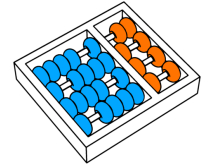


Eduardo de Paula Miranda

“Linked biology — from phenotypes towards
phylogenetic trees.”

“*Conectando dados biológicos — dos fenótipos às
árvores filogenéticas.*”

CAMPINAS
2013



University of Campinas
Institute of Computing

*Universidade Estadual de Campinas
Instituto de Computação*

Eduardo de Paula Miranda

“Linked biology — from phenotypes towards
phylogenetic trees.”

Supervisor: Prof. Dr. André Santanchè
Orientador(a):

“*Conectando dados biológicos — dos fenótipos às
árvores filogenéticas.*”

MSc Dissertation presented to the Post Graduate Program of the Institute of Computing of the University of Campinas to obtain a Mestre degree in Computer Science.

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Computação da Universidade Estadual de Campinas para obtenção do título de Mestre em Ciência da Computação.

THIS VOLUME CORRESPONDS TO THE FINAL VERSION OF THE DISSERTATION DEFENDED BY EDUARDO DE PAULA MIRANDA, UNDER THE SUPERVISION OF PROF. DR. ANDRÉ SANTANCHÈ.

ESTE EXEMPLAR CORRESPONDE À VERSÃO FINAL DA DISSERTAÇÃO DEFENDIDA POR EDUARDO DE PAULA MIRANDA, SOB ORIENTAÇÃO DE PROF. DR. ANDRÉ SANTANCHÈ.

A handwritten signature in black ink that reads "André Santanchè".

Supervisor's signature / *Assinatura do Orientador(a)*

CAMPINAS

2013

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Maria Fabiana Bezerra Muller - CRB 8/6162

M672L Miranda, Eduardo de Paula, 1984-
Linked biology - from phenotypes towards phylogenetic trees / Eduardo de
Paula Miranda. – Campinas, SP : [s.n.], 2013.

Orientador: André Santanchè.
Dissertação (mestrado) – Universidade Estadual de Campinas, Instituto de
Computação.

1. Grafo (Sistema de computador) - Banco de dados. 2. Conexão de dados. 3.
Fenótipo. I. Santanchè, André. II. Universidade Estadual de Campinas. Instituto de
Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Conectando dados biológicos - dos fenótipos às árvores filogenéticas

Palavras-chave em inglês:

Graph (Computer system)- Database

linked data

phenotype

Área de concentração: Ciência da Computação

Titulação: Mestre em Ciência da Computação

Banca examinadora:

André Santanchè [Orientador]

Jorge Alberto Prado de Campos

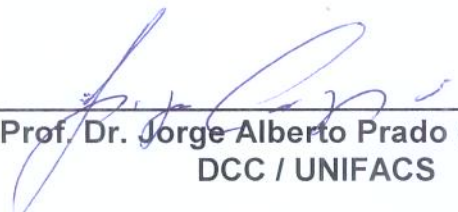
Claudia Maria Bauzer Medeiros

Data de defesa: 22-11-2013

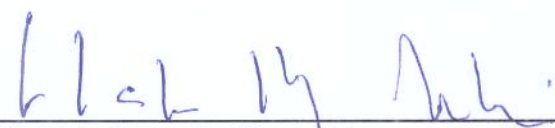
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 22 de novembro de 2013, pela
Banca examinadora composta pelos Professores Doutores:



Prof. Dr. Jorge Alberto Prado de Campos
DCC / UNIFACS



Prof.ª Dr.ª Claudia Maria Bauzer Medeiros
IC / UNICAMP



Prof. Dr. André Santanchè
IC / UNICAMP

Linked biology — from phenotypes towards phylogenetic trees.

Eduardo de Paula Miranda¹

November 22, 2013

Examiner Board/*Banca Examinadora*:

- Prof. Dr. André Santanchè (Supervisor/*Orientador*)
- Prof.^a Dr.^a Claudia Maria Bauzer Medeiros
Institute of Computing – UNICAMP
- Prof. Dr. Jorge Alberto Prado de Campos
Universidade Salvador – UNIFACS
- Prof. Dr. Ricardo da Silva Torres
Institute of Computing – UNICAMP (Substitute/*Suplente*)
- Dr.^a Carla Geovana do Nascimento Macário
CNPTIA – EMBRAPA (Substitute/*Suplente*)

¹Financial support: CNPq scholarship (process 138197) 2011–2013

Abstract

A large number of studies in biology, including those involving phylogenetic trees reconstruction, result in the production of a huge amount of data – e.g., phenotype descriptions, morphological data matrices, phylogenetic trees, etc. Biologists increasingly face a challenge and opportunity of effectively discovering useful knowledge crossing and comparing several pieces of information, not always linked and integrated. In this work, we are interested in a specific biology context, in which biologists apply computational tools to build and share digital descriptions of living beings. We propose a process that departs from fragmentary data sources, which we map to graphs, towards a full integration of descriptions through ontologies. Graph databases mediate this evolution process. They are less schema dependent and, since an ontology is also a graph, the mapping process from the initial graph towards an ontology becomes a sequence of graph transformations. Our motivation stems from the idea that transforming phenotypical descriptions in a network of relationships and looking for links among related elements will enhance the ability of solving more complex problems supported by machines. This work details the design principles behind our process and two practical implementations as proof of concept.

Resumo

Um grande número de estudos em biologia, incluindo os que envolvem a reconstrução de árvores filogenéticas, resultam na produção de uma enorme quantidade de dados – por exemplo, descrições fenotípicas, matrizes de dados morfológicos, árvores filogenéticas, etc. Biólogos enfrentam cada vez mais o desafio e a oportunidade de efetivamente descobrir conhecimento a partir do cruzamento e comparação de vários conjuntos de dados, nem sempre conectados e integrados. Neste trabalho, estamos interessados em um contexto específico da biologia em que biólogos aplicam ferramentas computacionais para construir e compartilhar descrições digitais dos seres vivos. Nós propomos um processo que parte de fontes de dados fragmentadas, que nós mapeamos para grafos, em direção a uma plena integração das descrições através de ontologias. Os bancos de dados de grafos intermediam o processo de evolução. Eles são menos dependentes de esquema e, uma vez que ontologias também são grafos, o processo de mapeamento do grafo inicial para uma ontologia torna-se uma sequência de transformações no grafo. Nossa motivação parte da ideia de que a conversão de descrições fenotípicas em uma rede de relações e a busca de conexões entre elementos relacionados irá aumentar a capacidade de resolver problemas mais complexos suportados por computadores. Este trabalho detalha os princípios de concepção por trás do nosso processo e duas implementações práticas como prova de conceito.

Acknowledgements

First and foremost, I would like to thank my parents, Elizabeth and José, who taught me the value of hard work and education and for the support and dedication throughout my life. My sisters, Mayara and Patrícia for their warmth over the years and for encouraging me to pursue my dreams. I also need to thank Patrícia for helping me to overcome my writing weakness and for correcting my papers. My grandparents, Carlos and Delizete for their wisdom and faith in me. My aunts, Elizete and Simone, my uncles Elizeu and Grei and my cousins, Nicolás and Thais for their continuous encouragement. My girlfriend, Ana Carolina for her support, encouragement and for being part of this journey, even when being at a distance. You experienced all of the ups and downs of my research and I would like to thank you all for the tolerance of my occasional bad moods and general crankiness. Thank you for continuously improving my humor during some of my most stressful moments. Thank you for being even happier than I am with every single achievement. Without you, I may never have gotten where I am today and I am who I am because of you. I would like to express my sincere thanks to my advisor professor André Santanchè for your patient guidance, insight, and most importantly, the friendship during this research. I would like to sincerely thank Kieran Murphy for sacrificing his time correcting my work and providing precious comments. I need to thank Anaïs Grand and Régine Vignes Lebbe for the great experience and creative ideas while we are working together. Next, I need to thank all the people who create such a good atmosphere in the lab, Alessandra, Bruno, Celso, Daniel, Ivelize, Ivo, Jaqueline, João, Joana, Jordi, Lucas, Matheus and Renato. Thank you all for interesting discussions, friendship and good laughs. I need to further thank all my friends at the Institute of Computing, particularly Atílio Gomes, Carlos Trujillo, Daniel Moraes and Raphael Rosa who took the time to share their knowledge during evenings and weekends of studying. I would also like to thank all the members of staff at Institute of Computing.

Finally, I would like to thank the financial support from Brazilian agencies: CNPq (grant 138197/2011-3), the Microsoft Research FAPESP Virtual Institute (NavScales project), CNPq (MuZOO Project and PRONEX-FAPESP), INCT in Web Science(CNPq 557.128/2009-9) and CAPES, as well as individual grants from CNPq.

