all the requests are managed, the parent thread W itself becomes blocked on all $\mathcal{K}'_i$ (really all, regardless we newly created W'_i or just sort of joined an already existing one), and starts waiting until it can be woken up once again.

When each particular nested thread W'_i achieves its evaluation goal posed on the given task $\mathcal{K}'_i$, it sends notifications to all parent threads W'' (one of them must be W, but there can be even others) on behalf of such $\mathcal{K}'_i$, the nested thread W'_i itself terminates its execution and is released. Once the parent thread W receives notifications from all the nested threads W'_i, i.e. is no longer blocked on any requested $\mathcal{K}'_i$, it is woken up, its execution resumes, and the evaluation of $\mathcal{K}$ may continue.

Note also that there is still one technical aspect we need to be aware of. Because of the multi-threaded environment, usage and access to the task cache as well as the management of threads itself have to be correctly synchronized in order to prevent race conditions potentially leading to unexpected failures or unwanted behavior.

To finally summarize the invoking multiple execution approach, at each moment there exists a whole hierarchy of nested threads, and so the correction is performed in parallel. On the other hand, the number of threads that need to be concurrently maintained by the system might be only too high, and hence such execution approach may generally represent an efficiency bottleneck.

4.4.3 Forwarding Single and Multiple Approaches

The last pair of execution approaches we are going to introduce right now is a pair of *forwarding single* and *forwarding multiple* approaches. Whereas their objective is to deal right with the previously identified issue, our solution is actually not difficult to grasp.

Whenever the nested execution is requested, the requesting thread does not become blocked, and so does not start waiting for the notifications – instead it immediately terminates its own execution and is released. However, losing the reference to this parent thread brings two questions that need to be figured out properly. First, how to resume the execution when backtracking, and, second, how to actually manage all the request relationships among tasks when the implicit blocking and notification mechanism can no longer be used.

The response to the former question is simple – we only need to create a new thread that commits itself to the responsibility of resuming the parent task evaluation. The solution of the latter question is brought by a container of *request dependencies*, i.e. the last component of a task structure we have not yet talked about. Using them, we are able to straightforwardly record and maintain all the task requests in a convenient way.

Technically, looking at a particular task $\mathcal{K}$, this container comprises of two sets: a set of *requesting tasks* $K_{requesting}$ (all the tasks that have currently requested the evaluation of $\mathcal{K}$), and a set of *requested tasks* $K_{requested}$ (all the tasks that $\mathcal{K}$ have currently requested for the evaluation).

Last but not least, there is actually yet another important consequence of the new forwarding execution approaches: regardless the correction strategy we use,

the execution now needs to become scattered – similarly as we had to scatter the evaluation in case of the refinement strategy as such.

At this moment we are ready to illustrate how the forwarding single approach in particular works. Suppose therefore that we are evaluating a correction task $\mathcal{K}$ in a thread W, and we want to request evaluation of a nested task $\mathcal{K}'$. We first add $\mathcal{K}'$ into the set of requested tasks of $\mathcal{K}$. Then we create a new thread W', associate it with $\mathcal{K}'$, add $\mathcal{K}$ into the set of requesting tasks of $\mathcal{K}'$, and, finally, commence the execution of W'. Then the parent thread W terminates its execution and is released.

When the nested thread W' achieves the requested evaluation goal posed on the task $\mathcal{K}'$, we first remove $\mathcal{K}$ from the set of requesting tasks of $\mathcal{K}'$, and also $\mathcal{K}'$ from the set of requested tasks of $\mathcal{K}$ analogously. Then a new thread W is created, associated with $\mathcal{K}$, and its execution commenced, so that the evaluation of $\mathcal{K}$ may continue. Finally, the nested thread W' terminates its execution and is released.

Let us now move to the forwarding multiple execution approach. Though we could once again discuss both the cases depending on intent signatures separately, we rather describe only the most complicated case, the one with enabled signatures.

So, assume we are currently evaluating a correction task $\mathcal{K}$ in a thread W, and we are about to request evaluation of nested tasks $\{\mathcal{K}'_1, \ldots, \mathcal{K}'_n\}$ for some $n \in \mathbb{N}_0$, such that all these nested tasks have mutually different intent signatures. First, we add all these $\mathcal{K}'_i$ into the set of requested tasks of $\mathcal{K}$.

Then for each and every one of them, i.e. for each $i \in \{1, \ldots, n\}$, we first find out whether there already exists a thread W'_i that is currently running and executing $\mathcal{K}'_i$. If and only if not, we create a new thread W'_i, associate it with $\mathcal{K}'_i$, and commence its execution. Regardless we newly created such W'_i or just sort of joined an existing one, we always add $\mathcal{K}$ into the set of requesting tasks of $\mathcal{K}'_i$. Finally, when all the $\mathcal{K}'_i$ are requested this way, the parent thread W itself terminates its execution and is released.

When each particular nested thread W'_i achieves its evaluation goal posed on the given task $\mathcal{K}'_i$, we first iterate over all the tasks $\mathcal{K}''$ that are stored in the set of requesting tasks such $\mathcal{K}'_i$ (one of them must be $\mathcal{K}$, but there can be even others), remove itself (i.e. $\mathcal{K}'_i$) from the set of requested tasks of such $\mathcal{K}''$, and in case this set becomes empty (i.e. $\mathcal{K}''$ no longer symbolically waits for any requested task), we create a new thread W'', associate it with $\mathcal{K}''$, and commence its execution.

Having iterated over all such requesting $\mathcal{K}''$, the set of requesting tasks of $\mathcal{K}'_i$ is emptied, the nested thread W'_i itself terminates its execution and is released. As a consequence, once all K'_i originally requested by $\mathcal{K}$ achieve their evaluation goals, the evaluation of $\mathcal{K}$ is thus resumed.

In case of the forwarding multiple approach with signatures disabled, the situation becomes easier. First, we can always straightforwardly create new threads for the nested requested tasks without any concurrency checks. Second, there is always right one requesting parent to be symbolically notified when terminating the nested execution.

Anyway and similarly to the invoking multiple approach, we must once again

be aware of the consequences of the multi-threaded environment. So, not only the usage and access to the tasks cache, but also all the containers of request dependencies of individual tasks have to be correctly synchronized here.

Moreover, this synchronization must be implemented in a way that is does not become a bottleneck. We must also avoid the possibility of causing deadlocks if we realize that these dependency containers may be mutually modified either by requesting, as well as by requested tasks at the same moment.

Fortunately enough, though with certain technical difficulties, this issue can be figured out by breaching one of the Coffman conditions [29] – the circular wait condition in particular. So, we always synchronize first on the requesting task, and only then on the requested one.

Last but not least, there is also one final aspect of the forwarding approaches that deserves attention. Although we have so far always talked about creation of new threads, it is apparent that a pool of prepared worker threads could be used as well. Thus we not only reduce the required overhead related to the management of threads in general, but we also gain the possibility to get in charge and manually or automatically control the overall number of tasks that can really be executed right at each particular moment.

To conclude, while there was always a chain of nested threads in case of the invoking single approach, there is just one thread in case of the forwarding single approach (or at least nearly one). Similarly, whereas there could be a whole hierarchy of threads in case of the invoking multiple approach, there is always just a set of threads in case of the forwarding multiple approach. However, although we were able to reduce the number of threads in general, this does not necessarily mean that the forwarding approaches will perform better than the invoking ones in practice.

4.5 Algorithm Configurations

Having introduced all the correction strategies, execution approaches and signature modes, we have finished with the description of all the correction algorithms we have proposed on top of our correction model. At the very end, we try to summarize all of them, discuss their most important characteristics as well as differences, and prepare a background for the experimental evaluation.

First, let us have a look at the proposed correction strategies. Their purpose is to provide different ways how the correction multigraphs can be explored and the shortest correction paths searched. In particular, which vertices and edges of correction multigraphs need to be explored, in order such paths are found, and found correctly.

- `DEF` – Default strategy. This first and most straightforward strategy directly follows all the definitions of the correction model as we have introduced them. This means that an entire correction multigraph with all its edges and vertices is always constructed, and so intent repairs for all the corresponding nested correction intents are always evaluated.

- `EXP` – Exploring strategy. In order to reduce the overall number of created correction intents, and so the number of the nested intents that need to

be evaluated, this strategy involves a horizontal pruning optimization. In particular, it always starts with an empty correction multigraph, and only explores those vertices and edges that are really required to be considered in order to still guarantee that all the shortest correction paths are found.

- `RFN` – Refinement strategy. Instead of assuming that all the nested correction intents always have to be evaluated completely, this strategy is able to rely only on partially evaluated nested intent repairs and estimates of their costs. As a consequence, the overall number of created intents is not only reduced because of the horizontal pruning applied, but because of the vertical pruning as well. In other words, this strategy is able to leave unpromising ways of the correction even to the depth of the recursive nesting of correction intents.

Table 4.1: Comparison of correction strategies

Strategy code	Intent evaluation	Multigraph exploration	Nested repairs	Edge costs
`DEF`	Complete	Entire	Full	Final
`EXP`	Complete	Pruned	Full	Final
`RFN`	Limited	Pruned	Partial	Estimated

Strategy code	Processed vertices	Requested edges	Execution style
`DEF`	Any	All outgoing	Continuous
`EXP`	Any	All outgoing	Continuous
`RFN`	Incomplete	Minimal open ingoing	Scattered

In Table 4.1 we list all the important characteristics in which the individual correction strategies mutually differ, or behave the same on the contrary.

The evaluation goal of both the default and exploring strategies is to always evaluate the given correction intent completely. It means to produce its fully evaluated intent repair with a final overall correction cost, and so the execution of these strategies does not need to be scattered. When processing a particular reached vertex, the recursive evaluation of all the nested intents on the outgoing edges is requested. However, whereas the default strategy always constructs the entire correction multigraphs with all their vertices and edges, the exploring one only needs to discover multigraphs on demand.

On the other hand, the refinement strategy builds on top of the evaluation limited by assigned quotas, and so leading only to partially evaluated intent repairs with overall correction costs just being estimated. As a consequence, the execution must be scattered. In the loop over the reached vertices, only the incomplete ones are involved in requesting the nested evaluation, in particular the evaluation of only the most perspective open ingoing edges.

Next, let us also have a look at the execution approaches. They represent different means how correction strategies can be implemented in practice from the

technical point of view, i.e. they discuss which particular programming constructs and system resources can be used to fulfill this implementation goal.

- `N1` – Nesting single. When a nested correction task is about to be requested for the evaluation, the recursive correction routine is simply called. When the expected evaluation goal is achieved, the context is returned back, and the execution immediately continues. All that in one thread.

- `I1` – Invoking single. When a nested task is about to be requested for the evaluation, a new thread is created and its execution commenced. The parent thread becomes blocked and starts to sleep. When the expected evaluation goal is achieved, the parent thread is notified, and the nested one terminates and is released. Then the parent thread is woken up and its execution is resumed.

- `IN` – Invoking multiple. The basic idea is the same, only there can be more nested tasks to be requested at a time. As a consequence, the parent thread becomes blocked on all of them, and so is woken up no sooner than the last nested evaluation goal is achieved. When signatures are enabled, new nested threads must be created carefully to avoid competing executions, and when the nested goals are achieved, there can be more parents as well.

- `F1` – Forwarding single. When a nested task is about to be requested for the evaluation, a new thread is created and the request dependencies are mutually recorded. Then the execution of the nested thread is commenced, the parent thread terminates its execution and is released. When the expected evaluation goal is achieved, a new thread is created for the parent and the request dependencies are mutually removed. Then the execution of the parent thread is commenced, the nested thread terminates its execution and is released.

- `FN` – Forwarding multiple. Once again, the basic idea is the same as in case of the forwarding single approach, except that there can be more nested tasks to be requested at a time. Similarly to the invoking multiple approach, competing executions have to be avoided, as well as there can be more parents that need to be treated accordingly.

Finally, we summarize both the intent signature modes.

- `D` – Disabled signatures. When a nested correction intent is explored, its evaluation is always requested.

- `E` – Enabled signatures. When a nested intent is about to be requested, we first try to fetch an already computed equivalent intent repair from the tasks cache using the corresponding intent signature, and we only proceed to the evaluation if and only if such repair is not yet available.

Putting it all together, we have introduced three different correction strategies $\Psi_{Stg} = \{\texttt{DEF}, \texttt{EXP}, \texttt{RFN}\}$, five execution approaches $\Psi_{App} = \{\texttt{N1}, \texttt{I1}, \texttt{IN}, \texttt{F1}, \texttt{FN}\}$, and also two signature handling modes $\Psi_{Sig} = \{\texttt{D}, \texttt{E}\}$.

We are absolutely free to combine them in order to obtain particular implementations of the correction algorithm – different *algorithm configurations* $\mathcal{Q} \in (\Psi_{Stg} \times \Psi_{App} \times \Psi_{Sig})$. It is obvious that neither all of them make sense from the practical point of view, nor that all of them would be interesting even from the theoretical perspective. However, at least several of them have reasonably good chances to be efficient enough, some of them even very efficient and scalable.

All the abbreviated codes of the introduced correction strategies, execution approaches and signature modes were not presented without purpose. They enable us to give names to particular algorithm configurations in a convenient way. So, we either can use the already suggested system of triples $\mathcal{Q}$, or we will also, and actually more frequently, use a more practical naming schema `?-?-?` with components for a chosen strategy, approach and mode, all that exactly in this order. For example, `DEF-N1-D` represents the configuration involving the default strategy with nesting single approach and disabled signatures.

At this moment it makes sense to recall our older correction algorithms [83, 86, 85] and describe them in terms of the algorithm configurations as we have just introduced them in this text and as they were also published in [84]. First, the *naive algorithm* corresponds to `DEF-N1-D`, the *dynamic algorithm* to `EXP-N1-D`, and the *caching algorithm* to `EXP-N1-E`.

Classifying the *incremental algorithm*, however, is not as straightforward as in case of the previous ones. Though it can be said that it corresponds to `RFN-F1-E` because it is in principle based on the refinement correction strategy with enabled signatures, there are important differences. First of all, it lacks several optimizations and improvements. To name the most significant one, the described mechanism of the first cost estimation was not applied there. And second, it was not based on the forwarding execution approach, but rather on yet another one that was controlled by our proprietary scheduler responsible for assigning prepared tasks to available worker threads. All in all, despite it followed the refinement strategy, it did not perform efficiently enough.

To conclude, though we have certain expectations that are not difficult to track according to all the observations we have discussed throughout this entire chapter, studying which particular configuration is really up to outperform all the remaining ones is the subject of the experimental evaluation.

However, before we finish, it is worth of highlighting one final, but fundamental fact – whichever particular algorithm configuration we chose, the results will always be the same, i.e. each configuration provides a solution to our data tree correction problem exactly as we have defined it in Definition 3.6.

5. Experiments

We managed to integrate the whole proposed correction model together with all the described correction strategies, execution approaches and signature handling modes into right one universal *Corrector* implementation in Java programming language [66].

Binding all the introduced correction configurations together into one piece was necessary to keep the required work in reasonable boundaries, though it is true that when a dedicated implementation would be created from scratch for a given particular configuration, slightly better design and efficiency could be most likely achieved. On the other hand, our solution does not involve any notable drawbacks and serves the desired evaluation objectives without any difficulties.

At this time our implementation works as a command line utility that accepts standard XML documents and regular tree grammars as input arguments, while the found corrections in a form of sequences of edit operations obtained from the translated and unfolded repair structures are presented to the user as an output.

In this chapter we focus on the experimental evaluation of our entire correction model and algorithms. We first describe parameters of data trees we used for the experiments, then we discuss our goals and hypotheses we would like to confirm, and most importantly we also provide description of characteristics we actually want to measure, study and compare. The main part of the following text, though, provides a wide set of tables and figures on which we illustrate our observations and conclusions we could afford to embrace.

Before we start, we should also mention that we made the whole Corrector implementation publicly available [82], including the fully integrated framework for conducting the experiments presented in this thesis. The mentioned profiling framework not only allows us to obtain more accurate measurements that are less influenced by the outer environment, but also allows us to focus on a wide range of different and very specific internal features and indicators, as well as to easily perform extensive experiments in an automated way.

5.1 Settings of Experiment

The main purpose of all the experiments we are about to present is to show capabilities of the model and algorithms we proposed. Although there are also other existing approaches like the most related one by Bouchou et al. [18, 19, 5], they pose different assumptions, consider different expressiveness of schemata, and may also produce different results. As a consequence, no direct comparison with our solution would make a sense. Moreover, even if we mutually flattened the considered assumptions, our approach would outperform simply according to features and expected time complexities at the theoretical level.

Therefore, our objective is to demonstrate practical usability of our solution, as well as to confirm or refuse behavior expectations we have from the theoretical point of view. However, the core part of our work lies in a mutual comparison of all the introduced correction strategies, execution approaches and signature handling modes. We not only want to study their features, but we would especially like to find right one correction configuration that performs the best.

In particular, though the impact of enabled signatures is apparently crucial, it is not obvious whether multi-threaded execution approaches may be more efficient, nor it is easy to conclude for the refinement strategy, whether the required overhead related to the scattered intent evaluation can still be overwhelmed by savings gained because of the more extensive pruning. These are just some of the questions we would like to answer.

5.1.1 Datasets

Data trees we used in all our experiments are based on the grammar we used throughout the entire thesis in examples. Although we have provided its definition in Example 2.8, we recall it right here once again.

Hence, let $\mathcal{G}_0 = (N,\ T,\ S,\ P)$ be a regular tree grammar such that $N = \{A,\ B,\ C,\ D_A,\ D_B\}$ is a set of nonterminal symbols, $T = \{a,\ b,\ c,\ d\}$ a set of terminal symbols, and $S = \{A,\ B\}$ are the starting nonterminal symbols. The set P contains the following production rules:

$$\begin{aligned}
&\mathcal{F}_1 = \mathcal{F}_{a,A} = [\,a,\ C.{D_A}^* \to A\,],\\
&\mathcal{F}_2 = \mathcal{F}_{b,B} = [\,b,\ {D_B}^* \to B\,],\\
&\mathcal{F}_3 = \mathcal{F}_{c,C} = [\,c,\ \epsilon \to C\,],\\
&\mathcal{F}_4 = \mathcal{F}_{d,D_A} = [\,d,\ C^* \to D_A\,] \text{ and}\\
&\mathcal{F}_5 = \mathcal{F}_{d,D_B} = [\,d,\ A|B|C \to D_B\,].
\end{aligned}$$

As we have also already shown, $\mathcal{G}_0$ is a single type tree grammar, but not a local tree grammar because of the presence of competing nonterminal symbols D_A and D_B.

Despite one might say that this grammar is a bit simple (and yes, it was our intention), it actually contains all the constructs we can encounter with – not only the involved regular expressions contain all the introduced operators (including the iteration * as the most tricky one because of the possibility of generating data trees of unlimited sizes to the width), but the grammar itself is also recursive (which once again can lead to data trees of unlimited sizes, this time to the depth). In other words, this grammar perfectly fulfills are requirements and still allows us to perform more than just meaningful experiments.

Unless otherwise stated, all the data trees were generated are valid with respect to $\mathcal{G}_0$. For this purpose we used our proprietary generator, which is also a part of the Corrector implementation. It enabled us to have the complete control over the process of data trees generation, especially to generate data trees of exactly the required sizes.

Before we move forward, let us now describe a few characteristics of data trees. They will become useful later on in order to provide at least some basic insight into the structure and nature of the data trees we used.

Definition 5.1 (Data Tree Characteristics)**.** *We define the following characteristics for a data tree $\mathcal{T} = (D,\ lab,\ val)$:*

- $fanOut^{min}(\mathcal{T}) = \min_{u \in (D \setminus LeafNodes(D))} fanOut(u)$ *as the* minimal fan-out *of internal nodes of $\mathcal{T}$; in case $D = \{\epsilon\}$ we put $fanOut^{min}(\mathcal{T}) = 0$ and when $D = \emptyset$ we put $fanOut^{min}(\mathcal{T}) = \perp$.*

- $fanOut^{max}(\mathcal{T}) = \max_{u \in (D \setminus LeafNodes(D))} fanOut(u)$ *as the* maximal fan-out *of internal nodes of* $\mathcal{T}$*; in case* $D = \{\epsilon\}$ *we put* $fanOut^{max}(\mathcal{T}) = 0$ *and when* $D = \emptyset$ *we put* $fanOut^{max}(\mathcal{T}) = \perp$*.*

- $depth^{min}(\mathcal{T}) = \min_{u \in LeafNodes(D)} depth(u)$ *as the* minimal depth *of leaf nodes of* $\mathcal{T}$*.*

- $depth^{max}(\mathcal{T}) = \max_{u \in LeafNodes(D)} depth(u)$ *as the* maximal depth *of leaf nodes of* $\mathcal{T}$*.*

Given a non-empty set of data trees $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ *for some* $n \in \mathbb{N}$*, we derive average values of the previous characteristics over* T *as follows:*

- $fanOut^{min}_{avg}(T) = (\sum_{\mathcal{T} \in T} fanOut^{min}(\mathcal{T}))$ / $|T|$ *as an average minimal fan-out of data trees in* T*,*

- $fanOut^{max}_{avg}(T) = (\sum_{\mathcal{T} \in T} fanOut^{max}(\mathcal{T}))$ / $|T|$ *as an average maximal fan-out of data trees in* T*,*

- $depth^{min}_{avg}(T) = (\sum_{\mathcal{T} \in T} depth^{min}(\mathcal{T}))$ / $|T|$ *as an average minimal depth of data trees in* T*,*

- $depth^{max}_{avg}(T) = (\sum_{\mathcal{T} \in T} depth^{max}(\mathcal{T}))$ / $|T|$ *as an average maximal depth of data trees in* T*.*

Example 5.1. *Let us now return to our sample data tree* $\mathcal{T}$ *from Example 2.2 and Figure 2.3. To recall,* $\mathcal{T} = (D, lab, val)$ *with underlying nodes* $D = \{\epsilon, 0, 0.0, 1, 1.0, 2, 2.0, 2.1\}$*, label function* $lab = \{(\epsilon, a), (0, x), (0.0, c), (1, d), (1.0, c), (2, d), (2.0, c), (2.1, a)\}$*, and value function* $val = \emptyset$*.*

For this data tree $\mathcal{T}$ *we can derive the following characteristics:* $fanOut^{min}(\mathcal{T}) = \min\{fanOut(\epsilon), fanOut(0), fanOut(1), fanOut(2)\} = \min\{3, 1, 1, 2\} = 1$ *is the minimal fan-out of* $\mathcal{T}$*, and analogously* $fanOut^{max}(\mathcal{T}) = 3$ *is the maximal fan-out of* $\mathcal{T}$*. Next,* $depth^{min}(\mathcal{T}) = \min\{depth(0.0), depth(1.0), depth(2.0), depth(2.1)\} = \min\{3, 3, 3, 3\} = 3$ *is the minimal depth of* $\mathcal{T}$*, and finally* $depth^{max}(\mathcal{T}) = 3$ *the maximal depth.*

5.1.2 Measured Characteristics

In our experiments, we focused on four main groups of characteristics that we believe are able to appropriately fulfill our evaluation expectations. First of all, we measured the number of created tasks (and hence of correction intents requested for the evaluation), since they represent a basic computation unit of the entire correction process, and so they are important with respect to the anticipated time complexity. To make the insight into the creation and evaluation of tasks complete, we decided to study numbers of created tasks of different intent types separately too.

The second group deals with characteristics of correction multigraphs; in particular, numbers of vertices and edges they comprise of, it means sizes of such multigraphs in other words. It is worth noticing that the overall number of created edges corresponds to the overall number of created correction intents, though

some of them may not be requested for the evaluation at all when the handling of signatures is enabled.

In fact, these intent evaluation requests deserve thorough attention as well. This is true especially with respect to the refinement correction strategy, for it increases the number of the required task calls on one hand because of the evaluation scattered into refinement steps, but it also decreases this number at the same time because of the applied vertical pruning.

Last but not least, execution times belong beyond doubt to standard characteristics we certainly cannot omit. In the following definition we introduce all the outlined characteristics more formally.

Definition 5.2 (Correction Characteristics)**.** *Assuming that $\mathcal{T}$ is a data tree to be corrected with respect to a regular tree grammar $\mathcal{G}$ using a particular correction configuration $\mathcal{Q} \in (\Psi_{Stg} \times \Psi_{App} \times \Psi_{Sig})$ and that $CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = CreatedTasks(\mathcal{T}, \mathcal{G})$ for the configuration $\mathcal{Q}$, we define the following correction characteristics:*

- $tasks(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = |\, CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q}) \,|$ *as the overall* number of created tasks *regardless their intent types.*
- $tasks^t(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = |\, CreatedTasks^t(\mathcal{T}, \mathcal{G}, \mathcal{Q}) \,|$ *as the* number of created tasks *of a given intent type $t \in \Omega$, where $CreatedTasks^t(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ is a set of all the created tasks of a particular intent type t, i.e. $CreatedTasks^t(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = \{\mathcal{K} \mid \mathcal{K} \in CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q})$, $\mathcal{K} = (\mathcal{R}^\circ$, phase, vars, quota, deps$)$, $\mathcal{I} = (id$, type, A, $L)$ and type $= t\}$.*
- $vertices(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = \sum_{\mathcal{K} \in CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q})} |\, V^\circ \,|$ *as the* overall number of explored vertices *among all the created tasks, where V° denotes a set of vertices of a correction multigraph $\mathcal{M}^\circ$ being constructed during the processing of a task $\mathcal{K} \in CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q})$.*
- $vertices^{avg}(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = vertices(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ / $|\, CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q}) \,|$ *as the* average number of explored vertices *per correction multigraph.*
- $edges(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = \sum_{\mathcal{K} \in CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q})} |\, E^\circ \,|$ *as the* overall number of explored edges *among all the created tasks.*
- $edges^{avg}(\mathcal{T}, \mathcal{G}, \mathcal{Q}) = edges(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ / $|\, CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q}) \,|$ *as the* average number of explored edges *per correction multigraph.*
- $calls(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ *as the* overall number of task calls*, i.e. how many times execution of the corresponding correction routine $correct(\mathcal{T}, \mathcal{G})$ (Algorithms 4.2, 4.6 and 4.8) has been requested for tasks from $CreatedTasks(\mathcal{T}, \mathcal{G}, \mathcal{Q})$.*
- $time(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ *as the overall* execution time *required for processing $\mathcal{K}_{\mathcal{I}_\bullet}$, i.e. evaluating an intent repair $\mathcal{R}_{\mathcal{I}_\bullet}$ for the starting correction intent $\mathcal{I}_\bullet$.*

Given a non-empty set of data trees $T = \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$ for some $n \in \mathbb{N}$, we derive average values of the previous characteristics over T as follows: for any characteristic $\phi \in \{tasks, tasks^{type}, vertices, vertices^{avg}, edges, edges^{avg}, calls, time\}$ we define $\phi_{avg}(T, \mathcal{G}, \mathcal{Q}) = (\sum_{\mathcal{T} \in T} \phi(\mathcal{T}, \mathcal{G}, \mathcal{Q}))$ / $|T|$ as an average value of ϕ over data trees from T.

At this point let us note that not all of the described characteristics behave as functions, i.e. their values are not always deterministic with respect to all the enumerated arguments, and so may return different values each time the correction is involved. This is most obvious in case of the execution time $time(\mathcal{T}, \mathcal{G}, \mathcal{Q})$. However and more surprisingly, though in very scarce circumstances and with very limited impact, the number of task calls $calls(\mathcal{T}, \mathcal{G}, \mathcal{Q})$ may also slightly vary in case of the multi-threaded execution approaches.

Anyway, to overcome these contingencies, and especially and more importantly to cope with the randomness hidden in the generated data trees themselves, we always performed all the proposed analyses on a larger set of data trees, and then computed average values of such characteristics to obtain more accurate results.

Furthermore, when execution times were measured, we always performed several let us say warming-up executions at first, and also thrown away a small number of the lowest as well as highest measured times to suppress the possibly impact of anomalies.

Finally, it is worth of emphasizing that the execution times do not contain the phase of input documents and grammars parsing, nor they involve the phase of repair translation to edit sequences; in other words, only the correction phase itself is measured to really underpin algorithms we deal with.

5.2 Results of Experiments

In the remaining parts of this chapter we discuss background and results of the experiments we conducted. We first try to demonstrate how important impact enabled signatures represent, and then we describe and compare pruning capabilities of all the considered correction strategies. Once we understand the basic principles, we look more deeply on the behavior of task, multigraph and execution characteristics over a sequence of larger data trees. We also look at execution times of all the promising correction configurations, and conclude with a view into the scaling possibilities of the most efficient one.

5.2.1 Signature Modes

The first area we focus on is the importance of having signature handling enabled. In particular, we look at numbers of created tasks depending on different correction strategies, always with disabled signatures on one hand, and enabled on the other. We do not need to consider different execution approaches now, since all of them would produce the same results.

For the purpose of answering the first outlined question, we created a dataset $\mathcal{D}^{FA} = \langle\, T^{FA}_{100} \,\rangle$ of 25 randomly generated data tree instances, all of them having the size equal exactly to 100 nodes. Their average maximal fan-out is equal to $fanOut^{max}_{avg}(T^{FA}_{100}) = 4$, average minimal depth $depth^{min}_{avg}(T^{FA}_{100}) = 6$, and average maximal depth $depth^{max}_{avg}(T^{FA}_{100}) = 11$.

The results are presented in Table 5.1. Although the numbers of created tasks in case of the refinement strategy seem to be very close to each other regardless the signatures, there are much notable differences in case of the remaining correction strategies. Especially in case of the default strategy with a difference of roughly

Table 5.1: Created tasks for data trees of 100 nodes

Algorithm configuration		All tasks	Created tasks depending on intent types				
			Correct	Insert	Delete	Repair	Rename
DEF	D	84 720	1	62 441	18 200	1 867	2 210
	E	421	1	21	100	147	151
		0.5%	100.0%	0.0%	0.5%	7.9%	6.8%
EXP	D	6 716	1	2 265	3 981	204	263
	E	400	1	21	100	147	130
		6.0%	100.0%	0.9%	2.5%	72.1%	49.4%
RFN	D	507	1	223	100	100	83
	E	289	1	5	100	100	83
		57.0%	100.0%	2.2%	100.0%	100.0%	100.0%

two orders of magnitude. In other words, even for data trees of such very low sizes, the impact of the enabled signatures might apparently be fundamental.

This observation becomes straightforward when looking at results of the second experiment presented in Table 5.2, this time based on a dataset $\mathcal{D}^{SA} = \langle T_{10}^{SA}, T_{20}^{SA}, \ldots, T_{100}^{SA} \rangle$ of data trees of 10 different sizes ranging from 10 nodes to 100 nodes. For each size we once again generated 25 random data tree instances. Whereas we started with $fanOut_{avg}^{max}(T_{10}^{SA}) = 3$, $depth_{avg}^{min}(T_{10}^{SA}) = 3$ and $depth_{avg}^{max}(T_{10}^{SA}) = 4$, we ended with data trees T_{100}^{SA} of the same characteristics in case of the already mentioned T_{100}^{FA}.

As we can see in Figure 5.1, $tasks_{avg}(\mathcal{T}_i^{SA}, \mathcal{G}_0, \texttt{DEF-?-D})$ grows so fast for $i \in \{10, 20, \ldots, 100\}$, that even for a bit larger data trees we would not able to obtain results in reasonable times at all. On the other hand, when signatures are enabled, the number of created tasks seems to be reasonably low and linear with respect to the size of processed data trees. Yet the best results are obtained in case of the refinement strategy, as Figure 5.2 suggests.