Contents

Abstract	ix
Resumo	xi
Acknowledgements	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Goals and Contributions	2
1.3 Dissertation Structure	3
2 Unifying Phenotypes to Support Semantic Descriptions	5
2.1 Introduction	5
2.2 Related Work	7
2.3 Common Denominator	8
2.4 From XML Structures to Graphs	12
2.5 Practical Experiment of Unifying Phenotypes	14
2.6 Conclusion	16
3 Linked biology — from phenotypes towards phylogenetic trees	19
3.1 Introduction	19
3.2 Foundations and Related Work	20
3.2.1 Building Phylogenetic Trees	21
3.2.2 Standards for Phenotype Description	21
3.2.3 Phylogenetic Trees and the 3ia Method	23
3.3 Three Layer Method and System Architecture	24
3.4 Unified Graph data model	25
3.5 Link Discovery	27
3.5.1 Similarity Index	29
3.5.2 Practical Implementation of the Similarity Measure	30

3.6	Conclusion	31
4	Linked biology technical aspects – linking phenotypes and phylogenetic trees	33
4.1	Introduction	33
4.2	Basic concepts	34
4.2.1	Standards for Phenotype Description	34
4.2.2	Life Science Identifiers (LSIDs)	36
4.2.3	The proposed graph data model	37
4.3	System Architecture and Implementation Details	38
4.3.1	SDD Parser	38
4.3.2	Tree Output	40
4.3.3	Global Names Resolver (GNR)	41
4.3.4	Graph Importer	42
4.3.5	Graph Database	43
4.3.6	Similarity Index	46
	Practical Implementation of the Similarity Measure	47
4.3.7	Tracing the Evolutionary History	49
4.4	Conclusion	52
5	Conclusions and Extensions	55
5.1	Contributions	55
	Bibliography	57
A	Demonstration	63
A.1	SDDParser.py	63
A.2	TeeOutput.py	70
A.3	GlobalNamesResolver.py	76
A.4	GNRResultObject.py	80
A.5	ITISServices.py	82
A.6	CoLServices.py	84
A.7	GraphImporter.py	88
A.8	SimilarityIndex.py	99
A.9	TraceEvolutionaryHistory.py	102

List of Figures

2.1	Three layers method diagram.	7
2.2	Character-by-taxon matrix	9
2.3	Fragment of SDD Schema with Instances ¹	9
2.4	Symbols and semantic used in the diagrams	10
2.5	Formats for representing phylogenetic data	11
2.6	<i>Property graph model to represents phenotype descriptions.</i>	13
2.7	Varanus knowledge base	15
2.8	Graph Diagram	16
3.1	Fragment of SDD Schema with Instances	23
3.2	Three layer method diagram	25
3.3	General System Architecture.	26
3.4	Property Graph Model	28
3.5	Practical Scenario	29
3.6	Practical Implementation	32
4.1	Fragment of SDD Schema with Instances	35
4.2	Property Graph Model	38
4.3	Retained Tree Example	41
4.4	Real Example	43
4.5	Practical Implementation	48
4.6	Bottom Up Aggregation	50
4.7	Top Down Refining	51
4.8	Evolved Traits Visualization	52

List of Abbreviations

3ia	Three-item analysis
APN	Australian Plant Name Index
C,CS	Characters – Character-states
CoL	Catalogue of Life
EAV	Entity – Attribute – Value
EoL	Encyclopedia of Life
EQ	Entity – Quality
GNR	Global Names Resolver
HTU	Hypothetical Taxonomic Unit
IPNI	International Plant Names Index
ITIS	Integrated Taxonomic Information System
LBS	Lateral Branching System
LSID	Life Science Identifiers
MIAPA	Minimum Information About a Phylogenetic Analysis
NCBI	National Center for Biotechnology Information
OMG	Object Management Group
OTU	Operational Taxonomic Unit
PATO	Phenotype and Trait Ontology
SDD	Structure Descriptive Data
TAO	Teleost Anatomy Ontology
TDWG	Biodiversity Information Standards
TU	Taxonomic Units
uBio	Universal Biological Indexer and Organizer

Chapter 1

Introduction

1.1 Motivation

There are large collections of biological data scattered in various resources that are, most of the time, produced as independent entities and are not linked. Potential links in these data sources can be discovered crossing and comparing pieces of information, as they hold implicit semantics that could enhance the ability of solving more complex problems supported by machines.

The focus of this research are phenotype descriptions and their application in phylogenetic trees, which are resources widely used in a variety of biological studies. A phenotype is a set of observable physical and behavioral characteristics of an individual, resulting from the interaction of its genotype (genetic makeup) with the environment. In this context, recent approaches enrich these descriptions via ontology annotations, using the Entity-Quality (EQ) formalism. EQ is a representation [6] which associates ontology entity terms (E) – e.g., bone or vertebra from the Teleost Anatomy Ontology (TAO) – with quality terms (Q) – e.g., triangular, horizontal or smooth shape from the Phenotype and Trait Ontology (PATO) [18].

Ontologies have gained wide acceptance in biology due to their ability to represent knowledge and also the advantage of querying and reasoning information [23]. Furthermore, semantic web standards allow unique identification of ontology concepts, facilitating interoperability across databases [7, 30]. Several tools have emerged to support annotation of biological phenotypes using ontologies, e.g., Phenex (<http://phenoscape.org/wiki/Phenex>) and Phenote (<http://www.phenote.org/>), both curation tools designed for annotation of phenotype characters with ontology concepts, using the EQ formalism [6].

Dahdul et al. [18] developed a workflow for curation of phenotypic characters from systematic studies. This workflow extracts phenotype characters from a large collection of phylogenetic studies – that have been documented in natural language using a semi-

structured format – and converts them into EQ representations. This process has limited scalability due to the curation process, which is very time-consuming and was executed manually by trained domain experts.

If, on one hand, to represent and integrate phenotype descriptions through the EQ formalism using ontologies appears to be the most promising approach to be adopted, on the other hand, it is not a straightforward task when we depart from existing non EQ resources.

The challenge in this work is to establish a model to represent a common denominator among phenotypical description standards, which will support findings in the latent semantics implicit in the relations. These semantics can guide the interaction between textual descriptions and ontologies. Our approach remodels semi-structured descriptions to a graph abstraction, in which the data can be integrated more easily. Graph transformations are applied for the transition from a semi-structured data representation to a more formalized representation through ontologies.

1.2 Goals and Contributions

The main goal of this research is to design and implement a linked biology approach to automatically connect and combine data from independent semi-structured resources of phenotype descriptions and/or phylogenetic trees, exploiting their latent semantics. We propose a graph data model that plays a crucial role, since it is the basis of our linking discovery and combination process.

The main contributions of the present work are:

- **A graph data model able to represent essential data from phylogenetic trees and phenotype descriptions**, which: (i) is the basis to exploit the latent semantics resulting from the interconnection of phenotype characters; (ii) provides the ground work for the transition from a semi-structured data representation to a more formalized representation through ontologies. The unified model enables to discover and to make explicit the latent semantics through links among previously unconnected information. Its ability of integrating knowledge around taxonomic units will enable, for instance, to generate new research questions, to gain insights and to confront evolutionary hypotheses.
- **The design of an approach and implementation of a prototype** to transform phenotype descriptions and phylogenetic trees – represented as semi-structured documents – into graph representations with the essential information for link discovery and ontology transformation.

An heuristic similarity measure that computes the similarity degree between two morphological character descriptions, which will represent how close related they are.

An algorithm to trace changes in traits of phylogenetic trees. This algorithm was built on top of our proposed graph data model. It searches in a given tree for traits (characters) that might be “responsible” for a tree branching.

A visual tool prototype to analyze phenotype taxonomic units, their characters and the correlation among them.

1.3 Dissertation Structure

The structure of this dissertation is a compilation of two research papers and a technical report, namely:

- Chapter 2: *Unifying Phenotypes to Support Semantic Descriptions*, presented to the VI Brazilian Conference on Ontological Research (Ontobras 2013), which was held in Belo Horizonte, Brazil.
- Chapter 3: Linked biology — from phenotypes towards phylogenetic trees, still to be submitted.
- Chapter 4: Linked biology technical aspects – linking phenotypes and phylogenetic trees, technical report to be submitted.

Each paper/report is presented in a chapter, following an evolutionary perspective of this research. Chapter 2 presents our initial approach, in which we propose a graph data model focused in phenotype descriptions, based in a comparative analysis of four standards related to this kind of description. A practical implementation, built on top of the graph, exploited existing biology phenotype descriptions and their latent semantics to discover links and integrate descriptions.

Chapter 3 presents an enhanced graph data model that links phylogenetic trees to phenotype descriptions. Our first proposed graph data model (Chapter 2) was based on the Entity – Attribute – Value (EAV) representation. While, in the second enhanced graph data model (Chapter 3) we made an important modification in order to make the graph more easily analyzed by a researcher. This chapter also presents a practical implementation, in which we introduce a heuristic that computes the similarity degree between two morphological character descriptions; it aims at supporting biologists in perceiving correlations.

Chapter 4 summarizes the main functionalities of the system and presents an algorithm to trace the phylogenetic history of trait changes. Chapter 5 presents the conclusions of this dissertation and future work. Appendix A shows details of the system architecture and implementation, in order to document the functionalities and operational features of the system.

Chapter 2

Unifying Phenotypes to Support Semantic Descriptions

2.1 Introduction

Bioinformatics is the science of integrating, managing, mining and interpreting information from biological data [22]. In the life science field, there are a large number of distributed biological datasets freely available and ready to use. However, this wealth of information has hardly been tapped even today due its distributed nature, heterogeneity and complex data types and representation [39]. In this scenario, their combination and interconnection are barely feasible [42]. A massive amount of relevant information is hidden in the potential connection of unrelated files.

In this work we are interested in a specific biology context, in which biologists apply computational tools to build and share digital descriptions of living beings as phenotypes. These descriptions are a fundamental starting point for several biology tasks, like living beings identification and tools for phylogenetic tree analysis. Even though the last generation of these tools is based on open standards (e.g., XML), the descriptions are still based on textual sentences in natural language [6].

Semantic integration in this context is one of the main challenges. Besides ontologies to support phenotype description, there are tools to annotate descriptions by associating ontology concepts to textual descriptions [6]. This distinction between description and their annotations based on ontologies does not consider that descriptions can conversely contribute to ontology expansion and revision. The challenge in this work is to establish a model to represent a common denominator among phenotypical description standards, which will support findings in the latent semantics implicit in relations in a strategy inspired by folksonomies. These semantics can guide the interaction between textual descriptions and ontologies.