Table 5.2: Created tasks for data trees of 10 to 100 nodes

Nodes	DEF		EXP		RFN	
	D	E	D	E	D	E
10	1 694	61	384	58	51	34
20	5 514	101	940	95	100	61
30	11 125	141	1 581	133	151	89
40	18 358	181	2 214	172	203	119
50	26 856	221	2 931	210	254	147
60	35 441	261	3 624	247	303	175
70	42 198	301	4 279	285	354	203
80	54 540	341	5 120	322	403	230
90	76 916	381	6 001	363	458	261
100	84 720	421	6 716	400	507	289

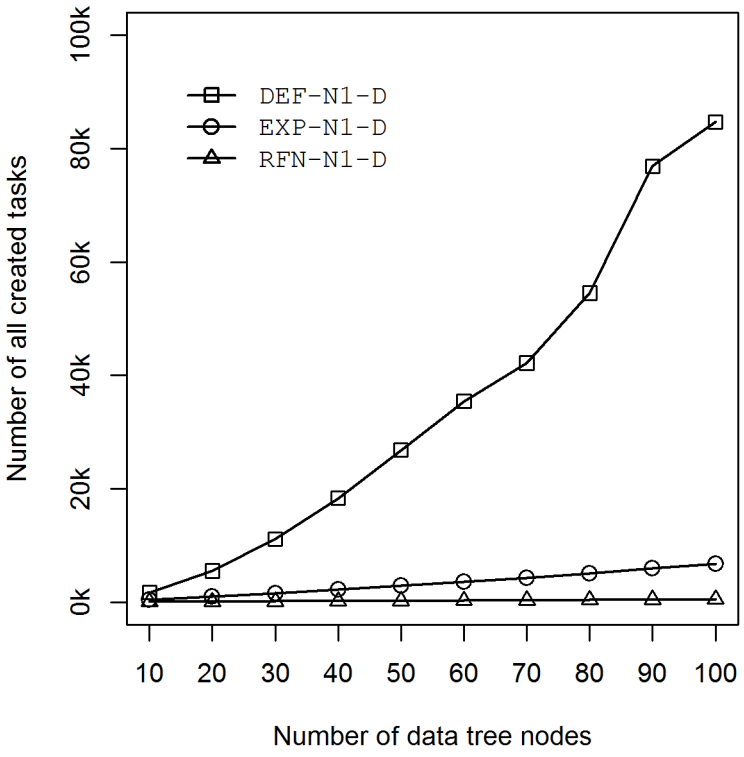


Figure 5.1: Created tasks for disabled signatures and trees of 10 to 100 nodes

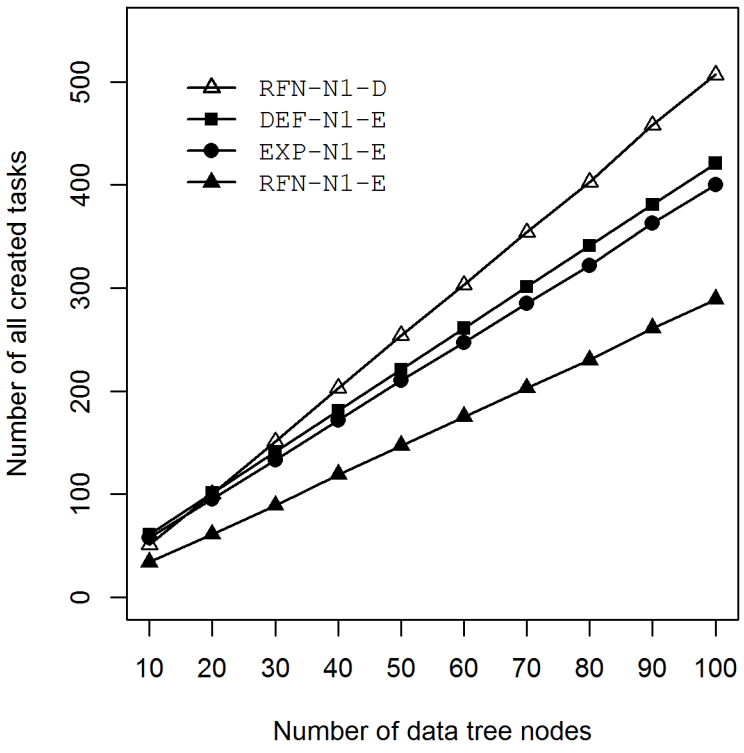


Figure 5.2: Created tasks for enabled signatures and trees of 10 to 100 nodes

Table 5.3: Created tasks for data trees of 1 000 nodes

Algorithm configuration		All tasks	Created tasks depending on intent types				
			Correct	Insert	Delete	Repair	Rename
DEF	E	4 021	1	21	1 000	1 494	1 504
EXP	E	3 737 93%	1 100%	21 100%	1 000 100%	1 494 100%	1 221 81%
RFN	E	2 779 69%	1 100%	5 24%	1 000 100%	1 000 67%	773 51%

To conclude, having signature handling enabled, i.e. caching of computed repairs and their reusing according to matching correction intent signatures is definitely a must. Because of this observation, we will no longer work with disabled signatures in the remaining tests, simply because we would not be able to compute any results except for the refinement strategy.

5.2.2 Pruning Effect of Strategies

Now we switch to another dataset $\mathcal{D}^{FB} = \langle T_{1k}^{FB} \rangle$ with 1 000 data trees instances, all with right 1 000 nodes. Their average maximal fan-out is $fanOut_{avg}^{max}(T_{1k}^{FB}) = 5$, average minimal depth $depth_{avg}^{min}(T_{1k}^{FB}) = 8$, and average maximal depth $depth_{avg}^{max}(T_{1k}^{FB}) = 15$.

Our general goal for the following two tests is to study both horizontal and vertical pruning capabilities of the correction strategies we introduced, i.e. their different ability to reduce the number of created tasks, and therefore to increase efficiency of the correction as a whole.

In Table 5.3 we have numbers of task for $\mathcal{D}^{FB}$, once again with a more fine task counts of the particular intent types. According to our expectations, the exploring strategy performs a bit better than the default one (the overall number of created tasks $tasks_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `EXP-?-E`) is reduced just to approximately 93% of the base $tasks_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `DEF-?-E`)), and the refinement strategy even better (only roughly 69% of $tasks_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `DEF-?-E`) in case of $tasks_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `RFN-?-E`)).

Without further details, these overall differences clearly have their interpretable roots in tasks of distinct intent types; and so whereas for example the number of `delete` tasks $tasks_{avg}^{\texttt{delete}}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `?-?-E`) is the same for all the strategies (and that is not a coincidence as we already know), notable pruning yields can be attained especially in case of the `repair` and `rename` types (in case we are interested in the absolute changes).

Table 5.4 works with data trees from $\mathcal{D}^{FB}$ as well, but provides different characteristics that also confirms the pruning capabilities of both the improved strategies – this time with respect to average sizes of correction multigraphs.

As we might already expect at this moment, the refinement strategy outperforms both the other strategies even from this point of view. Not only the overall number of explored vertices $vertices_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `RFN-?-E`) and edges (i.e. correction intents) $edges_{avg}(\mathcal{T}_{1k}^{FB}, \mathcal{G}_0,$ `RFN-?-E`) is reduced only to 34% and

Table 5.4: Multigraph characteristics for data trees of 1 000 nodes

Algorithm configuration		Created tasks	Explored vertices		Explored edges	
			Total	Average	Total	Average
DEF	E	4 021	15 029	3.74	22 397	5.57
EXP	E	3 737 93%	13 904 93%	3.72 99%	15 366 69%	4.11 74%
RFN	E	2 779 69%	5 141 34%	1.85 49%	4 961 22%	1.79 32%

22% respectively when having the default strategy as a baseline, but the average sizes of multigraphs are significantly reduced too. In particular, whereas correction multigraphs in case of the default strategy had on average 3.74 vertices and 5.57 edges, these numbers have been dwindled up to just $vertices^{avg}_{avg}(\mathcal{T}^{FB}_{1k}, \mathcal{G}_0,$ `RFN-?-E`$) = 1.85$ vertices and $edges^{avg}_{avg}(\mathcal{T}^{FB}_{1k}, \mathcal{G}_0,$ `RFN-?-E`$) = 1.79$ edges per a correction multigraph.

Having the previous observations in mind, we can safely conclude that the pruning impact of the refinement strategy is more than significant. This corresponds to our expectations from the theoretical point of view as well as to our original motivation. However, it is another question which configuration will actually provide the lowest execution times. In other words, one thing is the number of created tasks and its reduction, a completely other thing is the overhead related to the shattered evaluation, as we have already outlined.

5.2.3 Features of Strategies

Before we move toward the execution times, we still keep focused on value-based characteristics for a while. In order to understand how the structure of created tasks and multigraph sizes behave on larger documents. For this purpose we generated another sequence of data trees, this time of sizes starting at 1 000 nodes and ending at 10 000 nodes.

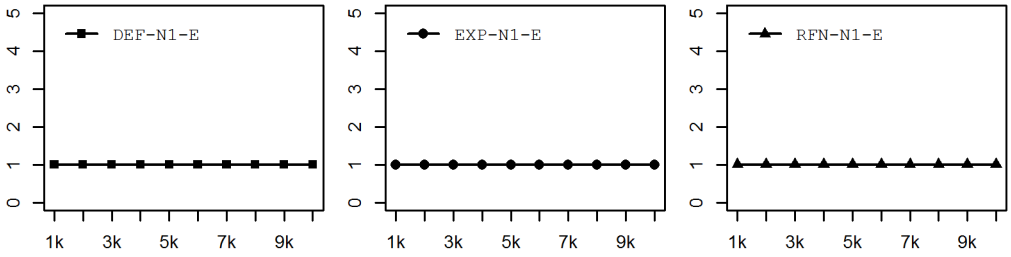
We have a dataset $\mathcal{D}^{SB} = \langle T^{SB}_{1k}, T^{SB}_{2k}, \ldots, T^{SB}_{10k} \rangle$, always having 100 data trees for each particular size. While starting with $fanOut^{max}_{avg}(T^{SB}_{1k}) = 7$, $depth^{min}_{avg}(T^{SB}_{1k}) = 7$ and $depth^{max}_{avg}(T^{SB}_{1k}) = 12$, we end with $fanOut^{max}_{avg}(T^{SB}_{10k}) = 7$, $depth^{min}_{avg}(T^{SB}_{10k}) = 9$ and $depth^{max}_{avg}(T^{SB}_{10k}) = 15$.

We first target at results presented in Table 5.5, i.e. numbers of created tasks $tasks_{avg}(\mathcal{T}^{SB}_i, \mathcal{G}_0,$ `?-?-E`$)$ for $i \in \{1k, \ldots, 10k\}$. Once again, regardless the chosen execution approach, the results would remain untouched. The first conclusion that could be made is that the overall number of created tasks is linear with respect to the data tree size. This is apparent especially when looking at Figure 5.4.

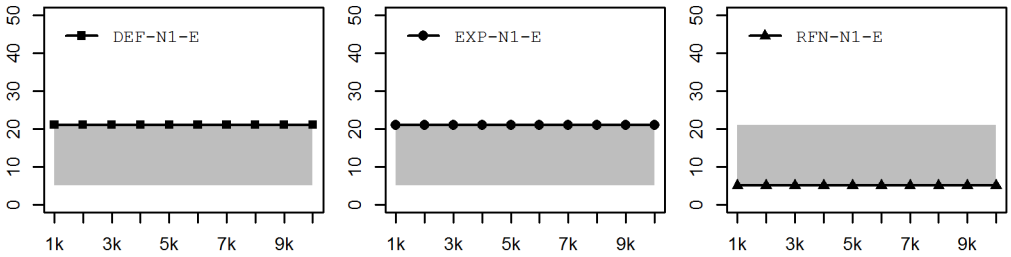
More interesting results, however, are depicted in Figure 5.3, where we consider all the intent types $t \in \Omega$ separately and study the differences between $tasks^t_{avg}(\mathcal{T}^{SB}_i, \mathcal{G}_0,$ `?-?-E`$)$ among the strategies.

Table 5.5: Created tasks for data trees of 1 000 to 10 000 nodes

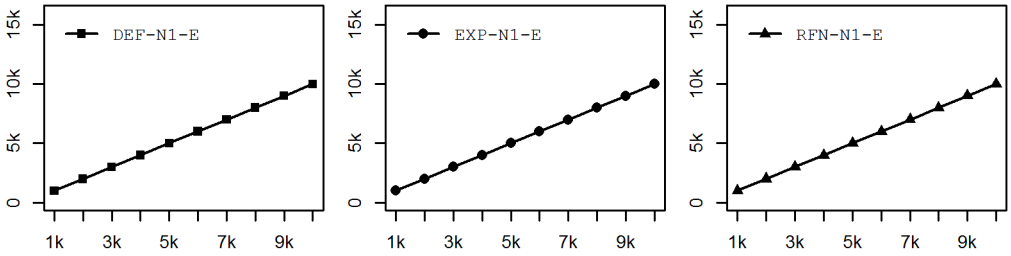
Nodes	Created tasks	Created tasks depending on intent types				
		Correct	Insert	Delete	Repair	Rename
`DEF` correction strategy:						
1 000	4 021	1	21	1 000	1 517	1 481
2 000	8 021	1	21	2 000	3 051	2 947
3 000	12 021	1	21	3 000	4 596	4 402
4 000	16 021	1	21	4 000	6 120	5 878
5 000	20 021	1	21	5 000	7 606	7 392
6 000	24 021	1	21	6 000	9 116	8 882
7 000	28 021	1	21	7 000	10 660	10 338
8 000	32 021	1	21	8 000	12 205	11 793
9 000	36 021	1	21	9 000	13 756	13 242
10 000	40 021	1	21	10 000	15 288	14 710
`EXP` correction strategy:						
1 000	3 661	1	21	1 000	1 517	1 122
2 000	7 267	1	21	2 000	3 051	2 193
3 000	10 892	1	21	3 000	4 596	3 274
4 000	14 534	1	21	4 000	6 120	4 392
5 000	18 232	1	21	5 000	7 606	5 604
6 000	21 877	1	21	6 000	9 116	6 738
7 000	25 443	1	21	7 000	10 660	7 760
8 000	29 043	1	21	8 000	12 205	8 816
9 000	32 630	1	21	9 000	13 756	9 852
10 000	36 240	1	21	10 000	15 288	10 930
`RFN` correction strategy:						
1 000	2 743	1	5	1 000	1 000	737
2 000	5 418	1	5	2 000	2 000	1 412
3 000	8 142	1	5	3 000	3 000	2 136
4 000	10 871	1	5	4 000	4 000	2 865
5 000	13 672	1	5	5 000	5 000	3 666
6 000	16 410	1	5	6 000	6 000	4 404
7 000	19 051	1	5	7 000	7 000	5 045
8 000	21 729	1	5	8 000	8 000	5 723
9 000	24 384	1	5	9 000	9 000	6 378
10 000	27 067	1	5	10 000	10 000	7 061



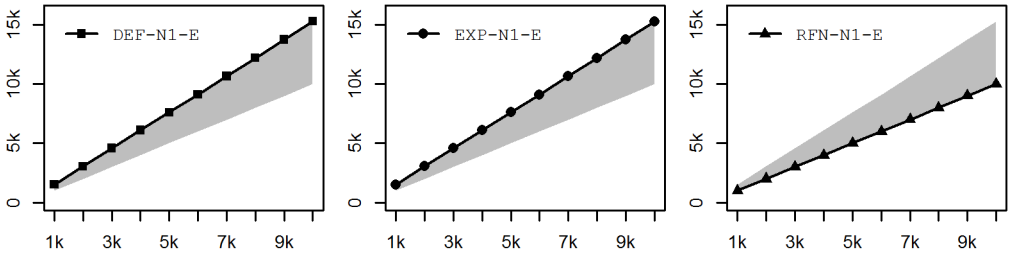
(a) Created tasks of type `correct`



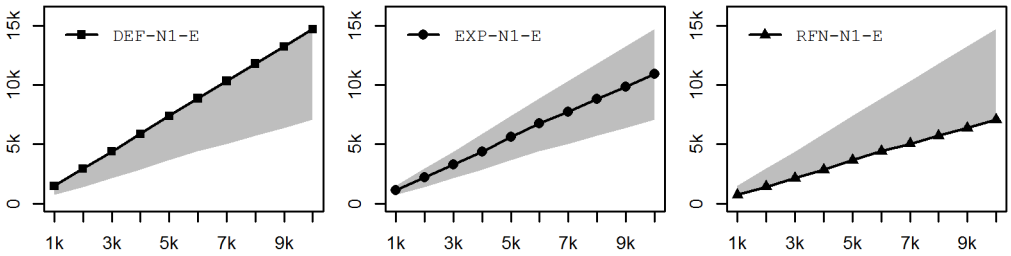
(b) Created tasks of type `insert`



(c) Created tasks of type `delete`



(d) Created tasks of type `repair`



(e) Created tasks of type `rename`

Figure 5.3: Created tasks of different types for trees of 1 000 to 10 000 nodes

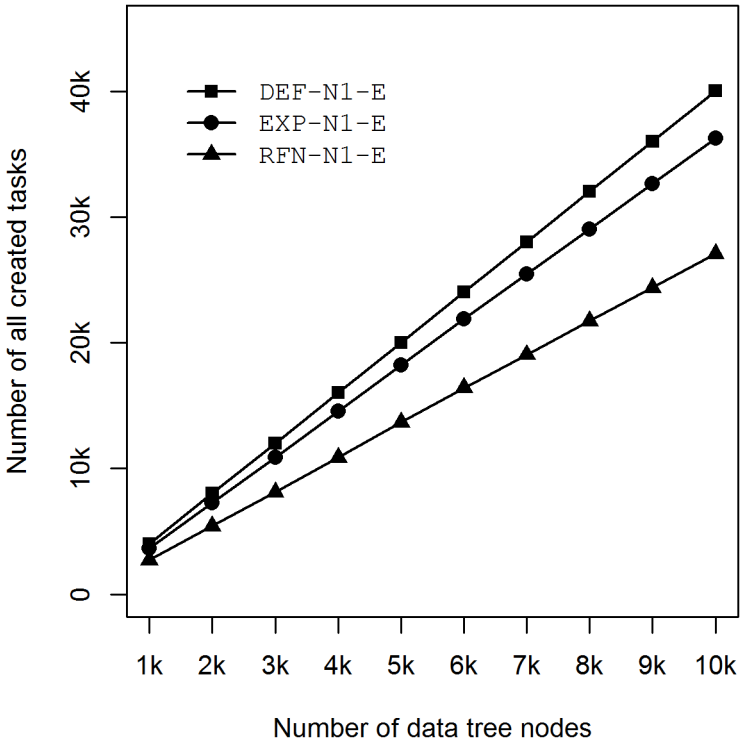


Figure 5.4: Created tasks of all types for trees of 1 000 to 10 000 nodes

Whereas the number of tasks of the type `correct` is not surprising, since it is always equal to 1 by definition, tasks of other intent types are worth of studying. First, the number of `insert` tasks is constant with respect to data tree sizes, because it only depends on the grammar characteristics. In case of the `delete` type, the number of tasks is equal to the number of nodes in data trees. Finally, `repair` and `rename` both depend on the grammar as well as data tree sizes. Just note that although the refinement strategy prunes more than the exploring one in general, the particular results may not necessarily always look exactly like the presented ones.

Furthermore, to make the mutual comparison easier to comprehend in Figure 5.3, we also enriched the individual figures (in case of `insert`, `repair` and `rename` types) with gray polygons that mark the boundaries of the worst and best results among strategies.

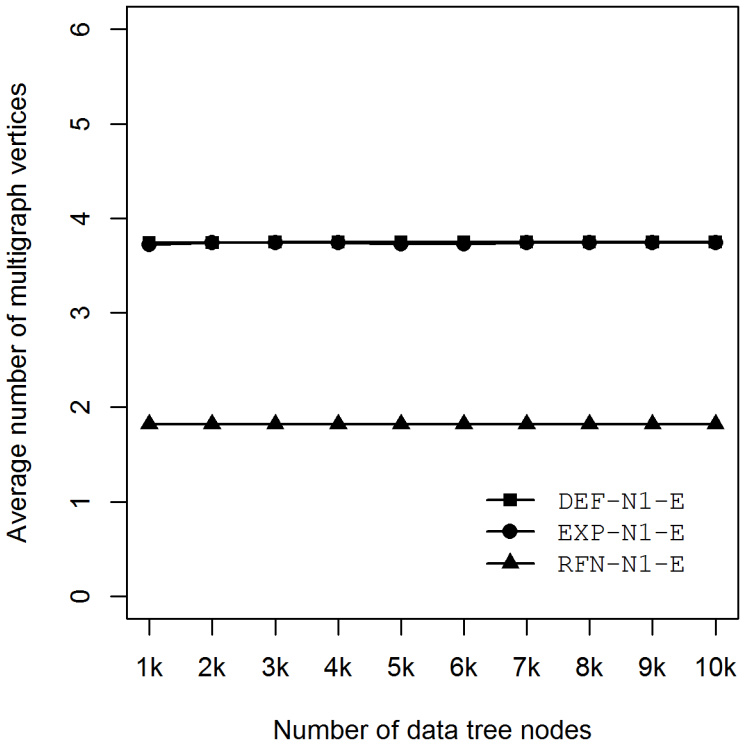
Now we shortly return to the multigraph characteristics and inspect their values on the currently assumed sequence dataset $\mathcal{D}^{SB}$. According to the data presented in Table 5.6, the overall number of explored vertices $vertices_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$ and edges $edges_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$ for $i \in \{1k, \ldots, 10k\}$ remains linear with respect to data tree sizes, which is in direct compliance with the overall number of created tasks $tasks_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$.

More interesting observation (but not from the theoretical perspective) is that average sizes of multigraphs remain nearly the same, i.e. both $vertices_{avg}^{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$ and $edges_{avg}^{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$ are independent on data tree sizes.

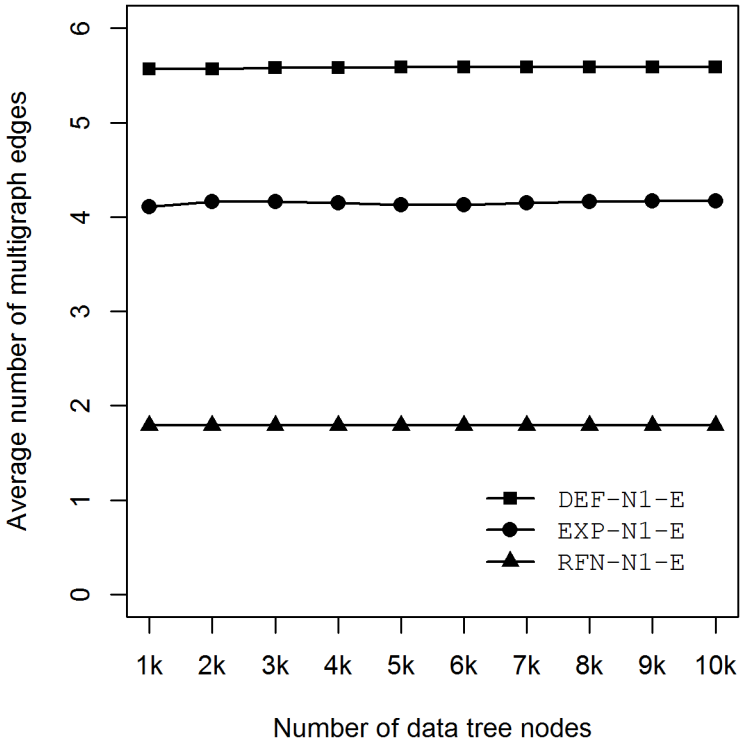
Figure 5.5 only illustrates this finding, while at the same time confirms the so far winning position of the refinement strategy.

Table 5.6: Multigraph characteristics for data trees of 1 000 to 10 000 nodes

Nodes	Created tasks	Explored vertices		Explored edges	
		Total	Average	Total	Average
`DEF` correction strategy:					
1 000	4 021	15 028	3.74	22 378	5.57
2 000	8 021	30 028	3.74	44 716	5.57
3 000	12 021	45 027	3.75	67 077	5.58
4 000	16 021	60 028	3.75	89 442	5.58
5 000	20 021	75 028	3.75	111 842	5.59
6 000	24 021	90 027	3.75	134 211	5.59
7 000	28 021	105 028	3.75	156 532	5.59
8 000	32 021	120 027	3.75	178 871	5.59
9 000	36 021	135 028	3.75	201 198	5.59
10 000	40 021	150 027	3.75	223 539	5.59
`EXP` correction strategy:					
1 000	3 661	13 619	3.72	15 051	4.11
2 000	7 267	27 171	3.74	30 219	4.16
3 000	10 892	40 723	3.74	45 318	4.16
4 000	14 534	54 317	3.74	60 362	4.15
5 000	18 232	68 035	3.73	75 379	4.13
6 000	21 877	81 648	3.73	90 456	4.13
7 000	25 443	95 098	3.74	105 589	4.15
8 000	29 043	108 621	3.74	120 757	4.16
9 000	32 630	122 130	3.74	135 940	4.17
10 000	36 240	135 687	3.74	151 109	4.17
`RFN` correction strategy:					
1 000	2 743	4 997	1.82	4 899	1.79
2 000	5 418	9 887	1.82	9 713	1.79
3 000	8 142	14 827	1.82	14 605	1.79
4 000	10 871	19 785	1.82	19 494	1.79
5 000	13 672	24 927	1.82	24 479	1.79
6 000	16 410	29 943	1.82	29 374	1.79
7 000	19 051	34 761	1.82	34 133	1.79
8 000	21 729	39 639	1.82	38 958	1.79
9 000	24 384	44 466	1.82	43 744	1.79
10 000	27 067	49 366	1.82	48 571	1.79



(a) Average number of vertices in correction multigraphs



(b) Average number of edges in correction multigraphs

Figure 5.5: Multigraph characteristics for data trees of 1 000 to 10 000 nodes

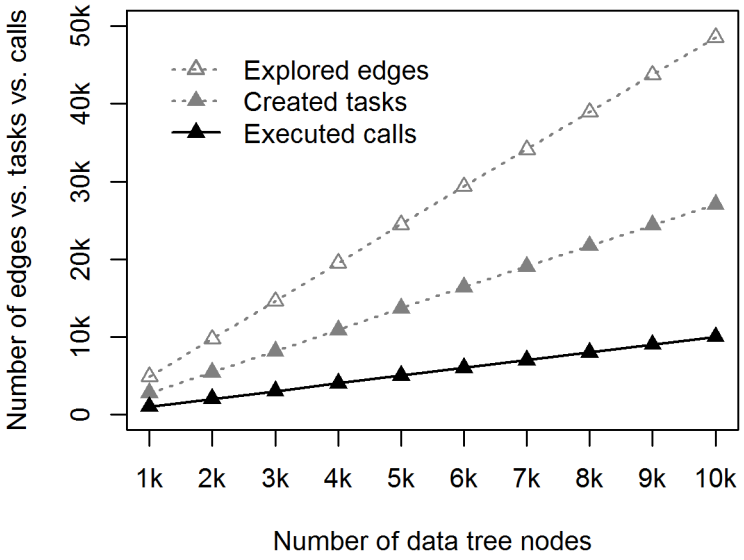


Figure 5.6: Task calls for `RFN-N1-E` and trees of 1 000 to 10 000 nodes

The last characteristic over $\mathcal{D}^{SB}$ we focus on is the number of task calls, i.e. $calls_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$. In case of the default and exploring strategy, the number of calls is identical to the overall number of created tasks, since tasks are created only when they are about to be requested, and their evaluation is always complete. On the other hand, in case of the refinement strategy with the evaluation scattered into smaller steps, the number of calls starts to differ.

As we see in Table 5.7 as well as in Figure 5.6, $calls_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{RFN-?-E})$ remains linear with respect to data tree sizes. Furthermore, the number of calls is notably lower than the number of all created tasks $tasks_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{RFN-?-E})$.

This necessarily means that there exists a significant amount of tasks that have not been requested for execution et all, i.e. the first cost estimates based on associated repairing instructions were sufficient to reject the given ways of correction as unpromising immediately after their first consideration. And since there most likely exist tasks that have been called for more than just one time, the number of such quickly rejected unpromising task will certainly not be lower than the derived minimal 37%. This finding means that we have confirmed one of our important expectations and motivations for the refinement strategy as such.

Finally, we also shortly look at another, though not yet introduced, characteristic dealing with the number of runs. It is similar to the number of calls, but it is technically related only to the forwarding execution approaches in general, since these approaches force to shatter the intent evaluation into smaller pieces at the implementation level (similarly as we discussed it in case of the refinement strategy, but from a different reason and toward a different end).

The number of runs tells how many times the given correction procedure has been invoked, even in the bottom-up direction when backtracking, i.e. returning back after having the requested task execution accomplished (whereas it was a complete or just partial evaluation). Therefore, this number is basically two times higher than the number of calls, but can be a bit lower in case of the `FN` approach because of the execution performed by multiple threads concurrently.

Table 5.7: Task calls and runs for trees of 1 000 to 10 000 nodes

Nodes	Created tasks	Task calls	Task runs depending on approaches		
			`N1`, `I1`, `IN`	`F1`	`FN`
`DEF` correction strategy:					
1 000	4 021	4 021	4 021	8 041	7 676
2 000	8 021	8 021	8 021	16 041	15 309
3 000	12 021	12 021	12 021	24 041	22 913
4 000	16 021	16 021	16 021	32 041	30 510
5 000	20 021	20 021	20 021	40 041	38 066
6 000	24 021	24 021	24 021	48 041	45 658
7 000	28 021	28 021	28 021	56 041	53 309
8 000	32 021	32 021	32 021	64 041	60 936
9 000	36 021	36 021	36 021	72 041	68 567
10 000	40 021	40 021	40 021	80 041	76 186
`EXP` correction strategy:					
1 000	3 661	3 661	3 661	7 322	6 923
2 000	7 267	7 267	7 267	14 533	13 720
3 000	10 892	10 892	10 892	21 784	20 517
4 000	14 534	14 534	14 534	29 068	27 355
5 000	18 232	18 232	18 232	36 464	34 251
6 000	21 877	21 877	21 877	43 753	41 080
7 000	25 443	25 443	25 443	50 885	47 841
8 000	29 043	29 043	29 043	58 085	54 598
9 000	32 630	32 630	32 630	65 259	61 372
10 000	36 240	36 240	36 240	72 480	68 173
`RFN` correction strategy:					
1 000	2 743	1 001	1 001	2 001	2 001
2 000	5 418	2 001	2 001	4 001	4 001
3 000	8 142	3 001	3 001	6 001	6 001
4 000	10 871	4 001	4 001	8 001	8 001
5 000	13 672	5 001	5 001	10 001	10 001
6 000	16 410	6 001	6 001	12 001	12 001
7 000	19 051	7 001	7 001	14 001	14 001
8 000	21 729	8 001	8 001	16 001	16 001
9 000	24 384	9 001	9 001	18 001	18 001
10 000	27 067	10 001	10 001	20 001	20 001

5.2.4 Execution Times

At this moment we have successfully demonstrated that the pruning impact of the refinement strategy seems to be the most promising. But we still need to take particular execution approaches into account and study the execution times to be really able to put everything together and make the final comparison.

To enable interpreting the measured execution times in a broader scope, we should describe parameters of the system we used for the experiments. In particular, it was a casual laptop Dell Latitude E5510 with Intel Core i5 M560, 2.67 GHz, 2 cores, hyper threading, 64-bit processor, 4 GB 1333 MHz DDR3 system memory with Windows 7 Professional SP1 and Java Standard Edition 8 Update 25 runtime environment.

We measured the times over $\mathcal{D}^{SB}$, i.e. a sequence $\langle\, T^{SB}_{1k}, T^{SB}_{2k}, \ldots, T^{SB}_{10k} \,\rangle$ of data trees of sizes ranging from 1 000 to 10 000 nodes as in the previous tests. However, we used only the first 20 data tree instances of each size in this experiment. We always first performed 3 warming-up executions which we did not consider, and then we performed 12 measured executions, from which we removed 1 maximal and 1 minimal. Thus we finally obtained 10 measured executions that we included into the average time characteristic $time_{avg}(\mathcal{T}_i^{SB}, \mathcal{G}_0, \texttt{?-?-E})$.

All the acquired execution times for all the correction strategies and execution approaches with enabled handling of signatures are shown in Table 5.8. In order to present the values in a more convenient and interpretable way, we decided to put the results into three separate figures, one for each of the correction strategies. So we have Figure 5.7 for the default strategy, Figure 5.8 for the exploring, and, finally, Figure 5.9 for the refinement strategy.

Each of these figures nicely illustrates the comparison of all the execution approaches within the given strategy. To make the mutual comparison among strategies possible as well, we also added a gray polygon in each of the figures to mark the boundaries of the best configuration (`RFN-N1-E`) and the worst one (`DEF-I1-E`).

As we can see from the figures, it is surprisingly difficult to make any general conclusion from the measured times. Anyway, contrary to our expectations, though multi-threaded alternatives of the invoking `IN` and forwarding `FN` execution approaches performed better than their single thread alternatives `I1` and `F1` in case of the default and exploring strategies, they did not in the refinement strategy. And all in all, they always performed worse than the most simple and straightforward nesting single approach `N1`.