In a previous work [2], we showed that the latent semantics presented in tags and their correlations, as a product of an organic work collectively produced by a community on the web (the folksonomies), can be exploited to expand and review ontologies. While the model behind folksonomies is based on the correlation of three elements – tags, resources and users – descriptions in the biological context present a more complex and specialized structures. Co-occurrence is a strong principle we considered to extract latent semantics. The main idea is that the set of tags put together in a given resource can provide a “context” to interpret each tag. Consider a tag *cell*, which can have a distinct interpretation according to the context. The co-occurrence with the tags *cytoplasm* or *organelle* will put it in the biology context. Moreover, the compilation of data concerning the occurrence and co-occurrence of millions of tags can support the analysis of similarity among terms – see more details in [2]. We consider that we can apply an equivalent technique to put terms of phenotype descriptions in a context, to improve their interpretation and correlation.

The present paper addresses this problem in exploiting existing biology assets related to phenotypic descriptions, and the latent semantics resulting from their interconnection, to support their development towards a richer semantical representation, as part of ontologies. It implies promoting relations among concepts to first class citizens. Accordingly, we designed a three layered method illustrated in Figure 2.1, in which graph databases intermediate this evolution process from fragmentary data sources to accomplish full integration descriptions as ontologies.

Our approach remodels semi-structured descriptions to a graph abstraction, in which the data can be integrated more easily. Graph transformations are applied for the transition from a semi-structured data representation to a more formalized representation through ontologies. As we will further explain, this graph representation will also support an analytical tool to compare data across studies, wherein it will help evolutionary biologists to answer evolutionary questions. This paper presents a work in progress concerning the first step of this method, focusing in the integration of data from the semi-structured data layer and their transition to the graph data abstraction layer. Our proposed graph-based model is derived from a comparative analysis among four standards related to phenotype description, plus a practical experiment.

This paper is organized as follows: Section 2.2 summarizes the related work; Section 2.3 presents the comparative analysis which subsidizes our minimal common denominator model; Section 2.4 presents our graph-based model; Section 2.5 shows a practical experiment of unifying phenotypes; Section 2.6 presents concluding remarks.

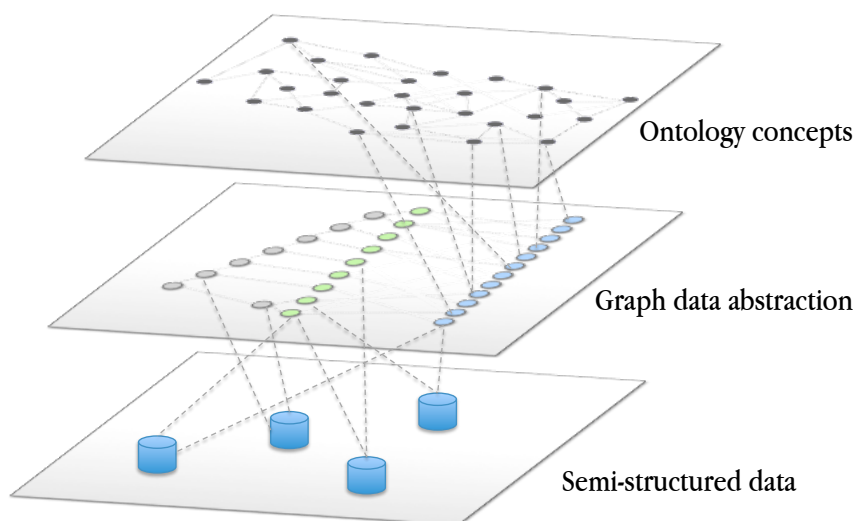


Figure 2.1: Three layers method diagram.

2.2 Related Work

Integration is a key point as humans are progressively unable of handling the sheer volume of data presented [8]. It is an important step towards knowledge discovery [28]. The integration of digital phenotype descriptions is a relevant challenge in this context since they support fundamental biology tasks as the building of identification keys for living beings and can support the creation of a complete evolutionary Tree of Life [39] assembling genomic and morphological data so as to congregate the phylogenetic relationships among all living or extinct organisms [15]. Likewise, integrating these data may contribute to better understanding of how a morphological trait became organized and evolved over time [29].

Recent approaches enrich descriptions via ontology annotations, using the Entity-Quality (EQ) formalism for phenotype modeling. EQ is a representation [6] which associates ontology entity terms (E) – e.g., bone or vertebra from Teleost Anatomy Ontology (TAO) – with quality terms (Q) – e.g., triangular, horizontal, smooth from the Phenotype and Trait Ontology (PATO) [18]. Ontologies have gained wide acceptance in biology due to their ability of representing knowledge and also the advantage of querying and reasoning information [23]. Furthermore, semantic web standards to represent ontology concepts with unique identifiers facilitates interoperability across databases [30]. Recently, several tools have emerged to support annotation of biological phenotypes using ontologies, e.g., Phenex (<http://phenoscape.org/wiki/Phenex>) and Phenote (<http://www.phenote.org/>), both curation tools designed for annotation of phenotypic characters with ontology concepts using EQ formalism [6].

[18] developed a workflow for curation of phenotypic characters extracted from scientific publications. It is important to note the limitations of this curation process, considering that it is very time-consuming since it is manually carried out by domain experts.

2.3 Common Denominator

There is a wide variety of representation formats for phenotype description, adopted by information systems and open standards, which represent differently the same information. In this section, we analyze four of them – Xper², SDD, Nexus and NeXML – looking for a minimal common denominator, which is the foundation for our graph-based model, to be used to link related information.

SDD, Nexus and NeXML are widely adopted open standards further detailed. Xper² (<http://lis-upmc.snv.jussieu.fr/lis/>) is a management system adopted by the systematist community, for the storing, editing and analyzing of phenotype descriptive data. It focuses mainly on taxonomic descriptions, allowing creation, sharing and comparison of identification keys [47, 48]. Xper² was developed in the Laboratoire Informatique & Systématique of the University Pierre et Marie Curie and this work is part of a bigger project in collaboration with this lab. Therefore, Xper² was adopted for our practical experiments.

In order to illustrate our analysis, let us consider a practical case, in which a biologist is building a phenotype description of monitor lizards (genus *Varanus*). The process starts with the biologist collecting observations of lizards, organized as characters and character states (C, CS). [41] defined character as “*a feature of organisms that can be evaluated as a variable with two or more mutually exclusive and ordered states*”. The observations involved the species *Varanus albiguralis* and *Varanus brevicauda*. The final result is the character-by-taxon matrix illustrated in Figure 2.2.

In order to transform these observations to digital records and generalize them – e.g., devising general characters and states observed in a genre of monitor lizards – the biologist will use a tool like Xper². Phenotypes descriptions can be stored in the Xper² native format or can be exported to the SDD open format. The Structure Descriptive Data (SDD) (<http://wiki.tdwg.org/SDD>) is a platform and application-independent XML-based standard developed by the Biodiversity Information Standards (historic acronym: TDWG) for recording and exchanging descriptions of biological and biodiversity data of any type [26]. SDD is adopted by several other phenotype description tools – e.g., Lucid Central (<http://www.lucidcentral.org>) and Linnaeus II (<http://www.eti.uva.nl/>).

We further introduce some key elements of the SDD format, which are recurrent in the formats confronted in this section. A SDD description comprises, in a single file, a

	nostrils' form	transversal section of the tail	nuchal scales
<i>Varanus albiguralis</i>	2	1	2
<i>Varanus brevicauda</i>	1	2	1

Nostrils' form
 1 – well round
 2 – oval or split-like

Transversal section of the tail
 1 – laterally compressed
 2 – roundish

Nuchal scales
 1 – same size than head scales
 2 – bigger than head scales

Figure 2.2: Character-by-taxon matrix

domain schema and its instances. Figure 2.3 shows a diagram with a fragment of a SDD file containing the description of a varanus lizard. A (C,CS) description in SDD has two main blocks: (i) defines the characters involved and their possible states – Figure 2.3 top; (ii) describes an Operational Taxonomic Unit (OTU) using the characters defined in (i) – Figure 2.3 bottom. OTU is a biology term which refers to a given entity in sampling level adopted to the study – e.g., a specimen, a gender etc.

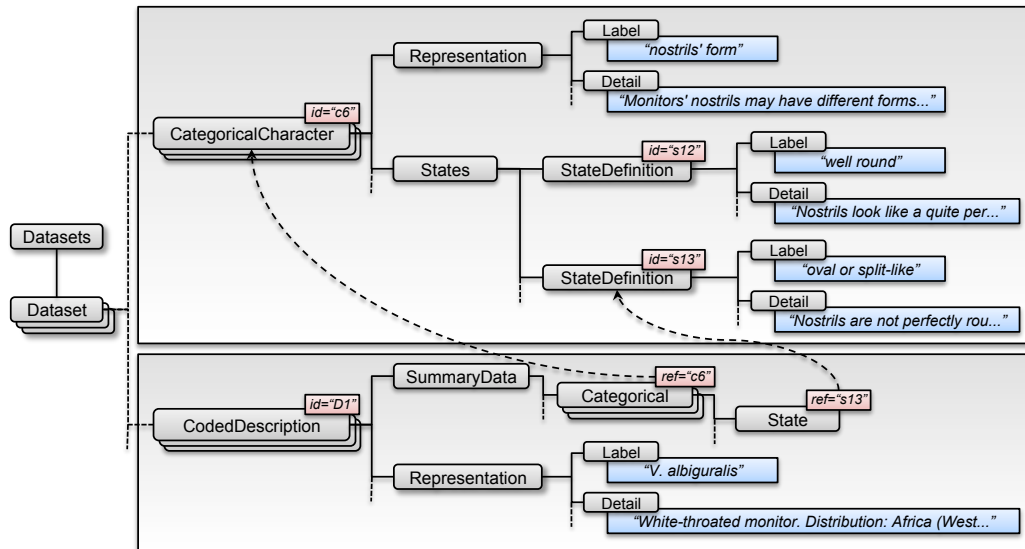


Figure 2.3: Fragment of SDD Schema with Instances ¹

<CategoricalCharacter>s and their <States> (shown in Figure 2.3 top) are primitives to describe an OTU [26]. Each <CategoricalCharacter> has its <Representation> – com-

¹Knowledge base: <http://lis-upmc.snv.jussieu.fr/xper2/infosXper2Bases/liste-bases-recherche.php>

prising a label and a description as plain texts – and a set of *<StateDefinition>* elements with their possible states. *<CategoricalCharacter>* and *<StateDefinition>* elements defined here will be referred throughout the XML document by their ids.