In other words, the best execution approach for all the strategies seems to be the nesting single approach. The reason might be based on two main explanations. First, the required synchronization between individual threads probably became a bottleneck of the entire correction. Second, in case of the refinement strategy, the scattering of the evaluation brought only too small amounts of work that the overhead related to management of threads (though harnessing a pool of prepared worker threads) simply prevailed.

Table 5.8: Times for data trees of 1 000 to 10 000 nodes

Nodes	Times in milliseconds depending on execution approaches				
	N1	I1	IN	F1	FN
DEF correction strategy:					
1 000	41	215	65	174	69
2 000	81	430	121	321	136
3 000	125	771	192	605	203
4 000	170	933	282	633	265
5 000	234	1 195	372	990	348
6 000	329	1 604	470	1 395	456
7 000	393	1 941	591	1 790	561
8 000	651	2 330	810	2 239	814
9 000	814	2 851	1 085	2 617	1 030
10 000	977	3 200	1 270	2 821	1 256
EXP correction strategy:					
1 000	31	232	135	148	50
2 000	66	330	99	257	101
3 000	105	483	167	408	142
4 000	142	759	221	448	191
5 000	187	735	267	669	244
6 000	233	1 004	347	846	297
7 000	291	1 226	410	1 055	373
8 000	346	1 628	507	1 492	437
9 000	423	1 965	560	1 561	521
10 000	602	2 047	748	2 000	700
RFN correction strategy:					
1 000	18	98	116	70	51
2 000	27	144	169	92	129
3 000	43	240	261	153	207
4 000	59	294	285	166	242
5 000	76	342	384	211	348
6 000	91	488	493	254	340
7 000	110	538	542	322	498
8 000	132	599	694	425	561
9 000	147	657	757	473	617
10 000	164	831	881	482	577

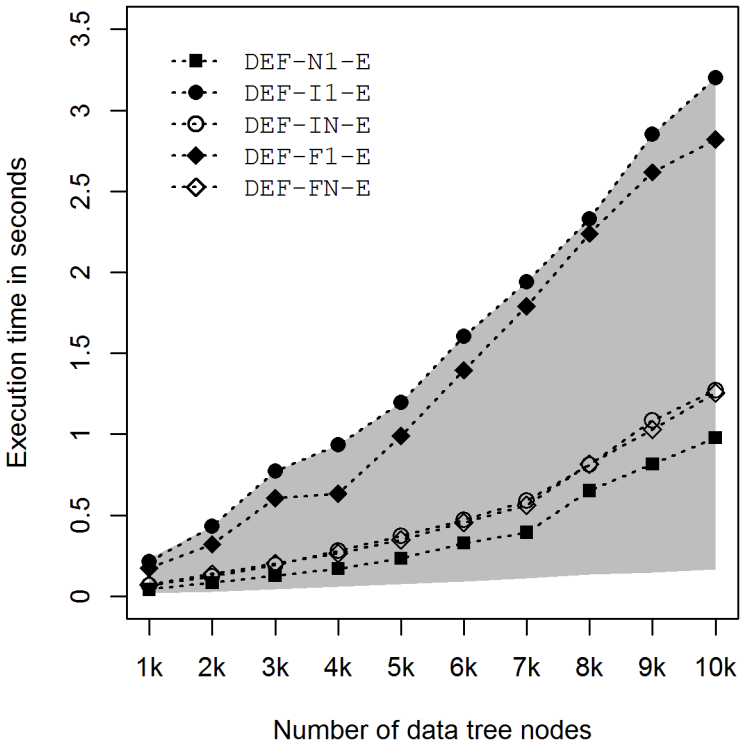


Figure 5.7: Times for the `DEF` strategy and trees of 1 000 to 10 000 nodes

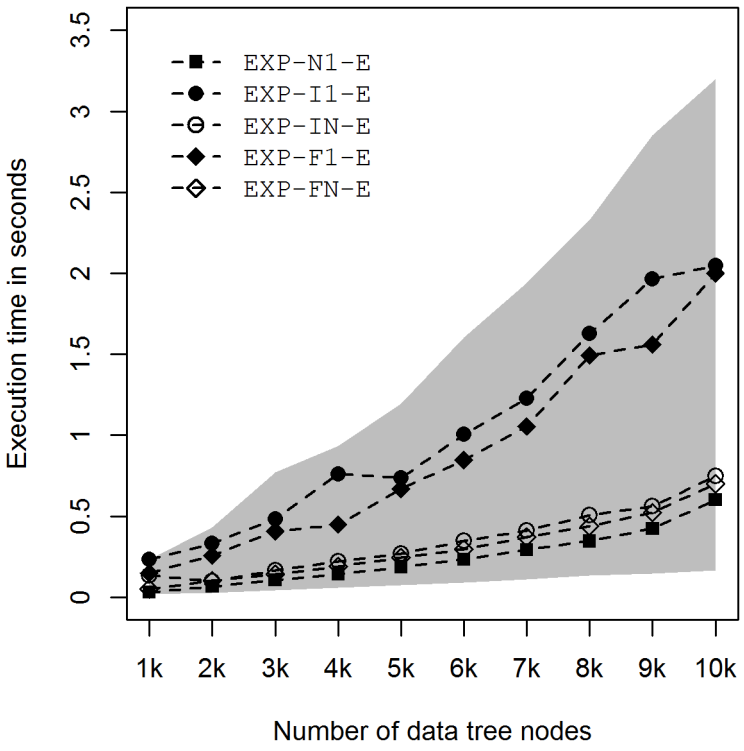


Figure 5.8: Times for the `EXP` strategy and trees of 1 000 to 10 000 nodes

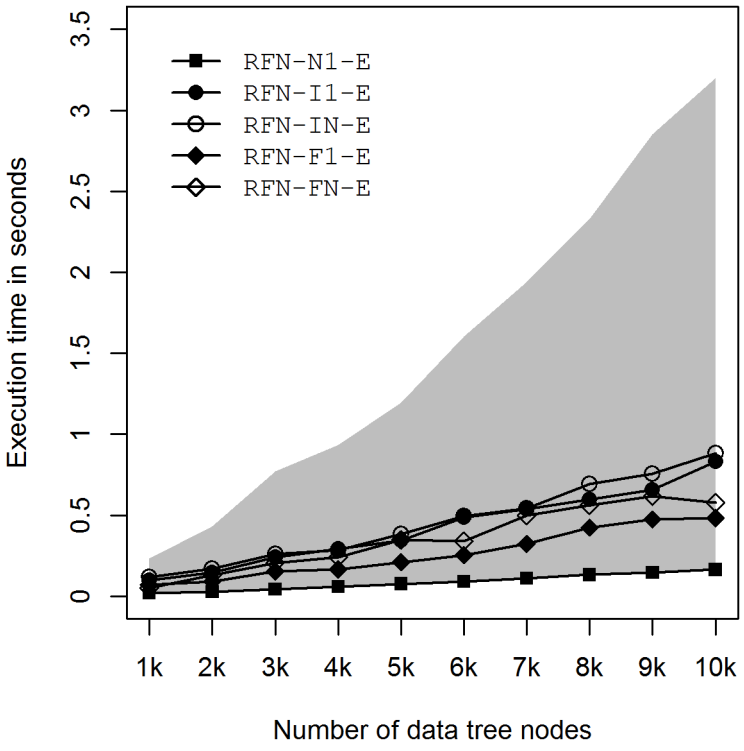


Figure 5.9: Times for the `RFN` strategy and trees of 1 000 to 10 000 nodes

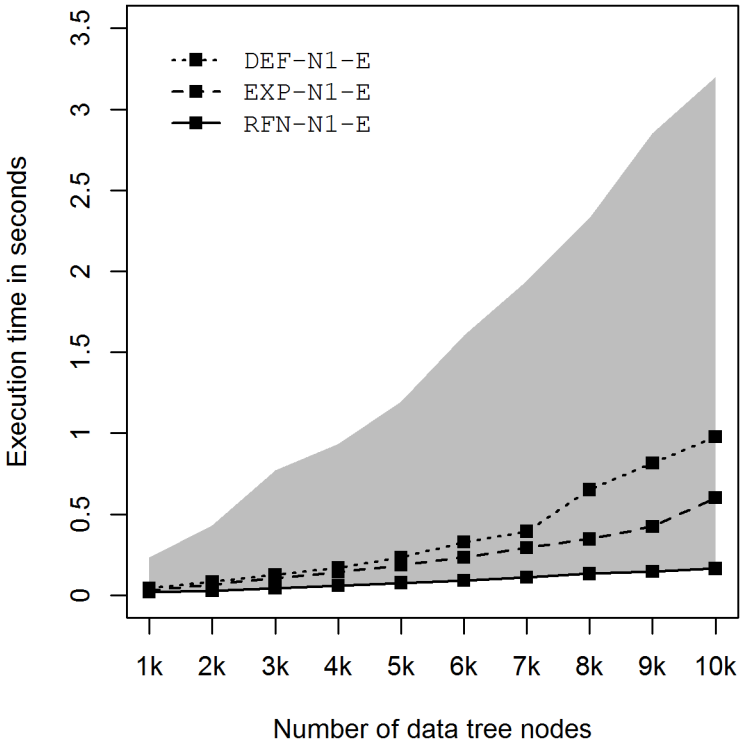


Figure 5.10: Times for the `N1` approach and trees of 1 000 to 10 000 nodes

Table 5.9: Times for the `N1` approach and data trees of 1 000 to 10 000 nodes

Nodes	Times in milliseconds depending on correction strategies				
	`DEF`	`EXP`		`RFN`	
1 000	41	31	76%	18	44%
2 000	81	66	81%	27	33%
3 000	125	105	84%	43	34%
4 000	170	142	84%	59	35%
5 000	234	187	80%	76	32%
6 000	329	233	71%	91	28%
7 000	393	291	74%	110	28%
8 000	651	346	53%	132	20%
9 000	814	423	52%	147	18%
10 000	977	602	62%	164	17%

Anyway, the winning configuration apparently is the `RFN-N1-E`, as we can see from the direct comparison presented in Table 5.9 as well as in Figure 5.10, where only execution times for the nesting single execution approaches were considered among all the strategies.

5.2.5 Effect of Invalidity Extents

Until now we have assumed that all the data trees were valid with respect to $\mathcal{G}_0$. But what effect could invalidity have on the overall correction process? We try to focus on this question right in the following paragraphs.

First, let us shortly describe how we have created such invalid data trees. We simply used our generator and randomly changed node labels (element names) to a specific and certainly not permitted name (in particular `x`) with a given parameterized probability – *invalidity extent*.

Table 5.10: Created tasks for invalid data trees of 1 000 nodes

Invalidity extent	Created tasks	Created tasks depending on intent types				
		Correct	Insert	Delete	Repair	Rename
0 %	2 733	1	5	1 000	1 000	727
5 %	2 855	1	13	1 000	975	866
10 %	2 953	1	13	1 000	953	986
15 %	3 051	1	13	1 000	931	1 106
20 %	3 145	1	13	1 000	905	1 226
25 %	3 234	1	13	1 000	873	1 346
30 %	3 317	1	13	1 000	839	1 464
35 %	3 395	1	13	1 000	799	1 582
40 %	3 472	1	13	1 000	755	1 703
45 %	3 542	1	13	1 000	707	1 820
50 %	3 609	1	13	1 000	657	1 938

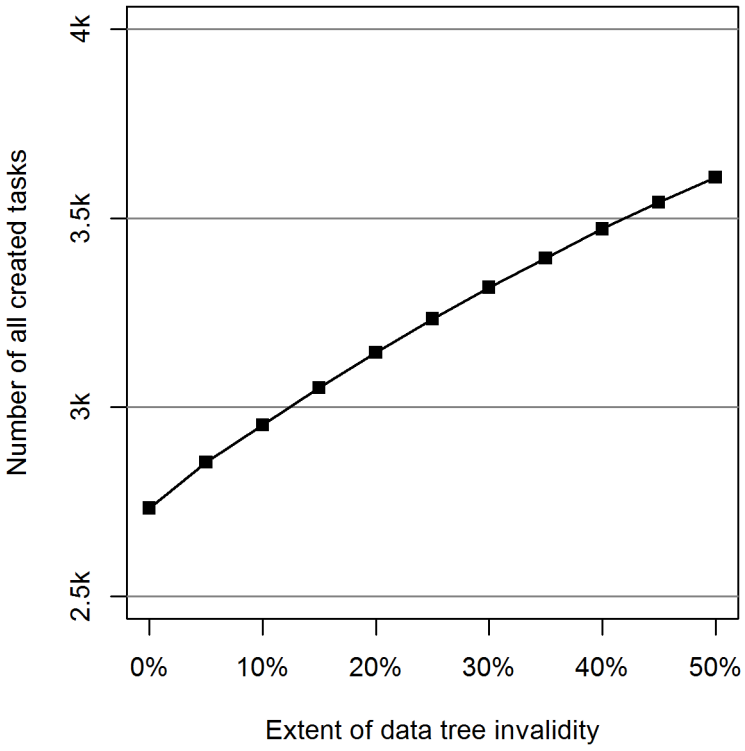


Figure 5.11: Created tasks for invalid data trees of 1 000 nodes

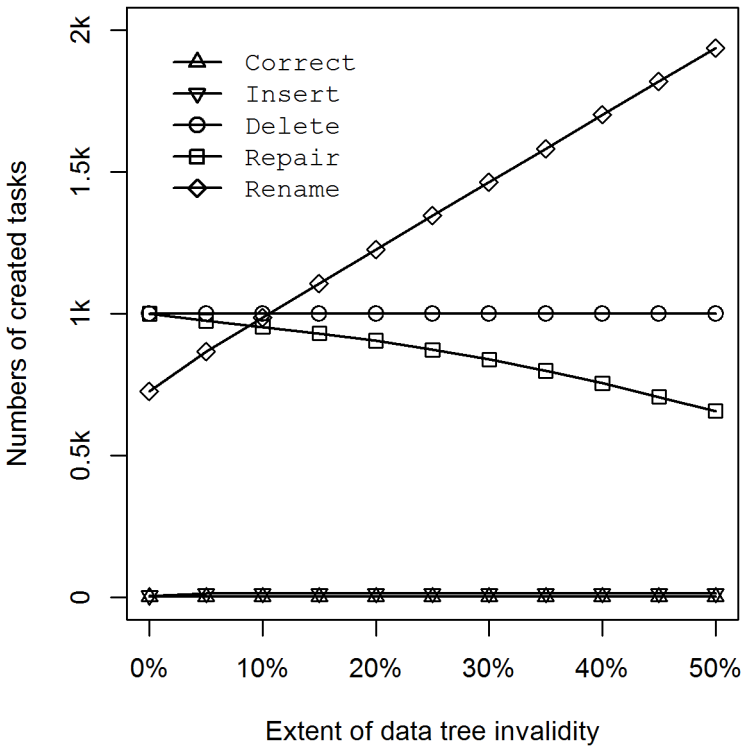


Figure 5.12: Created typed tasks for invalid data trees of 1 000 nodes

Table 5.11: Multigraph characteristics for invalid data trees of 1 000 nodes

Invalidity extent	Created tasks	Explored vertices		Explored edges	
		Total	Average	Total	Average
0 %	2 733	4 981	1.82	4 884	1.79
5 %	2 855	6 202	2.17	6 105	2.14
10 %	2 953	7 126	2.41	7 186	2.43
15 %	3 051	7 952	2.61	8 274	2.71
20 %	3 145	8 680	2.76	9 293	2.95
25 %	3 234	9 348	2.89	10 268	3.18
30 %	3 317	9 935	3.00	11 174	3.37
35 %	3 395	10 473	3.08	12 032	3.54
40 %	3 472	10 978	3.16	12 867	3.71
45 %	3 542	11 435	3.23	13 644	3.85
50 %	3 609	11 863	3.29	14 398	3.99

In all the invalidity test scenarios we worked with a dataset $\mathcal{D}^{IA} = \langle T^{IA}_{0.00}, T^{IA}_{0.05}, T^{IA}_{0.10}, \ldots, T^{IA}_{0.50} \rangle$ of data trees having exactly 1 000 nodes each, but with different invalidity extents. So, whereas data trees in $T^{IA}_{0.00}$ are perfectly valid, data trees in T^{IA}_{i} have on the other hand the introduced invalidity extent equal to $i \in \{0.00, 0.05, \ldots, 0.50\}$. Each particular set contains 500 instances in order to obtain more accurate average characteristics once again.