The *<CodedDescription>* (Figure 2.3 bottom) links the OTU being described to *States* of each *<CategoricalCharacter>*. It has two essential items: (i) the OTU being described, where its name and description are listed in natural language under *<Representation>*; (ii) a set of character and values (*<Categorical>* and *<State>*), which address the characters defined in the previous section through the *ref* attribute. It is possible and usual to define multiple states for a character of a given OTU. A first integration, problem observed here is that each character or OTU described does not have a global unique identification among documents. Therefore, the description can only be used by the document where it was declared and it is not possible to guarantee the equivalence of two or more *<CategoricalCharacters>*.

In Figure 2.5 we expand our analysis to the Xper² native format, Nexus and NeXML. Our study addresses mainly morphological character descriptions. Figure 2.5 provides simplified diagrams focusing on the elements to record descriptions, which will be confronted here. Figure 2.4 presents the symbols adopted in the diagram. All the formats adopt XML and the symbols represent the relations among elements and their respective cardinality. Five types of elements, which are focus of our analysis, receive special symbols: the **Entity** being described, which can be a taxon or a specimen; the **Character** definition and its respective association with entities (**Character instance**); the **State** definition and its respective association with entities (**State instance**).

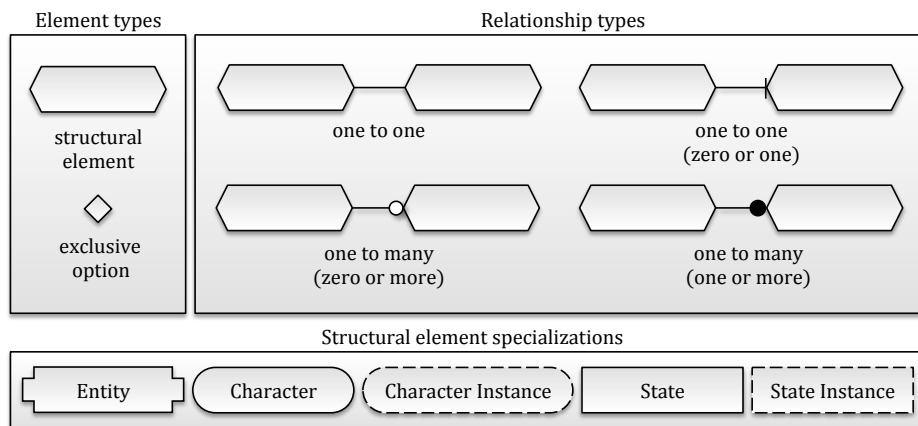


Figure 2.4: Symbols and semantic used in the diagrams

Nexus [31] is an extensively used file format developed for storage and exchange of phylogenetic data, including morphological and molecular characters, taxa distances, genetic codes, phylogenetic trees etc. It was designed in 1987 and it is still used by many popular

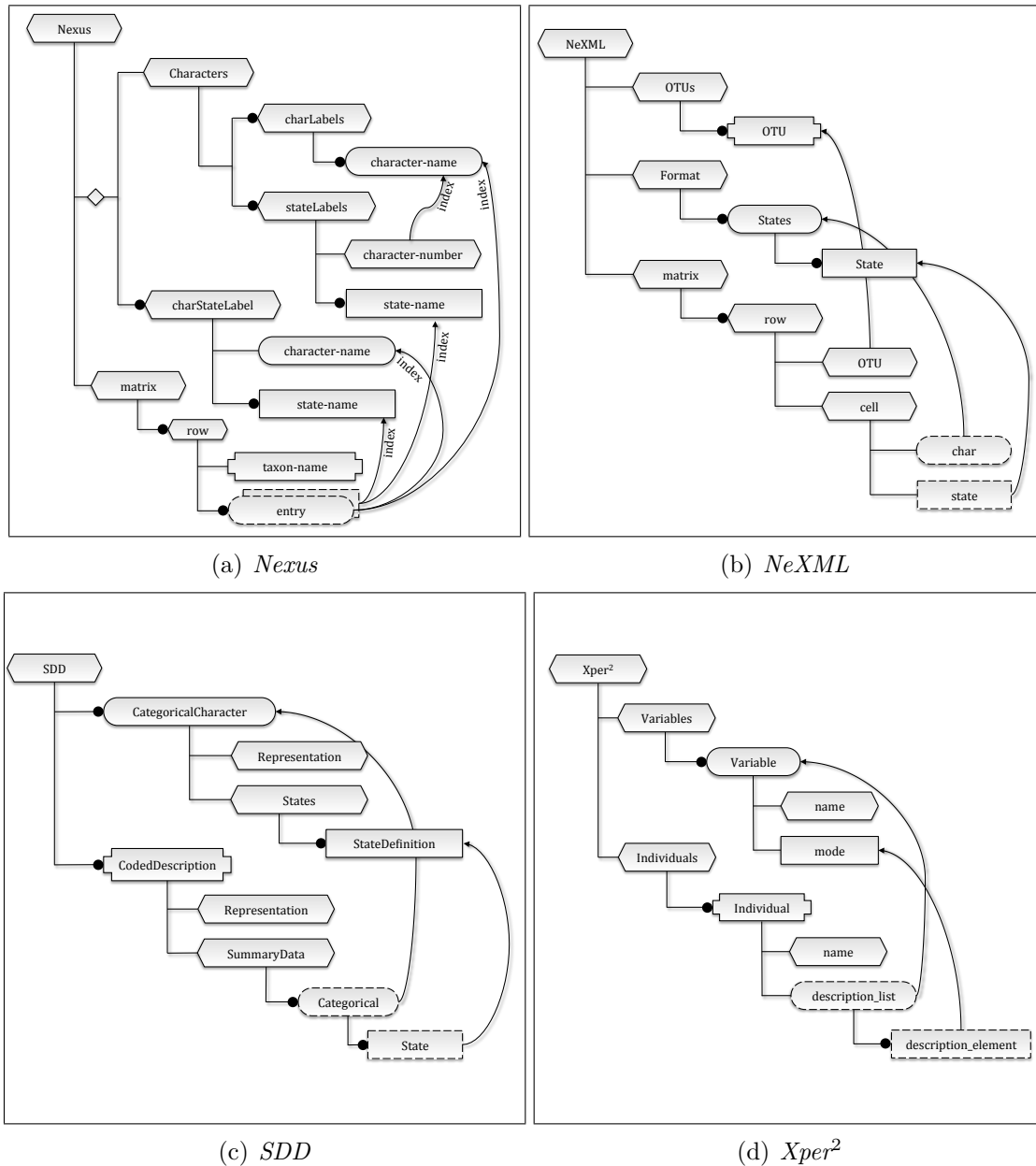


Figure 2.5: Formats for representing phylogenetic data

software as Xper² (<http://lis-upmc.snv.jussieu.fr/lis/>), Mesquite (<http://mesquiteproject.org/>), MrBayes (<http://mrbayes.sourceforge.net/>) and data repositories, like TreeBASE (<http://treebase.org/>) and Dryad (<http://datadryad.org/>). Nexus gathers together (C,CS) based descriptions and related trees [49].

NeXML (<http://www.nexml.org>) [49] is a standard inspired by the Nexus. It supports and extends Nexus functionalities and addresses some Nexus limitations – e.g., connects

objects with ontology concepts, supports citations and annotations [49]. In order to accomplish full compatibility and interoperability among different environments, NeXML defines a formalized XSD grammar and enables semantic annotations of any element in a NeXML document, which goes towards to a “Minimum Information About a Phylogenetic Analysis” (MIAPA) standard.

These comparative diagrams show that even if the structures are arranged differently, they address the same key elements. All formats organize data in accordance with the (C,CS) data model that, in practice, is an entity-attribute-value (EAV) model, in which entities are OTUs, attributes are characters and values are character-states [49]. Nexus and NeXML formats define a matrix, in which OTUs are listed in rows, characters are columns and the cells contain a numeric code for a specific character-state (see Figure 2.2). Although Xper² and SDD do not define a matrix, both formats have a similar structure to describe OTUs with their (C, CS) records.

2.4 From XML Structures to Graphs

The next step in our Three Tier Method is designing a graph model. In a previous work [2], we have compared several approaches to capture latent relations+semantics among tags produced collaboratively. Graph models to represent and analyze data were a common denominator. The role of the graph is not to reflect all details of the original model. The central challenge is how to abstract key elements, for which we are looking for potential relations to be discovered. It is a movement from the latent semantics to an explicit semantics expressed as links.

On one hand, we devised in the previous section the common denominator we are looking for: OTUs, character and character states. On the other hand, a second important ingredient is devising what is our target in ontologies. As mentioned in Section 2.2, a predominant ontology model for phenotype descriptions is the Entity-Quality (EQ) [6]. An Entity refers to the “part” of the OTU being described, which is related to one or more Qualities. In a comparison with the (C, CS) approach, a Character comprises an Entity plus the Quality involved in the description in a single textual sentence. A State is a complementary part of the Quality. Even though it is not a trivial task to split Characters into their components of Entity and Quality, a first step will be linking disperse elements referring to the same semantic concept.

Departing from the key elements identified in the previous section, we can devise the following linking discovery challenges:

- Which OTUs in the graph refer to the same real world OTU (link OTU-OTU)?
- Which characters can be applied to each OTU (link OTU-character)?

- Which states for each character can be observed in each OTU (link OTU-character-state)? Conversely, which OTUs have a given character+state?

The answer to these questions will enable to integrate, summarize and compare data concerning each OTU and each character. Therefore, it becomes possible to answer queries like:

- What are the possible colors of a *Varanus* tongue?
- Which animals present an oval nostrils form?

The discovery process is carried by graph transformations. As graphs are crucial for our modeling approach, our method was built over graph databases. These databases reduce the gap between how data is modeled (as graphs) and how it is stored. It is capable of representing data structures with high abundance. Compared with relational databases, graph databases do not require join operations because it is done implicitly traversing the graph from node to node. Graph databases are less schema-dependent and for this reason, they can scale more easily in size and complexity as the application evolves.

The questions stated before were the basis to conceive the model presented in Figure 2.6. We adopted the *property graph* model, in which nodes and relationships can maintain extra metadata as a set of key/value pairs. Moreover, relationships are typed, enabling to create multi-relational networks with heterogeneous sets of edges. Different from single-relational networks, in which edges are of the same type, multi-relational networks are more appropriate to represent complex domain models, due the variety of relationship types in the same graph [45].

In our graph model, OTUs and character-states are nodes connected by characters (edges). Therefore the statement “*V. albiguralis* has a well round tail shape” becomes *V. albiguralis* (node) → tail shape (edge) → well round (node).

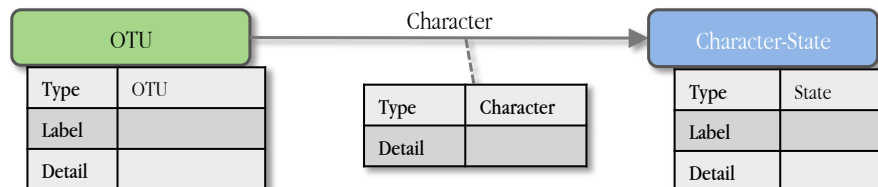


Figure 2.6: *Property graph model to represents phenotype descriptions.*

2.5 Practical Experiment of Unifying Phenotypes

We have implemented an automatic process to ingest SDD files into a graph database, in order to show the linking possibilities raised by our model. In our experiments, we use the Neo4j (<http://www.neo4j.org/>), an open-source graph database. Our data integration processing flow is divided into the main stages: preprocessing, data ingestion, data linkage.

One of the problems faced in bioinformatics is related to the identification of objects within and across repositories [38]. More precisely, an object may refer to a taxon, gene, anatomical feature, phenotypic description, geographical location etc. Uniquely identifying those objects is undoubtedly a key point for the success of our proposed solution.

In order to address this issue, some organizations – e.g., Universal Biological Index and Organizer (uBio), Integrated Taxonomic Information System (ITIS), Catalogue of Life (CoL), The International Plant Names Index (IPNI), National Center for Biotechnology Information (NCBI) etc. – incorporated into their projects the Life Science Identifiers (LSIDs), which was proposed by the Object Management Group (OMG) (<http://www.omg.org/>). LSID is a persistent, location-independent resource identifier, whose purpose is to uniquely identify biological resources [16]. The persistent property refers to the fact that LSID identifiers are unique, can be assigned to only one object forever and they never expire. The location-independent property specifies that each authority locally creates LSIDs and they are the responsible to guaranteeing the uniqueness of LSIDs.

We applied LSIDs to unify OTUs in the graph referring to the same real world object. In order to find a valid LSID, we adopted the Global Names Resolver (GNR) web service (<http://resolver.globalnames.org/>) that executes exact or fuzzy matching against canonical forms of scientific names in 170 distinct data sources. The Canonical form (cf) is the simplest, most complete and unambiguous form of a name. The Canonical form of scientific names consists of the genus and species – when applied – with no authorship, rank, nomenclatural annotation or subgenus.

Our system used three of the six types of matching offered by the GNR resolver: (i) exact matching; (ii) exact matching of canonical forms – this process reduce a given name to its canonical form and checks it with an exact match; (iii) fuzzy matching of canonical forms – uses a modified version of the TaxaMatch algorithm [43] and it intends to work around misspellings errors. It does a fuzzy match of the canonical form of a given name – even with mistakes – against spellings considered correct. The GNR resolver reports the matching quality (“*confidence score*”) for each match.

The matching module of the system is still a work in progress, but we already have obtained some relevant results to show the viability of our approach. From the LIS knowledge base we collected 7 distinct morphological descriptions: genus *Varanus*; species *Varanus*

gouldii, *Varanus timorensis*, *Varanus auffmanbergi* and *Varanus scalaris*; species groups *Varanus indicus*, *Varanus prasinus*, *Varanus salvator*; and Australian spiny-tailed monitor lizards. Through Xper² those morphological descriptions were exported to the SDD format and imported into the graph database, with no preprocessing. Figure 2.7(a) shows an overview of the resulting graph without labels. We can note the disconnectedness of the graph (7-partite graph). On the other hand, Figure 2.7(b) shows the same knowledge after employing the LSID unification. The graphs became connected. Before applying the LSID unification the graph had 74 distinct taxonomic units (TUs). After performing the LSID unification its total reduced to 44 TUs, i.e., 30 taxonomic units (40%) were recurring and were integrated in a single node.

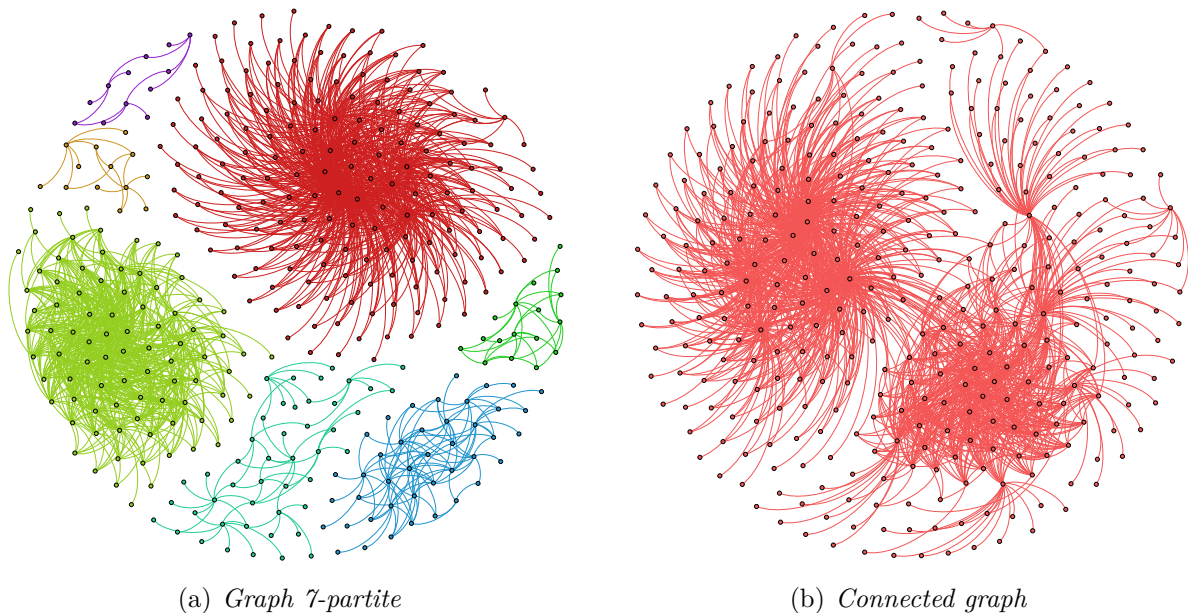


Figure 2.7: Varanus knowledge base

The next step is to link equivalent characters of the same OTU, enabling integration of states of the same character. In the present stage of this research we apply a simple matching algorithm. One example of our preliminary results is presented in the diagram of Figure 2.8. As can be seen, our algorithm was able to unify all “nuchal scales” characters, by defining the same type to the edges. Moreover, we unified and congregated the possible states observed for this character across different description files.