To be complete, the average maximal fan-out of all data trees in $\mathcal{D}^{IA}$ regardless the invalidity extent is equal to $fanOut^{max}_{avg}(T^{IA}_{i}) = 7$, average minimal depth $depth^{min}_{avg}(T^{IA}_{i}) = 7$ too, and average maximal depth $depth^{max}_{avg}(T^{IA}_{i}) = 11$.

Since we have already identified the most efficient correction configuration to be the `RFN-N1-E`, we perform the following experiments using right this configuration only, i.e. we are no longer interested in the remaining execution approaches, nor correction strategies.

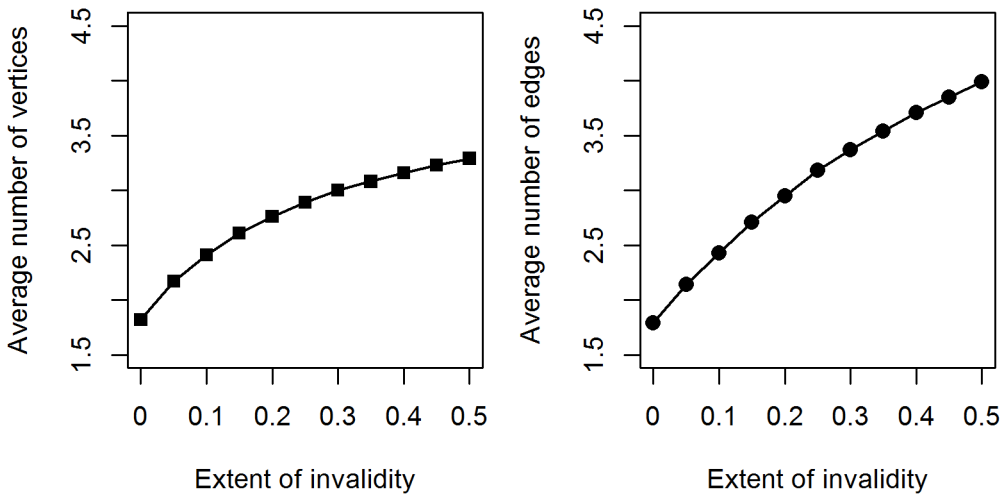


Figure 5.13: Multigraph characteristics for invalid data trees of 1 000 nodes

Numbers of the created tasks (including their structure according to individual intent types) is presented in Table 5.10. As we see, the higher the invalidity extent, the higher the overall number of tasks $tasks_{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$. However, more interesting results can be observed in case of the `repair` and `rename` intent types. Whereas the former one $tasks_{avg}^{\texttt{repair}}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ decreases (since repairing of the introduced `x` nodes is no longer the cheapest correction option), the latter one $tasks_{avg}^{\texttt{rename}}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ thus increases (and increases faster). This mutual relationship is easy to see in Figure 5.12, whereas Figure 5.11 depicts the overall number of the created tasks.

Multigraph characteristics for the dataset $\mathcal{D}^{IA}$ are then listed in Table 5.11. It is obvious that although both the average numbers of vertices $vertices_{avg}^{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ and edges $edges_{avg}^{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ for $i \in \{0.00, 0.05, \ldots, 0.50\}$ increase with the rising extent of invalidity, this rise is no worse than linear with respect to the invalidity, i.e. does not cause any difficulties that could notably affect the behavior of the whole correction as such. For clarity, both the yet discussed average multigraph size dependencies are the subject of Figure 5.13.

Table 5.12: Times and task calls for invalid data trees of 1 000 nodes

Invalidity extent	Created tasks	Explored edges	Task calls	Task runs	Execution times
0 %	2 733	4 884	1 001	1 001	13
5 %	2 855	6 105	2 548	2 548	20
10 %	2 953	7 186	4 031	4 031	26
15 %	3 051	8 274	5 275	5 275	32
20 %	3 145	9 293	6 324	6 324	37
25 %	3 234	10 268	7 261	7 261	43
30 %	3 317	11 174	7 996	7 996	47
35 %	3 395	12 032	8 609	8 609	50
40 %	3 472	12 867	9 136	9 136	54
45 %	3 542	13 644	9 573	9 573	57
50 %	3 609	14 398	9 940	9 940	60

Let us now focus on Table 5.12 with both the execution characteristics. Although the number of executed task calls $calls_{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ grows with respect to the extent of invalidity and even exceeds the number of all created tasks $tasks_{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ according to Figure 5.14, this growth is once again in very reasonable boundaries. Finally in Figure 5.15, the execution times $times_{avg}(\mathcal{T}_i^{IA}, \mathcal{G}_0, \texttt{RFN-N1-E})$ are depicted (only for the first 100 data trees from each $\mathcal{T}_i^{IA}$, 3 ignored executions, and then 10 measured ones without extremes).

To conclude the previous experiments, though the extent of invalidity generally influences the performance of the correction algorithm (in particular the considered `RFN-N1-E`), this influence is not worse than linear, and so more than acceptable. Especially when we realize that invalidity extents higher than let us say 10% imply data trees probably only too inconsistent.

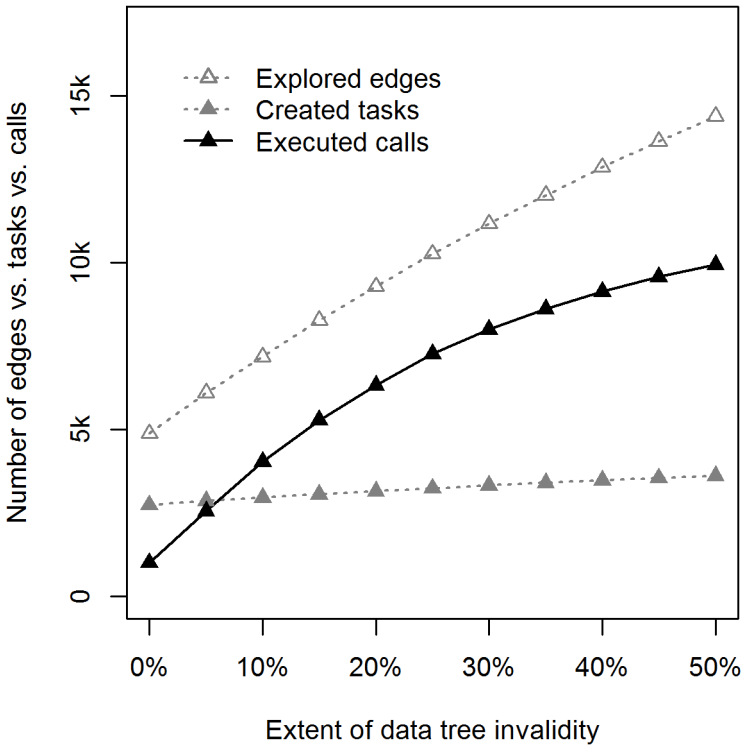


Figure 5.14: Task calls for invalid data trees of 1 000 nodes

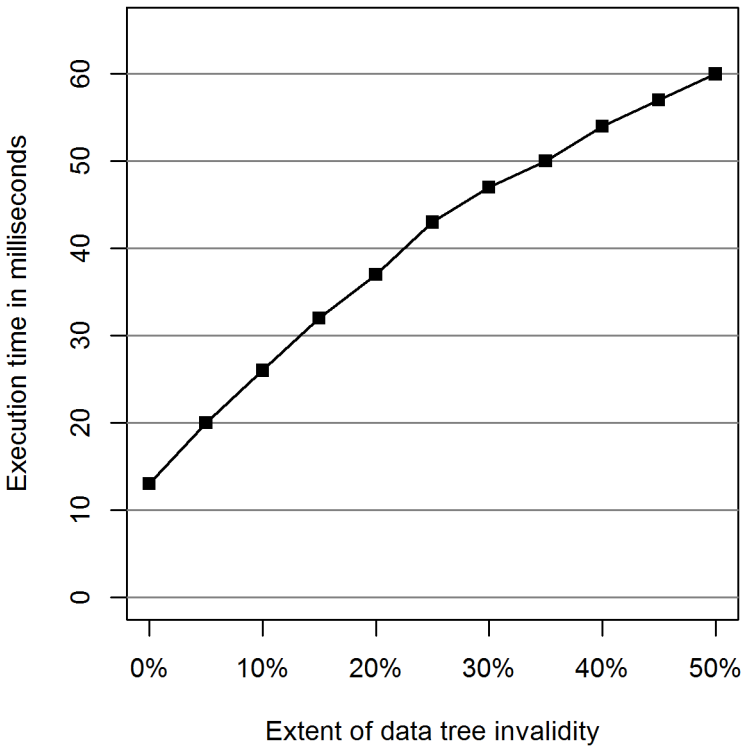


Figure 5.15: Times for invalid data trees of 1 000 nodes

5.2.6 Scaling Perspectives

Last but not least, we present results of yet another experiment – this time dealing with the efficiency possibilities of the `RFN-N1-E` configuration even for very large data trees.

For this purpose assume we have a dataset $\mathcal{D}^{SC} = \langle T^{SC}_{10k}, T^{SC}_{20k}, \ldots, T^{SC}_{100k} \rangle$ of data trees of sizes starting at 10 000 nodes and ending at 100 000 nodes. For each particular size we generated 100 data tree instances and their basic characteristics are as follows: the average maximal fan-out $fanOut^{max}_{avg}(T^{SC}_{10k}) = 13$, average minimal depth $depth^{min}_{avg}(T^{SC}_{10k}) = 8$ and average maximal depth $depth^{max}_{avg}(T^{SC}_{10k}) = 11$ for the smallest data trees, and $fanOut^{max}_{avg}(T^{SC}_{100k}) = 13$, $depth^{min}_{avg}(T^{SC}_{100k}) = 9$ and $depth^{max}_{avg}(T^{SC}_{100k}) = 14$ for the largest ones.

Since measuring of the times would be affected by computing all the remaining value-based characteristics, we first focused only on them, and in a separate test scenario we detected the execution times themselves. For them we always worked with the first 50 data tree instances only, while performing 3 ignored start-up executions and then 7 measured ones, from which 5 remained for the final results after having 1 minimal and 1 maximal extremes ignored too.

All the obtained scaling results are presented in Table 5.13. Without any further discussion required, all our previously accepted general conclusions related to the task numbers, multigraph sizes and task calls are still valid.

The execution times are then depicted in Figure 5.16. They still seem to be nearly linear, which only confirms our theoretical expectations.

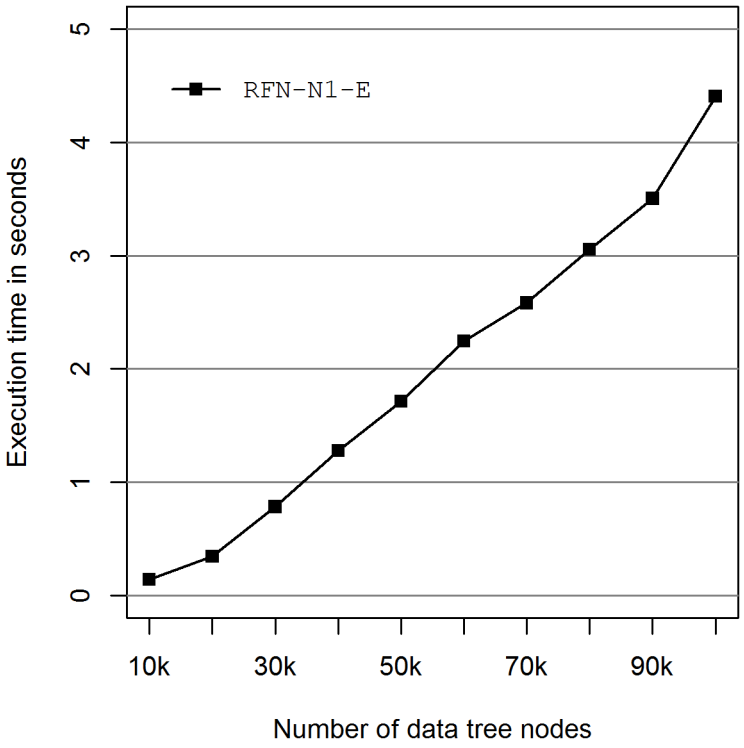


Figure 5.16: Times for `RFN-N1-E` and trees of 10 000 to 100 000 nodes

Table 5.13: Scaling characteristics for data trees of 10 000 to 100 000 nodes

Nodes	Created tasks	Graph averages		Task calls	Execution times
		Vertices	Edges		
10 000	25 243	1.79	1.82	10 001	141
20 000	51 938	1.76	1.82	20 001	345
30 000	80 406	1.78	1.80	30 001	783
40 000	109 092	1.79	1.79	40 001	1 277
50 000	134 768	1.80	1.79	50 001	1 713
60 000	158 717	1.79	1.80	60 001	2 244
70 000	182 651	1.79	1.81	70 001	2 583
80 000	206 481	1.79	1.81	80 001	3 057
90 000	230 673	1.79	1.82	90 001	3 505
100 000	255 263	1.79	1.82	100 001	4 409

To summarize all the performed experiments presented in this work, we first concluded that having signatures enabled is no doubt a fundamental requirement to obtain a correction algorithm that would be usable in practice.

Then we also compared the pruning capabilities of all the correction strategies and confirmed our initial expectation that the refinement strategy might be the most efficient one.

However, it was no sooner until we performed the execution times tests, so we could make our final conclusion that the winning configuration that outperforms all the remaining ones, deals more than appropriately with different extents of invalidity, and also scales well at the same time, is really built on top of the refinement strategy – in particular the `RFN-N1-E` configuration utilizing the nesting single execution approach.

6. Conclusion

XML documents and related technologies represent one of the most widespread ways of interchanging and managing data on the contemporary Web. However, its size, distributed, and dynamic nature, as well as mere aspects of human behavior often cause that these XML documents contain various inconsistency issues.

Though they span from the low level well-formedness to complex integrity constraints, we focused on the problem of the validity. In particular, structural invalidity with respect to schemata expressed by constructs of DTD and XSD (XML Schema) languages. In this thesis we presented results of our research dealing with the problem of the correction of the invalid XML documents, directly covering our former publications [83, 76, 86, 75, 85, 84].

Having a potentially invalid XML document modeled as a data tree, and a schema modeled as a regular tree grammar, our goal is to find all its corrections, i.e. valid data trees that are as close as possible to the original data tree. For this purpose we introduced a set of edit operations that allow us to add new leaf nodes, remove existing ones, or change their labels. Composing these edit operations into sequences, we are able to insert entire new subtrees, delete existing ones, or repair them. Thus we achieve a mechanism that enables us to transform invalid data trees into valid ones.

Given a data tree $\mathcal{T}$ and a regular tree grammar $\mathcal{G}$, we formally want to find a set $\operatorname{argmin}_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}, \mathcal{T}_2)$, i.e. the set of all the valid data trees that have the minimal possible distance from $\mathcal{T}$.

The provided data tree $\mathcal{T}$ is processed from its root node ϵ toward leaves. Beginning with the starting correction intent $\mathcal{I}_\bullet$ responsible for the correction of this root node ϵ with respect to a set S of starting nonterminal symbols of grammar $\mathcal{G}$, we then recursively invoke other nested correction intents responsible for dealing with simpler subproblems.

The evaluation goal of each particular correction intent $\mathcal{I}$ is to correct a sequence u of original siblings nodes with respect to a particular grammar context $\mathcal{C}$ that describes the allowed content model via a regular expression r. All the horizontal actions we can consider for this purpose are directly motivated by the possibilities of the introduced edit operations, and transition function δ of the corresponding finite automaton $\mathcal{A}_r$ capable of recognizing all the words belonging to the language $L(r)$.

However, we do not dynamically traverse the state space of such automaton and generate the allowed sequences one by one, step by step. Instead, we represent them statically within a correction multigraph $\mathcal{M}_\mathcal{I}$. This structure then immediately allows us to transform the problem of node sequence u correction to the problem of finding the shortest correction paths $P^{min}_{v_S, V_T}$. Once they are found, they are encapsulated into a sequence repair $\mathcal{N}_\mathcal{I}$ first, and then to an intent repair $\mathcal{R}_\mathcal{I}$ in the end.

Having backtracked to the very beginning, i.e. having found the intent repair $\mathcal{R}_{\mathcal{I}_\bullet}$ for the starting correction intent $\mathcal{I}_\bullet$, we have successfully corrected the entire data tree $\mathcal{T}$. If required at all, the intent repair $\mathcal{R}_{\mathcal{I}_\bullet}$ can be unfolded to obtain a set of edit operations $fix(\mathcal{R}_{\mathcal{I}_\bullet})$ that can be applied to the original data tree $\mathcal{T}$ to explicitly acquire all the corrected data trees from $\operatorname{argmin}_{\mathcal{T}_2 \in L(\mathcal{G})} dist(\mathcal{T}, \mathcal{T}_2)$.