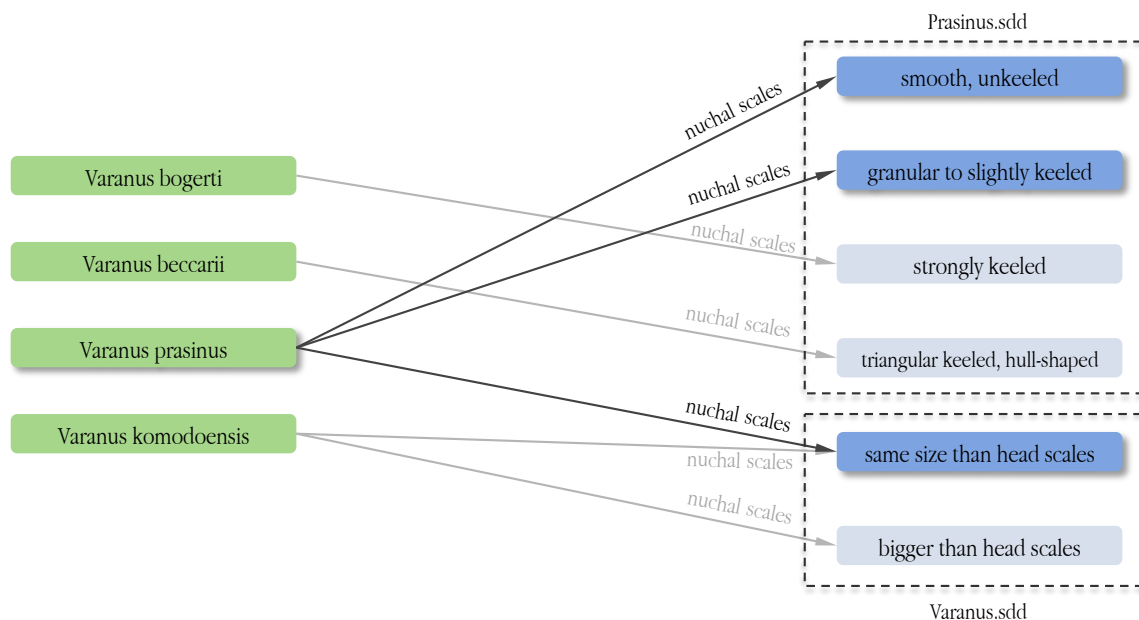


Figure 2.8: Graph Diagram

2.6 Conclusion

Several initiatives propose to relate phenotype descriptions with ontologies to enable a semantic integration. The challenge is how to expand and revise the ontology while new descriptions were created. Tools which annotate descriptions with ontologies address them as an external artifact crafted apart, disregarding the synergy between building an ontology and using it. [Shirky 2005] emphasizes the importance of the semantics organically built by a community, where a binary categorization approach – in which a concept A “is” or “is not” part of a category B – to a probabilistic approach – in which a percentage of people relates A to B. This work contributes in this direction. Inspired by previous work, which explores latent semantics in folksonomies, this work analyzes standards to describe phenotypes to find a common denominator, which is the bases to link descriptions.

The main contribution of this work is to create the basis to exploit the latent semantics in the descriptions. The viability and the potential of our approach were tested by experiments. These experiments are the first steps to exploit a bigger latent semantics scenario. Moreover, having the capability of integrating knowledge around taxonomic units will enable, for instance, evolutionary biologists to generate new research questions, gain predictive insight or confront evolutionary hypotheses. More complete answers might be provided as new data sources are integrated.

Our representation in a graph database is aligned with the RDF [32] graph-based representation, which will be the next step to achieve the third layer. The challenge will be to map labels of character/character-states in RDF properties/values. The unification of characters and states, as shown on this preliminary work, is a first and high relevant step for this mapping. Since several ontologies related to phenotype descriptions are in OWL, the relations discovered in our graph can subsidize a better matching of labels and concepts in OWL ontologies by confronting relations. For example, to enhance the match of a character label (in the graph database) with an OWL property, it is possible to consider the states allowed by the character, confronting them with the property range (values allowed by the property).

There are several possible ways to extend this work. One possible way is to incorporate morphological descriptions stored in other knowledge bases, e.g., MorphoBank (<http://morphobank.org/>) or Dryad (<http://datadryad.org/>). Another direction is to investigate correlations between *State* nodes and ontology terms.

Chapter 3

Linked biology — from phenotypes towards phylogenetic trees

3.1 Introduction

Traditionally, evolutionary biologists organize and classify the extant and extinct organisms around the classical Tree of Life model. This model is an abstract form of representation of hypotheses about evolutionary relationships where all taxa (i.e. groups of organisms) are related according to the characteristics they share. This representation is called a phylogenetic tree. Besides relationships among taxa, a phylogenetic tree provides hypotheses about the homology (i.e. sameness) of entities (organs, anatomical features etc.). In this model, taxa are the leaves of the tree and the internal nodes are common ancestors, or hypothetical taxa from which the leaves are differentiated. For instance, given the taxa Human, Horse and Frog, a tree can indicate that Human and Horse are more closely related than they are with Frog. The node grouping Human and Horse is a taxon called Mammalia. The characteristics shared by all mammals (i.e. taxa connected to the Mammalia node) are the presence of hairs and of mammary glands, among others.

Biologists usually work in their own domain of expertise. This domain is centered on the studied taxonomic group and also depends on the source of data: some taxonomists use molecular data, others use morphological data, or behavioural, ecological, physiological data etc. Each phylogenetic tree that is published is a particular view.

In spite of several initiatives to publish open data (e.g., Scratchpads, Dryad, EU-BrazilOpenBio and TreeBASE) and to combine phylogenetic trees (e.g., Open Tree of Life), there is still a high amount of latent knowledge hidden in potentially linkable data, which are fragmented in several heterogeneous datasources. In this paper, we focus on the interconnection of phylogenetic information with the observations and descriptions of living (extant and extinct) beings, aiming at comparing this phylogenetic information

(such as homology hypotheses, characters and trees), related to its source, that is namely the observations and descriptions of taxa.

This heterogeneous multitude of resources can be seen as a dataspace [21], where pieces of data maintain unexploited potential links. This work addresses this problem in a specific scenario. We gather together in a graph database data coming from distinct sources, containing phenotype descriptions and phylogenetic trees. This graph subsidizes links discovery, aimed at supporting biologists in the analysis and comparison of phylogenetic information (such as homology hypotheses, characters and trees) of hypothetical phylogenetic trees. The graph model was designed to afford publication on the Web in a Linked Data approach. Moreover, in a previous work [2], we showed that the latent semantics of data resulting from an organic work, collectively produced by a community on the Web, can be exploited to expand and review ontologies. Linking data in graphs is a first step towards ontologies.

This paper is organized as follows: Section 3.2 summarizes the foundations and related work; Section 3.3 presents our three layer method and the architecture of our system; Section 3.4 presents our graph-based model; Section 3.5 shows a practical implementation and our approach to discover links based on similarity; Section 3.6 presents concluding remarks.

3.2 Foundations and Related Work

Several initiatives aim at facilitating the interchange of scientific data. Dryad (<http://datadryad.org/>) is an online repository to share files associated with published research papers. In the biodiversity domain, the EDIT Platform for Cybertaxonomy (<http://wp5.e-taxonomy.eu/>) and the Scratchpads (<http://scratchpads.eu/>) are two types of platform to store and publish more or less structured data. The EUBrazilOpenBio project (<http://www.eubrazilopenbio.eu/>) is developing “an open-access platform from the federation and integration of existing European and Brazilian infrastructures and resources” for the biodiversity scientific community. The BioVel project is a Biodiversity Virtual e-Laboratory. It stores both data and workflows, allowing to process data from cross-disciplinary sources. Phylogenetic trees and related data (e.g., data matrices) published in research papers can be gathered and stored in the online repository TreeBASE (<http://treebase.org/>). The Open Tree of Life project (<http://opentreeoflife.org/>), still under development, intends to produce the first-draft of a complete tree that will combine existing smaller trees with phylogenetic and taxonomic knowledge of every taxa [50].

3.2.1 Building Phylogenetic Trees

In order to build taxonomic classifications (such as phylogenetic trees), biologists gather information about taxa of their interest and structure this information upon the character / character-state (C,CS) formalism. Under this formalism, a character is a statement about an entity (e.g. an organ, an anatomical feature, a part of the organism etc.). This entity is described under a particular property that can take multiple values. For instance, the statement “shape of the leaf” is a character in which the entity is the “leaf” and the property is the “shape”. The values (i.e. character-states) that the shape can take might be, for example, “elongated”, “bilobed”, “trilobed” among others. This information is usually organized in data matrices [52], in which operational taxonomic units (OTUs) are listed in rows, characters are columns and the cells contain a numeric code for a specific character-state.

Some biologists use unordered character-states to apply phylogenetic analysis, whereas others prefer to relate the character-states prior to the analysis. In the latter case, the character-states can be related following some punctual order, or hierarchy. These relationships represent hypotheses about homology of the described entities [10, 35]. For instance, “trilobed” and “bilobed” leaves can be hypothesized to be homologous as “lobed leaves”, i.e. they are the same despite their variety of form. In other words, “trilobed” and “bilobed” leaves are more closely related than they are to “elongated” leaves. The relationship among entities (here, leaves) can be extended among taxa: all taxa bearing lobed leaves (either “trilobed” or “bilobed”) are more closely related than they are to taxa with other leaf shapes such as elongated leaves. Each character defined in the context of a phylogenetic study relies on a distinct homology hypothesis. Different hypotheses can yield conflictual relationships among taxa. In the context of morpho-anatomical studies, phylogenetic methods aim to minimize this conflict based on the parsimony criterion. The method of maximum parsimony can be applied to unordered and ordered character-states [51, 46, 10]. For hierarchical characters, the method based on three-item statements (i.e. elementary hierarchies including three taxa) is applied. This is called three-item analysis (3ia) [36]. Other methods are available – such as Bayesian or Maximum likelihood analyses (based on probabilities) – and can be applied to molecular characters. The output of a phylogenetic analysis is one or several phylogenetic tree(s).

3.2.2 Standards for Phenotype Description

There is a wide variety of representation formats for phenotype description adopted by information systems and open standards, which represent differently the same information. In [34] we analyze four of them – Xper², SDD, Nexus and NeXML – looking for a common denominator which is the foundation for our graph-based model. SDD, Nexus and

NeXML are widely adopted open standards. Xper² (<http://lis-upmc.snv.jussieu.fr/lis/>) is a management system adopted by the systematist community, for the storing, editing and analyzing of phenotype descriptive data. It focuses mainly on taxonomic descriptions, allowing creation, sharing and comparison of identification keys [47, 48]. Xper² was developed in the Laboratoire Informatique & Systématique of the University Pierre et Marie Curie and this work is part of a bigger project in collaboration with this lab. Therefore, Xper² was adopted for our practical implementation.