Despite the correction model is built on top of a plain idea of recursive processing, there are several difficulties that needed to be appropriately taken into account. Starting with competing nonterminal symbols in grammars, and so the possibility of existence of more interpretation trees when resolving the validity, we also had to be aware of recursive production rules. And even when grammars as such are consistent, which means that they do not lead to empty data tree languages, they still may contain inconsistent production rules in general, as well as reachable inconsistent production rules in particular.

Also note that regular expressions in production rules themselves may also represent empty regular languages. However, the main difficulty is that these regular expressions are over the alphabet of nonterminal symbols, whereas labels of data tree nodes are based on terminal symbols. Another issue is related to regular expressions that are not 1-unambiguous. In this case we may not be able to directly construct a corresponding finite automaton that would be deterministic, though each nondeterministic one can always be transformed into a deterministic one. Anyway, even when we have a deterministic automaton where the transition function always offers at most one possibility for each particular input symbol, there can be cycles, loops, as well as no reachable accepting states, or more of them on the contrary.

However, the key component of the whole correction model is the problem of searching for the shortest correction paths themselves. In order to find them in a given correction multigraph, we have to be provided with costs of all the relevant edges, and so the overall correction costs of all the involved nested correction intents. In other words, in order to evaluate a particular correction intent, we need to assume that the subproblems have already been evaluated.

The fundamental idea of the efficient evaluation of correction intents lies in the concept of intent signatures. They allow us to avoid repeated evaluations when it is obvious from the theoretical point of view that the resulting intent repair structures simply have to be identical.

Besides, we also introduced three correction strategies – different ways how the shortest correction paths are in particular being found. Whereas the default strategy always constructs the entire correction multigraphs and only then starts searching for the paths themselves, the exploring strategy discovers only those parts of the multigraphs that really need to be considered to ensure that all the required paths are still found.

The most extended strategy is the refinement one. On the contrary to both the previous strategies, it is able to rely only on partially evaluated nested correction intents, and so only on estimates of their overall correction costs. As a consequence, it not only incorporates the horizontal pruning idea of the exploring strategy, but it is also able to prune unpromising correction intents even to the depth of their recursive nesting.

Finally, we also considered a set of execution approaches, i.e. different ways how the model and strategies can be implemented in practice using the available programming and system constructs. Whereas the nesting approach uses direct recursive calls of the correction procedures, the nesting and forwarding approaches utilize worker threads to which correction tasks are being assigned in order to be evaluated.

Having introduced two signatures modes (`D`, `E`), three correction strategies (`DEF`, `EXP`, `RFN`), and five execution approaches (`N1`, `I1`, `IN`, `F1`, `FN`), we obtained a whole set of particular algorithm configurations, i.e. particular correction algorithm implementations. Despite not all of them make sense even from the theoretical point of view, several of them proved to perform well in practice, as we have shown in our thorough experimental evaluation.

Focusing on a wide range of general characteristics dealing with numbers of created tasks or features of correction multigraphs, as well as execution times, we first confirmed the intrinsic importance of the enabled handling of signatures and caching of the already evaluated intent repairs. Then we mutually compared all the individual configurations, and concluded that the most efficient one is `RFN-N1-E`, i.e. the refinement strategy with the nesting single execution approach with the enabled signatures.

Although we originally started with XML documents and their DTD and XSD schemata, we eventually managed to extend our model to deal with a more general issue – the problem of the structural correction of trees with respect to regular tree grammars. From the abstract point of view, the main complexity of our entire solution is related to the efficient search for the shortest paths in recursively nested multigraphs.

Beside the practical motivation in the correction of XML documents themselves, our model and algorithms are directly linked to the similarity among trees, and trees with respect to tree languages. Therefore it is connected to a wide range of applications like integration and interchange of XML data, searching and composition of web services, querying over inconsistent data, classification or ranking of XML documents, as well as document and schema evolution.

Last but not least, we summarize all the important features and contributions of the correction model and algorithms we proposed.

- First of all, our correction model is the only one that supports the full expressive power of the regular tree grammars, where competing nonterminal symbols may occur without limitations. On the contrary, the approach by Suzuki [81] assumes single-type tree grammars, and the approaches by Boobna and de Rougemont [16] and Bouchou et al. [18] even more restricted local tree grammars.

- Moreover, our correction model can also handle even grammars that are inconsistent or contain useless production rules, as well as it can work with all the regular expressions, and not just with those that are 1-unambiguous (as required by both DTD and XML Schema languages).

- Regardless the particular algorithm configuration we choose, we are always able to find all the minimal corrections. This means that we are able to find them regardless the extent of invalidity of the original data trees to be corrected. While [16] also aims at finding the minimal corrections, it can only be successful when the data trees are invalid only a bit. Next, [18] aims at finding all the corrections within a given similarity threshold, but if this threshold is set too low, no correction is found at all. Finally, [81] searches for the k-closest corrections.

- We do not require to be provided with any parameter in order to commence and manage the correction. On the other hand, [18] requires the already mentioned similarity threshold parameter in order to deal with potentially infinite loops caused by iterations in regular expressions or by recursive production rules. Unfortunately, it is not easy to appropriately set a value of this threshold because it is not related to the extent of invalidity. If too small, no corrections are found; if unnecessarily high, only too high and unacceptable efficiency impact can be observed.

- Although [18] as well as [81] generate sequences of edit operations through which the corrected data trees can directly be obtained, our correction model does not need to generate them explicitly in order to find the corrections themselves. In other words, we encode all the found corrections within compact and recursively nested structures of intent repairs. Only when the user is really interested in enumerating the sequences, we obtain them by unfolding and translating the intent repairs. This solution represents another important efficiency improvement in case of data trees with more and mutually independent validity issues with more potential local corrections.

- We implemented all the proposed algorithm configurations and made this implementation including all the source files publicly available [82]. Similarly, both the executable application and source codes are available for [18]. However, this is not the case of [16], nor [81].

- When the handling of signatures is enabled, the worst-case time complexity of all our correction algorithms is polynomial with respect to the size of data trees measured in a number of nodes. And when the maximal observed fan-out is sharply lower than the overall number of nodes, the most efficient configuration `RFN-N1-E` tends to be even nearly linear in practice. Whereas [16] was evaluated on data trees of 800 nodes, [18] assumed 450 nodes. And while the latter approach led to execution times in the order of minutes, we only require a few seconds for the correction of data trees of even 100 000 nodes. Notwithstanding the different assumptions and capabilities, or algorithm therefore performs several orders of magnitude better.

To conclude, though there are still several minor open questions that could be tackled, we claim that our correction model and algorithms not only represent a notable contribution with respect to the other existing approaches, but that the refinement correction strategy represent a well-scalable approach that can be directly and successfully used in practice.

Bibliography

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. (Vienna, Austria). VLDB '07. VLDB Endowment, 2007, pp. 411–422. ISBN: 978-1-59593-649-3.

[2] Maria Adriana Abrao, Beatrice Bouchou, Mirian Halfeld Ferrari, Dominique Laurent, and Martin A. Musicante. Incremental Constraint Checking for XML Documents. In: *Database and XML Technologies*. (Toronto, Canada). Vol. 3186. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 358–379. ISBN: 978-3-540-22969-8. DOI: 10.1007/b100256.

[3] Cyril Allauzen and Mehryar Mohri. A Unified Construction of the Glushkov, Follow, and Antimirov Automata. In: *Mathematical Foundations of Computer Science 2006*. (Stara Lesna, Slovakia). Vol. 4162. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 110–121. ISBN: 978-3-540-37791-7. DOI: 10.1007/11821069_10.

[4] Rajeev Alur and P. Madhusudan. Visibly Pushdown Languages. In: *Proceedings of the 36th Annual ACM Symposium on Theory of Computing*. (Chicago, IL, USA). STOC '04. New York, NY, USA: ACM, 2004, pp. 202–211. ISBN: 1-58113-852-0. DOI: 10.1145/1007352.1007390.

[5] Joshua Amavi, Beatrice Bouchou, and Agata Savary. On Correcting XML Documents with Respect to a Schema. In: *The Computer Journal* 57.5 (2014), pp. 639–674. DOI: 10.1093/comjnl/bxt006.

[6] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit" loaded: A Scalable Lightweight Join Query Processor for RDF Data. In: *Proceedings of the 19th International Conference on World Wide Web*. (Raleigh, NC, USA). WWW '10. New York, NY, USA: ACM, 2010, pp. 41–50. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772696.

[7] Denilson Barbosa, Alberto O. Mendelzon, Leonid Libkin, Laurent Mignet, and Marcelo Arenas. Efficient Incremental Validation of XML Documents. In: *Proceedings of the 20th International Conference on Data Engineering*. (Boston, MA, USA). ICDE '04. IEEE, 2004, pp. 671–682. DOI: 10.1109/ICDE.2004.1320036.

[8] David Beckett, Gavin Carothers, and Andy Seaborne. *RDF 1.1 N-Triples: A Line-based Syntax for an RDF Graph*. 2014. URL: http://www.w3.org/TR/n-triples/.

[9] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. *RDF 1.1 Turtle: Terse RDF Triple Language*. 2014. URL: http://www.w3.org/TR/turtle/.

[10] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Simeon Jerome. *XML Path Language (XPath) 2.0 (Second Edition)*. 2010. URL: http://www.w3.org/TR/xpath20/.

[11] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. In: *RFC* 3986 (2005). URL: http://www.rfc-editor.org/info/rfc3986.

[12] Jean Berstel and Luc Boasson. Balanced Grammars and Their Languages. In: *Formal and Natural Computing*. Vol. 2300. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2002, pp. 3–25. ISBN: 978-3-540-43190-9. DOI: 10.1007/3-540-45711-9_1.

[13] Geert Jan Bex, Frank Neven, and Jan Van den Bussche. DTDs Versus XML Schema: A Practical Study. In: *Proceedings of the 7th International Workshop on the Web and Databases: Colocated with ACM SIGMOD/PODS 2004*. (Paris, France). WebDB '04. New York, NY, USA: ACM, 2004, pp. 79–84. DOI: 10.1145/1017074.1017095.

[14] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data – The Story so far. In: *International Journal on Semantic Web and Information Systems* 5.3 (2009), pp. 1–22.

[15] Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. *XQuery 1.0: An XML Query Language (Second Edition)*. 2010. URL: http://www.w3.org/TR/xquery/.

[16] Utsav Boobna and Michel de Rougemont. Correctors for XML Data. In: *Database and XML Technologies*. (Toronto, Canada). Vol. 3186. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 97–111. ISBN: 978-3-540-22969-8. DOI: 10.1007/b100256.

[17] Beatrice Bouchou and Mirian Halfeld Ferrari Alves. Updates and Incremental Validation of XML Documents. In: *Database Programming Languages*. (Potsdam, Germany). Vol. 2921. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, 2004, pp. 139–140. ISBN: 978-3-540-20896-9. DOI: 10.1007/b95340.

[18] Beatrice Bouchou, Ahmed Cheriat, Mirian Halfeld Ferrari, and Agata Savary. Integrating Correction into Incremental Validation. In: *BDA*. (Lille, France). 2006.

[19] Beatrice Bouchou, Ahmed Cheriat, Myrian Halfeld Ferrari, and Agata Savary. XML Document Correction: Incremental Approach Activated by Schema Validation. In: *The 10th International Database Engineering and Applications Symposium*. (Delhi, India). IDEAS '06. Los Alamitos, CA, USA: IEEE, 2006, pp. 228–238. ISBN: 0-7695-2577-6. DOI: 10.1109/IDEAS.2006.54.

[20] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Francois Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. 2006. URL: http://www.w3.org/TR/xml11/.

[21] Dan Brickley and R. V. Guha. *RDF Vocabulary Description Language 1.0: RDF Schema*. 2004. URL: http://www.w3.org/TR/rdf-schema/.

[22] David Carlisle, Patrick Ion, and Robert Miner. *Mathematical Markup Language (MathML) Version 3.0 (2nd Edition)*. 2014. URL: http://www.w3.org/TR/MathML3/.

[23] Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. *XML Query Use Cases*. 2007. URL: http://www.w3.org/TR/xquery-use-cases/.

[24] Ahmed Cheriat, Agata Savary, Beatrice Bouchou, and Mirian Halfeld Ferrari Alves. Incremental String Correction: Towards Correction of XML Documents. In: *Stringology: Proceedings of the Prague Stringology Conference*. (Prague, Czech Republic). Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2005, pp. 201–215. ISBN: 80-01-03307-4.

[25] Cristiana Chitic and Daniela Rosu. On Validation of XML Streams Using Finite State Machines. In: *Proceedings of the 7th International Workshop on the Web and Databases*. (Paris, France). WebDB '04. New York, NY, USA: ACM, 2004, pp. 85–90. DOI: 10.1145/1017074.1017096.

[26] James Clark. *XSL Transformations (XSLT) Version 1.0*. 1999. URL: http://www.w3.org/TR/xslt/.

[27] James Clark and Murata Makoto. *RELAX NG Specification*. 2001. URL: http://www.relaxng.org/spec-20011203.html.

[28] Gregory Cobena, Serge Abiteboul, and Amelie Marian. Detecting Changes in XML Documents. In: *Proceedings of the 18th International Conference on Data Engineering*. (San Jose, CA, USA). ICDE '02. Washington, DC, USA: IEEE, 2002, pp. 41–52. ISBN: 0-7695-1531-2. DOI: 10.1109/ICDE.2002.994696.

[29] Edward G. Coffman, M. Elphick, and Arie Shoshani. System Deadlocks. In: *ACM Comput. Surv.* 3.2 (1971), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588.

[30] Richard Cyganiak, David Wood, and Markus Lanthaler. *RDF 1.1 Concepts and Abstract Syntax*. 2014. URL: http://www.w3.org/TR/rdf11-concepts/.

[31] Erik Dahlstrom, Patrick Dengler, Anthony Grasso, Chris Lilley, Cameron McCormack, Doug Schepers, and Jonathan Watt. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 2011. URL: http://www.w3.org/TR/SVG11/.

[32] Steve DeRose, Eve Maler, David Orchard, and Norman Walsh. *XML Linking Language (XLink) Version 1.1*. 2010. URL: http://www.w3.org/TR/xlink11/.

[33] Li Ding and Tim Finin. Characterizing the Semantic Web on the Web. In: *The Semantic Web – ISWC 2006*. (Athens, GA, USA). Vol. 4273. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 242–257. ISBN: 978-3-540-49029-6. DOI: 10.1007/11926078_18.

[34] Denise Draper, Michael Dyck, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jerome Simeon, and Philip Wadler. *XQuery 1.0 and XPath 2.0 Formal Semantics (Second Edition)*. 2010. URL: http://www.w3.org/TR/xquery-semantics/.

[35] David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer (Second Edition)*. 2004. URL: http://www.w3.org/TR/xmlschema-0/.

[36] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. In: *RFC* 2616 (1999). URL: http://www.rfc-editor.org/info/rfc2616.

[37] Sergio Flesca, Filippo Furfaro, Sergio Greco, and Ester Zumpano. Querying and Repairing Inconsistent XML Data. In: *Proceedings of the 6th Interntaional Conference on Web Information Systems Engineering, WISE '05*. (New York, NY, USA). Vol. 3806. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 175–188. ISBN: 978-3-540-30017-5. DOI: 10.1007/11581062_14.

[38] Sergio Flesca, Filippo Furfaro, Sergio Greco, and Ester Zumpano. Repairs and Consistent Answers for XML Data with Functional Dependencies. In: *Database and XML Technologies*. (Berlin, Germany). Vol. 2824. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 238–253. ISBN: 978-3-540-20055-0. DOI: 10.1007/b13480.

[39] Fabien Gandon and Guus Schreiber. *RDF 1.1 XML Syntax*. 2014. URL: http://www.w3.org/TR/rdf-syntax-grammar/.

[40] Paul Grosso, Eve Maler, Jonathan Marsh, and Norman Walsh. *XPointer Framework*. 2003. URL: http://www.w3.org/TR/xptr-framework/.

[41] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language*. 2013. URL: http://www.w3.org/TR/sparql11-query/.

[42] Andreas Harth and Stefan Decker. Optimized Index Structures for Querying RDF from the Web. In: *The Third Latin American Web Congress*. LA-WEB '05. IEEE, 2005. ISBN: 0-7695-2471-0. DOI: 10.1109/LAWEB.2005.25.

[43] Andreas Harth, Katja Hose, Marcel Karnstedt, Axel Polleres, Kai-Uwe Sattler, and Jurgen Umbrich. Data Summaries for On-demand Queries over Linked Data. In: *Proceedings of the 19th International Conference on World Wide Web*. (Raleigh, North Carolina, USA). WWW '10. New York, NY, USA: ACM, 2010, pp. 411–420. ISBN: 978-1-60558-799-8. DOI: 10.1145/1772690.1772733.

[44] Patrick J. Hayes and Peter F. Patel-Schneider. *RDF 1.1 Semantics*. 2014. URL: http://www.w3.org/TR/rdf11-mt/.

[45] John E. Hopcroft. *Introduction to automata theory, languages, and computation*. Pearson Education India, 1979.

[46] Institute for System Programming RAS. *Sedna 3.5, Native XML Database System*. 2011. URL: http://www.sedna.org/.

[47] Nils Klarlund, Thomas Schwentick, and Dan Suciu. XML: Model, Schemas, Types, Logics, and Queries. In: *Logics for Emerging Applications of Databases*. Springer Berlin Heidelberg, 2004, pp. 1–41. ISBN: 978-3-642-62248-9. DOI: 10.1007/978-3-642-18690-5_1.

[48] Tomas Knap, Martin Necasky, and Martin Svoboda. A Framework for Storing and Providing Aggregated Governmental Linked Open Data. In: *Advancing Democracy, Government and Governance*. Vol. 7452. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 264–270. ISBN: 978-3-642-32700-1. DOI: 10.1007/978-3-642-32701-8_23.

[49] Vojtech Kolomicenko, Martin Svoboda, and Irena Mlynkova Holubova. Experimental Comparison of Graph Databases. In: *Proceedings of International Conference on Information Integration and Web-based Applications & Services*. (Vienna, Austria). IIWAS '13. New York, NY, USA: ACM, 2013, pp. 115–124. ISBN: 978-1-4503-2113-6. DOI: 10.1145/2539150.2539155.

[50] Viraj Kumar, P. Madhusudan, and Mahesh Viswanathan. Visibly Pushdown Automata for Streaming XML. In: *Proceedings of the 16th International Conference on World Wide Web*. (Banff, Alberta, Canada). WWW '07. New York, NY, USA: ACM, 2007, pp. 1053–1062. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242714.

[51] Library of Congress. *MARC 21 XML Schema*. 2004. URL: http://www.loc.gov/standards/marcxml/.

[52] Library of Congress. *METS: Metadata Encoding & Transmission Standard: Primer and Reference Manual, Version 1.6 Revised*. 2010. URL: http://www.loc.gov/standards/mets/.

[53] Baolin Liu and Bo Hu. Path Queries Based RDF Index. In: *Proceedings of the First International Conference on Semantics, Knowledge and Grid*. (Beijing, China). Los Alamitos, CA, USA: IEEE, 2005, pp. 91–93. ISBN: 0-7695-2534-2. DOI: 10.1109/SKG.2005.100.

[54] Shiyong Lu, Yezhou Sun, Mustafa Atay, and Farshad Fotouhi. A Sufficient and Necessary Condition for the Consistency of XML DTDs. In: *Conceptual Modeling for Novel Application Domains*. Vol. 2814. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 250–260. ISBN: 978-3-540-20257-8. DOI: 10.1007/978-3-540-39597-3_26.

[55] MarkLogic Corporation. *MarkLogic Server 7*. 2013. URL: http://www.marklogic.com/.

[56] Microsoft Corporation. *Microsoft SQL Server 2008*. 2012. URL: http://www.microsoft.com/sqlserver/2008/.

[57] Laurent Mignet, Denilson Barbosa, and Pierangelo Veltri. The XML Web: A First Study. In: *Proceedings of the 12th International Conference on World Wide Web*. (Budapest, Hungary). WWW '03. New York, NY, USA: ACM, 2003, pp. 500–510. ISBN: 1-58113-680-3. DOI: 10.1145/775152.775223.

[58] Nilo Mitra and Yves Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*. 2007. URL: http://www.w3.org/TR/soap12-part0/.

[59] Irena Mlynkova, Kamil Toman, and Jaroslav Pokorny. Statistical Analysis of Real XML Data Collections. In: *Proceedings of the 13th International Conference on Management of Data*. (Delhi, India). Vol. 6. COMAD '06. Tata McGraw-Hill Publishing, 2006, pp. 20–31.

[60] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. In: *ACM Transactions on Internet Technology* 5.4 (2005), pp. 660–704. ISSN: 1533-5399. DOI: 10.1145/1111627.1111631.

[61] Thomas Neumann and Gerhard Weikum. RDF-3X: A RISC-style Engine for RDF. In: *Proc. VLDB Endow.* 1 (1 2008), pp. 647–659. ISSN: 2150-8097. DOI: 10.1145/1453856.1453927.

[62] Frank Neven. Automata Theory for XML Researchers. In: *SIGMOD Rec.* 31.3 (2002), pp. 39–46. ISSN: 0163-5808. DOI: 10.1145/601858.601869.

[63] Patrick K. L. Ng and Vincent T. Y. Ng. Structural Similarity between XML Documents and DTDs. In: *Computational Science – ICCS 2003*. Vol. 2659. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 412–421. ISBN: 978-3-540-40196-4. DOI: 10.1007/3-540-44863-2_41.

[64] Andrew Nierman and HV Jagadish. Evaluating Structural Similarity in XML Documents. In: *Proceedings of the 5th International Workshop on the Web and Databases*. WebDB '02. 2002, pp. 61–66.

[65] OpenLink Software. *Virtuoso Universal Server 7.1*. 2014. URL: http://virtuoso.openlinksw.com/.

[66] Oracle Corporation. *Java 8 Standard Edition*. 2014. URL: http://www.oracle.com/java/.

[67] Oracle Corporation. *Oracle Database 11g*. 2012. URL: https://www.oracle.com/database/.

[68] Bastian Quilitz and Ulf Leser. Querying Distributed RDF Data Sources with SPARQL. In: *The Semantic Web: Research and Applications*. (Tenerife, Canary Islands, Spain). Vol. 5021. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 524–538. DOI: 10.1007/978-3-540-68234-9_39.

[69] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. 1999. URL: http://www.w3.org/TR/html401/.

[70] RSS Advisory Board. *RSS 2.0 Specification (version 2.0.11)*. 2009. URL: http://www.rssboard.org/rss-specification/.

[71] *Schematron – Rule-based validation – ISO/IEC 19757-3:2006*. 2006. URL: http://www.schematron.com/.

[72] Klaus Dieter Schewe, Bernhard Thalheim, and Qing Wang. Validation of Streaming XML Documents with Abstract State Machines. In: *Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*. (Linz, Austria). iiWAS '08. New York, NY, USA: ACM, 2008, pp. 147–153. ISBN: 978-1-60558-349-5. DOI: 10.1145/1497308.1497339.

[73] Luc Segoufin and Victor Vianu. Validating Streaming XML Documents. In: *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. (Madison, WI, USA). PODS '02. New York, NY, USA: ACM, 2002, pp. 53–64. ISBN: 1-58113-507-6. DOI: 10.1145/543613.543622.

[74] Jakub Starka, Martin Svoboda, and Irena Mlynkova. Analyses of RDF Triples in Sample Datasets. In: *Proceedings of the Third International Workshop on Consuming Linked Data, COLD '12, ISWC '12*. (Boston, MA, USA). Vol. 905. CEUR-WS.org, 2012.

[75] Jakub Starka, Martin Svoboda, Jan Sochna, Jiri Schejbal, Irena Mlynkova, and David Bednarek. Analyzer – A Complex System for Data Analysis. In: *The Computer Journal* 55.5 (2012), pp. 590–615. ISSN: 0010-4620. DOI: 10.1093/comjnl/bxr103.

[76] Jakub Starka, Martin Svoboda, Jiri Schejbal, Irena Mlynkova, and David Bednarek. XML Document Correction and XQuery Analysis with Analyzer. In: *DATESO 2011: Databases, Texts, Specifications, Objects 2011*. (Pisek, Czech Republic). Vol. 706. CEUR-WS.org, 2011, pp. 61–72.

[77] Slawomir Staworko and Jan Chomicky. Validity-Sensitive Querying of XML Databases. In: *Current Trends in Database Technology, EDBT '06, DataX '06*. (Munich, Germany). Vol. 4254. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 164–177. ISBN: 978-3-540-46788-5. DOI: 10.1007/11896548_16.

[78] Slawomir Staworko, Emmanuel Filiot, and Jan Chomicki. Querying Regular Sets of XML Documents. In: *International Workshop on Logic in Databases (LiD)*. (Rome, Italy). 2008.

[79] Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index Structures and Algorithms for Querying Distributed RDF Repositories. In: *Proceedings of the 13th International Conference on World Wide Web*. (New York, NY, USA). WWW '04. New York, NY, USA: ACM, 2004, pp. 631–639. ISBN: 1-58113-844-X. DOI: 10.1145/988672.988758.

[80] Nobutaka Suzuki. Finding an Optimum Edit Script between an XML Document and a DTD. In: *Proceedings of the 2005 ACM Symposium on Applied Computing*. (Santa Fe, New Mexico). SAC '05. New York, NY, USA: ACM, 2005, pp. 647–653. ISBN: 1-58113-964-0. DOI: 10.1145/1066677.1066825.

[81] Nobutaka Suzuki. Finding K Optimum Edit Scripts between an XML Document and a RegularTree Grammar. In: *Proceedings of the 1st Workshop on Emerging Research Opportunities for Web Data Management (EROW 2007), Collocated with the 11th International Conference on Database Theory (ICDT 2007)*. (Barcelona, Spain). Vol. 229. CEUR-WS.org, 2007.

[82] Martin Svoboda. *Corrector Iplementation*. URL: http://www.ksi.mff.cuni.cz/~svoboda/projects/corrector/.

[83] Martin Svoboda. Processing of Incorrect XML Data. MA thesis. Malostranske namesti 25, 118 00 Praha 1, Czech Republic: Department of Software Engineering, Charles University in Prague, Czech Republic, 2010.

[84] Martin Svoboda and Irena Holubova (Mlynkova). Refinement Correction Strategy for Invalid XML Documents and Regular Tree Grammars. In: *Database and Expert Systems Applications*. (Munich, Germany). Vol. 8644. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 308–316. ISBN: 978-3-319-10072-2. DOI: 10.1007/978-3-319-10073-9_25.

[85] Martin Svoboda and Irena Mlynkova. An Incremental Correction Algorithm for XML Documents and Single Type Tree Grammars. In: *Networked Digital Technologies*. (Dubai, UAE). Vol. 293. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, pp. 235–249. ISBN: 978-3-642-30506-1. DOI: 10.1007/978-3-642-30507-8_21.

[86] Martin Svoboda and Irena Mlynkova. Correction of Invalid XML Documents with Respect to Single Type Tree Grammars. In: *Networked Digital Technologies*. (Macau, China). Vol. 136. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2011, pp. 179–194. ISBN: 978-3-642-22184-2. DOI: 10.1007/978-3-642-22185-9_16.

[87] Martin Svoboda and Irena Mlynkova. Efficient Querying of Distributed Linked Data. In: *Proceedings of the 2011 Joint EDBT/ICDT Ph.D. Workshop*. (Uppsala, Sweden). PhD '11. New York, NY, USA: ACM, 2011, pp. 45–50. ISBN: 978-1-4503-0696-6. DOI: 10.1145/1966874.1966882.

[88] Martin Svoboda and Irena Mlynkova. Linked Data Indexing Methods: A Survey. In: *On the Move to Meaningful Internet Systems: OTM 2011 Workshops*. (Crete, Greece). Vol. 7046. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 474–483. ISBN: 978-3-642-25125-2. DOI: 10.1007/978-3-642-25126-9_59.

[89] Martin Svoboda, Jakub Starka, and Irena Mlynkova. On Distributed Querying of Linked Data. In: *DATESO 2012: Databases, Texts, Specifications, Objects 2012*. (Zernov, Rovensko pod Troskami, Czech Republic). Vol. 837. CEUR-WS.org, 2012, pp. 143–150.

[90] Martin Svoboda, Jakub Starka, Jan Sochna, Jiri Schejbal, and Irena Mlynkova. Analyzer: A Framework for File Analysis. In: *Database Systems for Advanced Applications*. (Tsukuba, Japan). Vol. 6193. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 227–238. ISBN: 978-3-642-14588-9. DOI: 10.1007/978-3-642-14589-6_23.

[91] Masako Takahashi. Generalizations of Regular Sets and their Application to a Study of Context-free Languages. In: *Information and Control* 27.1 (1975), pp. 1–36.

[92] Zijing Tan and Liyong Zhang. Repairing XML Functional Dependency Violations. In: *Information Sciences* 181.23 (2011), pp. 5304–5320. ISSN: 0020-0255. DOI: 10.1016/j.ins.2011.07.022.

[93] Zijing Tan, Zijun Zhang, Wei Wang, and Baile Shi. Computing Repairs for Inconsistent XML Document Using Chase. In: *Advances in Data and Web Management*. (Huang Shan, China). Vol. 4505. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 293–304. ISBN: 978-3-540-72483-4. DOI: 10.1007/978-3-540-72524-4_32.

[94] Zijing Tan, Wei Wang, Jian Jun Xu, and Baile Shi. Repairing Inconsistent XML Documents. In: *Knowledge Science, Engineering and Management*. (Guilin, China). Vol. 4092. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 379–391. ISBN: 978-3-540-37033-8. DOI: 10.1007/11811220_32.

[95] Joe Tekli, Richard Chbeir, and Kokou Yetongnon. Structural Similarity Evaluation Between XML Documents and DTDs. In: *Web Information Systems Engineering – WISE 2007*. Vol. 4831. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 196–211. ISBN: 978-3-540-76992-7. DOI: 10.1007/978-3-540-76993-4_17.

[96] Joe Tekli, Richard Chbeir, Agma J. M. Traina, and Caetano Traina. XML Document-Grammar Comparison: Related Problems and Applications. In: *Central European Journal of Computer Science* 1.1 (2011), pp. 117–136. ISSN: 1896-1533. DOI: 10.2478/s13537-011-0005-1.

[97] Alex Thomo, Srinivasan Venkatesh, and YingYing Ye. Visibly Pushdown Transducers for Approximate Validation of Streaming XML. In: *Foundations of Information and Knowledge Systems*. Vol. 4932. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 219–238. ISBN: 978-3-540-77683-3. DOI: 10.1007/978-3-540-77684-0_16.

[98] Thanh Tran and Gunter Ladwig. Structure Index for RDF Data. In: *Proceedings of the Workshop on Semantic Data Management at VLDB 2010*. (Singapore). CEUR-WS.org, 2010.

[99] Octavian Udrea, Andrea Pugliese, and V. S. Subrahmanian. GRIN: A Graph Based RDF Index. In: *Proceedings of the 22nd National Conference on Artificial Intelligence – Volume 2*. (Vancouver, British Columbia, Canada). AAAI Press, 2007, pp. 1465–1470. ISBN: 978-1-57735-323-2.

[100] W3C HTML Working Group. *XHTML 1.0: The Extensible HyperText Markup Language (Second Edition)*. 2000. URL: http://www.w3.org/TR/xhtml1/.

[101] W3C OWL Working Group. *OWL 2 Web Ontology Language: Document Overview (Second Edition)*. 2012. URL: http://www.w3.org/TR/owl2-overview/.

[102] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. In: *Proc. VLDB Endow.* 1 (1 2008), pp. 1008–1019. ISSN: 2150-8097. DOI: 10.1145/1453856.1453965.

[103] Guangming Xing, Chaitanya R. Malla, Zhonghang Xia, and Snigdha Dantala Venkata. Computing Edit Distances Between an XML Document and a Schema and Its Application in Document Classification. In: *Proceedings of the 2006 ACM Symposium on Applied Computing*. (Dijon, France). SAC '06. New York, NY, USA: ACM, 2006, pp. 831–835. ISBN: 1-59593-108-2. DOI: 10.1145/1141277.1141467.

List of Definitions

List of Figures

List of Tables

List of Algorithms