In order to transform phenotype observations to digital records and generalize them – e.g., devising general characters and states observed in a genus of monitor lizards – the biologist may use a tool as Xper². Phenotype descriptions can be stored in the Xper² native format or can be exported to the SDD open format. The Structured Descriptive Data (SDD) (<http://wiki.tdwg.org/SDD>) is a platform and application-independent XML-based standard developed by the Biodiversity Information Standards (historic acronym: TDWG) for recording and exchanging descriptions of biological and biodiversity data of any type [26]. SDD is adopted by several other phenotype description tools – e.g., Lucid Central (<http://www.lucidcentral.org>) and Linnaeus II (<http://www.eti.uva.nl/>).

We further introduce some key elements of the SDD format, which are recurrent in the formats confronted in [34]. A SDD description comprises, in a single file, a domain schema and its instances. Figure 3.1 shows a diagram with a fragment of a SDD file containing the description of a varanus lizard. A (C,CS) description in SDD has two main blocks: (i) defines the characters involved and their possible states – Figure 3.1 top; (ii) describes an Operational Taxonomic Unit (OTU) using the characters defined in (i) – Figure 3.1 bottom. OTU is a biology term which refers to a given taxon at the rank adopted to the study – e.g., a specimen, a species, a genus etc.

*<CategoricalCharacter>*s and their *<States>* (shown in Figure 3.1 top) are primitives to describe an OTU [26]. Each *<CategoricalCharacter>* has its *<Representation>* – comprising a label and a description as plain texts – and a set of *<StateDefinition>* elements with their possible states. *<CategoricalCharacter>* and *<StateDefinition>* elements defined here will be referred throughout the XML document by their ids. The *<CodedDescription>* (Figure 3.1 bottom) links the described OTU to *States* of each *<CategoricalCharacter>*. It has two essential items: (i) the described OTU, where its name and description are listed in natural language under *<Representation>*; (ii) a set of character and values (*<Categorical>* and *<State>*), which address the characters defined in the previous section through the *ref* attribute. It is possible and usual to attribute multiple character-states for a given OTU (i.e. in case of polymorphism). A first integration, problem observed here is that each character or OTU described does not have a global unique identification among documents. Therefore, the description can only be used by the document where it was declared and it is not possible to guarantee the equivalence of

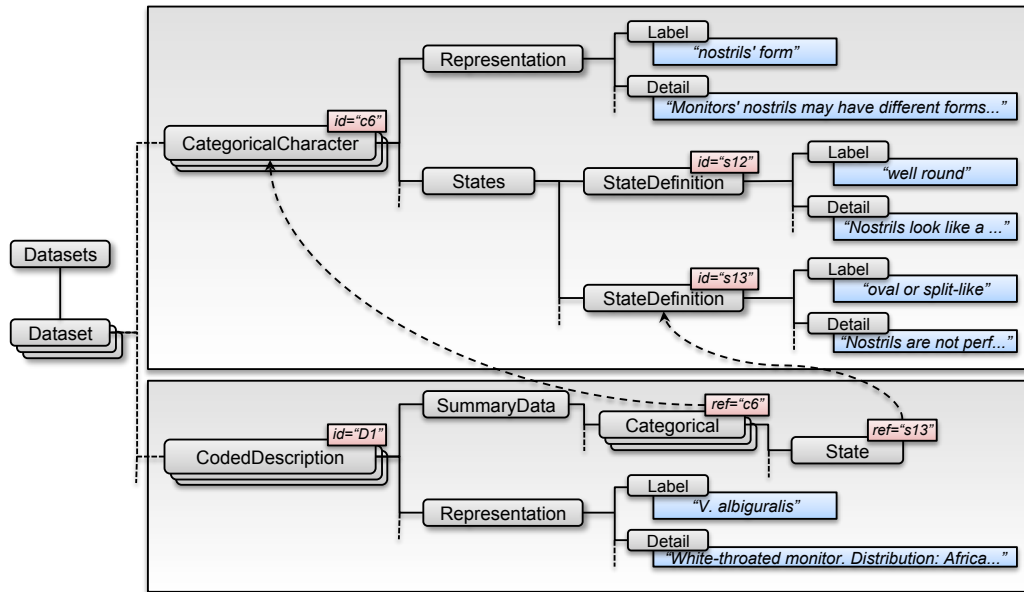


Figure 3.1: Fragment of SDD Schema with Instances

two or more $\langle \text{CategoricalCharacters} \rangle$.

At this stage it is important to define which SDD primitive would better correspond to the character / character-state (C,CS) formalism. Pimentel and Riggins [41] defined character as “a feature of organisms that can be evaluated as a variable with two or more mutually exclusive and ordered states”. The SDD *States* primitive, as shown in Figure 3.1, fits in the character definition and the SDD *StateDefinition* primitive fits in the character-state.

3.2.3 Phylogenetic Trees and the 3ia Method

The present paper also draws upon phylogenetic trees generated from LisBeth (<http://lis-upmc.snv.jussieu.fr/lis/>). LisBeth is a cladistic software for phylogenetics and biogeography [5] that implements the three-item analysis (3ia) method of phylogenetic inference [37], mentioned in the previous subsection. It minimizes the conflictual relationships within a set of characters, or maximizes the compatible relationships so as to reconstruct one or several optimal tree(s). The LisBeth model for the representation of characters is a hierarchy: characters are represented as rooted trees. Grand et al. [25] compared hierarchical characters with ordered and unordered character-states and showed, with simulated data, that the hierarchical treatment of characters results in the greatest resolving power (i.e. amount of correct relationships within the phylogenetic tree). Their results did not show, for most of them, statistically significant differences considering the

artefactual resolution rate (i.e. amount of incorrect relationships).

Williams and Ebach [52] pointed out that the data matrix have limitations to represent the relationship among character-states accordingly. Williams and Ebach [52] mentioned the BPA method (Brooks Parsimony Analysis [11, 12, 13]) that promotes a different structuring of data within a matrix, in order to represent branchings. Each column of a BPA matrix is a node, each row is a taxon. The values entered in the cell are the absence/presence of a taxon at a given node. However, Ebach et al. [20] showed that incorrect relationships can be reconstructed from a BPA matrix that is given as input for phylogenetic programs. As a consequence, a BPA matrix is not optimal to represent relationships among character-states. Other researchers order their character-states and represent the relationship among character-states as unrooted trees or as cyclic graphs using step matrices. In every case, the character-states are entered in a matrix. The three-item analysis (3ia) is the only phylogenetic method that does not rely on a matrix representation. Characters are represented directly as rooted trees, i.e. the relationship among character-states is the inclusion. This representation presents the advantage to distinguish missing data from inapplicable data [53], since the taxa with missing data are truly absent from the characters. This distinction is not implemented in phylogenetic programs that rely on matrices, since every matrix cell has to be filled and programs do not distinguish between the symbols “?” (missing) and “NA” (non applicable). Zaragüeta et al. [14] suggested to enhance a character matrix with an additional row, providing the hierarchy of states for every character column. Such enhanced matrices can be imported in the 3ia program LisBeth.

The present work follows the same approach proposed in [52] and forsake the use of data matrices, since the hierarchical representation has proven to be relevant [25, 53]. As we will detail in Section 3.4, this hierarchical representation is not only aligned with our unifying graph model, but also takes advantage of the richer relations compared to matrices.

3.3 Three Layer Method and System Architecture

Figure 3.2 illustrates our three layer method, which is the major goal of this project. In this method, a graph database mediates the evolution process from fragmentary data sources to accomplish full integration descriptions as ontologies. Our approach remodels sources from the dataspace to a graph representation, in which the data can be unified and linked, subsidizing the discovery of latent knowledge, which raises from the relations. The graph model was designed to be published on the Web in a Linked Data approach. Graph transformations will be applied for the transition from representations in the dataspace to a more formalized representation through ontologies. This work focuses in the graph

representation as an intermediary step towards ontologies and its application to support an analytical tool to compare data across studies.

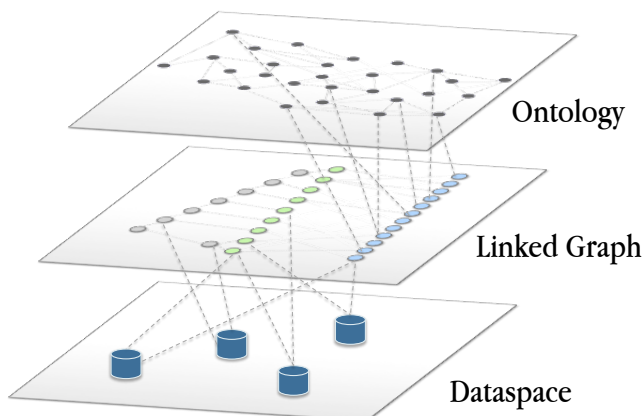


Figure 3.2: Three layer method diagram

Figure 3.3 summarizes the general architecture of our system. From a set of heterogeneous data sources (1), we ingest and transform data in a graph (2) stored in a graph database (3). In this stage of the research, we are interested in phenotype descriptions and phylogenetic trees, but the architecture was designed to afford smooth future extensions to other kinds of biological data. In step (3) each data source will result in a distinct graph. We apply LSIDs to unify Operational Taxonomic Units (OTUs) in the graph referring to the same real world object (4). LSID is a persistent, location-independent resource identifier, whose purpose is to uniquely identify biological resources [16]. In order to find a valid LSID, we adopted the Global Names Resolver (GNR) web service (<http://resolver.globalnames.org/>) that executes exact or fuzzy matching against canonical forms of scientific names in 170 distinct data sources. In step (4), we are developing algorithms to discover relations and find similarities among nodes in the graph, which are made explicit by adding new nodes in the graph. This step is detailed in Section 3.5. The resulting graph can be locally analyzed by a researcher; can be published on the Web in a Linked Data approach to be remotely exploited (6); and will subsidize the expansion and enrichment of ontologies in the future (7).

3.4 Unified Graph data model

In this section we will present an overview of our proposed graph model. A significant factor to consider is determining which graph data model should be used. Angles [3] pointed out that a graph data model provides an abstraction layer, which is more modeling

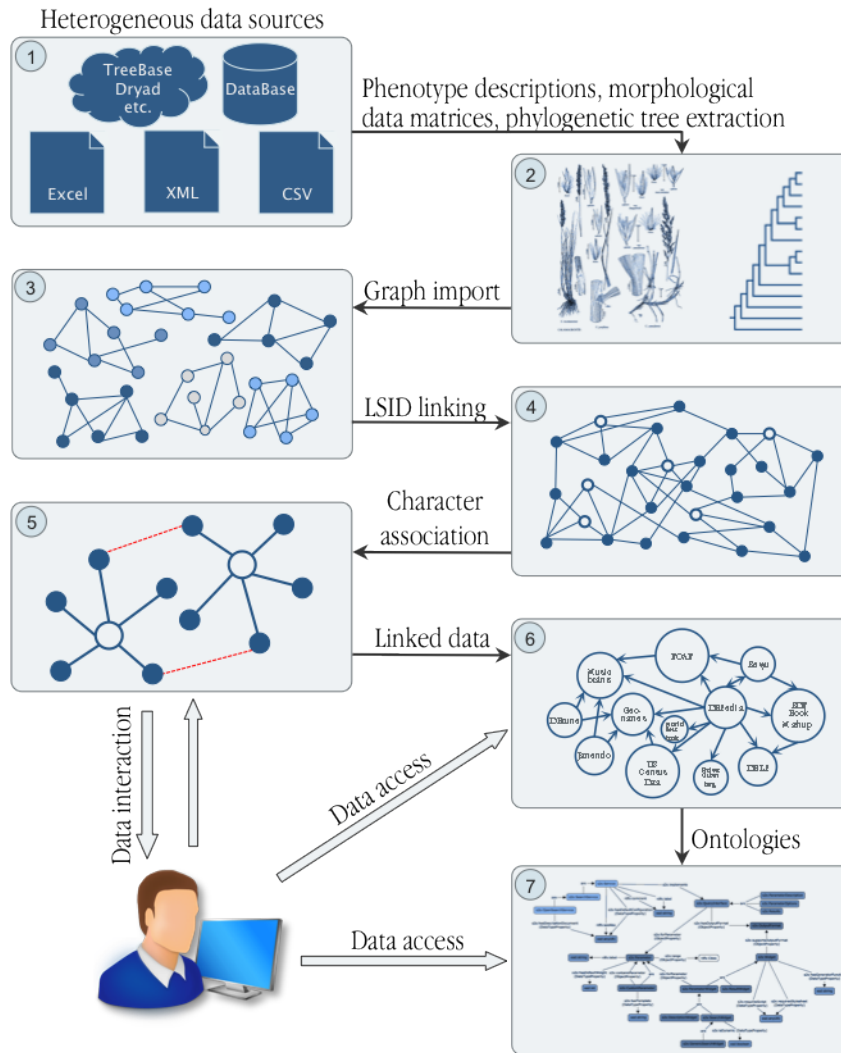


Figure 3.3: General System Architecture.

intuitive, as well as its query languages and algorithms to store, retrieve and manipulate data. Real world data can usually be represented in terms of nodes and edges only. The ease of modeling has a collateral effect on the interpretation, since sometimes it will not be easy to visualize and understand data in such a simple model [4]. From the numerous graph data models proposed – see [3, 4, 44] for more details – the *property graph* model was adopted in the present work. In a *property graph*, nodes and relationships can maintain extra metadata as a set of key/value pairs. Moreover, relationships are typed, enabling to create multi-relational networks with heterogeneous sets of edges. Different from single-relational networks, in which edges are of the same type, multi-relational networks are more appropriate to represent complex domain models, due to the variety

of relationship types in the same graph [45]. For example: relationships may either represent membership in a social group (family membership) or professional relationships (employer-worker relationship) simultaneously in the same network.

Figure 3.4 shows our graph data model. The tables below the nodes/edges represent their types and metadata. We mapped the SDD format to the graph model as follows: OTUs are entities (e.g., “*Varanus prasinus*”) and, therefore, were mapped to nodes. A future target of this project is to enrich our model by associating identifiable entities to ontology concepts. One may consider to map Characters and Characters-States to key/value pairs, to be related to OTU nodes. However, we decided to map Characters to nodes, in order to unify in the same node equivalent characters observed in several OTUs and, in a future work, to relate the unified characters with ontologies. Finally, the Character-state makes a semantic bridge (relationship) between OTUs and Characters. Thus, a statement like “*Varanus gouldi ventral pattern is randomly scattered dark spots*” is represented in our model as *Varanus gouldi* (node) \rightarrow *randomly scattered dark spots* (edge) \rightarrow *ventral pattern* (node). This part of the model, is a common denominator of phenotype descriptions, conceived in our previous work [34].

Our model comprises, in a single place, phenotype descriptions and phylogenetic trees. For this reason a new node called HTU (Hypothetical Taxonomic Unit) is present in it. HTUs are internal nodes in phylogenetic trees that represent an inferred ancestral organism. HTUs are hypothetical common ancestors of OTU nodes and, therefore, can only be connected to themselves (HTU \rightarrow HTU) or to OTUs (HTU \rightarrow OTU). For the sake of modeling simplicity, only the *TreeEdge* relationship is allowed between HTU \rightarrow HTU and HTU \rightarrow OTU.

3.5 Link Discovery

In order to illustrate the possibilities raised by the unification and linking of data of phenotype descriptions with phylogenetic trees, we present a practical implementation executed in our system, which involves the linking discovery among characters. Most of existing literature related to morphological character description is expressed in textual form, which are sometimes not consistent among authors. In general, researchers tend to reuse characters already published in the literature, in large part to make their descriptions comparable with other taxa. However, textual descriptions convey little semantics of the character, which prevents the correct understanding of authors’ meaning. The analysis of graphs combining phenotype and phylogenetic data enable us to discover links among characters and their respective states even when they are part of descriptions developed independently.

In order to illustrate our analysis, consider a practical scenario – illustrated in Figure

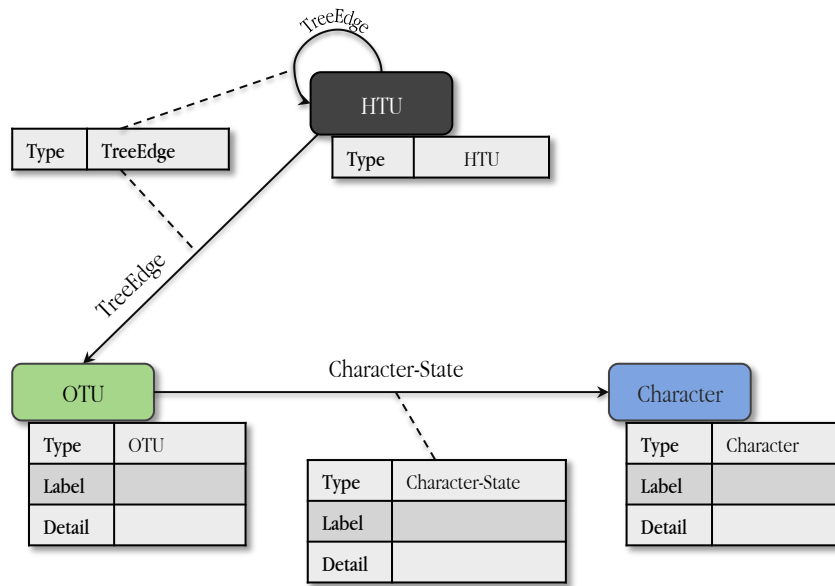


Figure 3.4: Property Graph Model

3.5 as Scenario 1 – where two authors are working in different projects, but both are describing extinct ferns. One author describes *Pseudosporochnus* in terms of “*Extent of the planated parts within the LBS*” (Lateral Branching System, see [17]) (character) and coded it as “*restricted to the extremities*” (character-state). In a different study another author is analyzing a *Pseudosporochnus* fossil as well, and describes it in terms of “*Planation*”(character) intending to mean “*Extent of the planated parts within the LBS*” and also coded it as “*restricted to the extremities*” (character-state). In this scenario, two characters with different labels have the same meaning. The opposite scenario – illustrated in Figure 3.5 as Scenario 2 – might happen as well, in which two or more characters with the same label (character) have different meanings. For example, one author uses the character “*Planation of vegetative leaves*” and consider that *Pseudosporochnus* fossil has no leaf and coded it as “*inapplicable*” (character-state). Another author uses the same character “*Planation of vegetative leaves*” meaning “*Extent of the planated parts within the LBS*” and consider that *Pseudosporochnus* planation is “*restricted to the extremities*”.

From the integration point-of-view, a machine will interpret the descriptions of Scenario 1 as distinct, even though the two characters (that are different statements) have the same meaning. In Scenario 2, two textual characters that are identical, and could be interpreted in a string matching as equivalent, do not have the same meaning. Figure 3.5 shows the scenario presented above modeled in our proposed graph model.

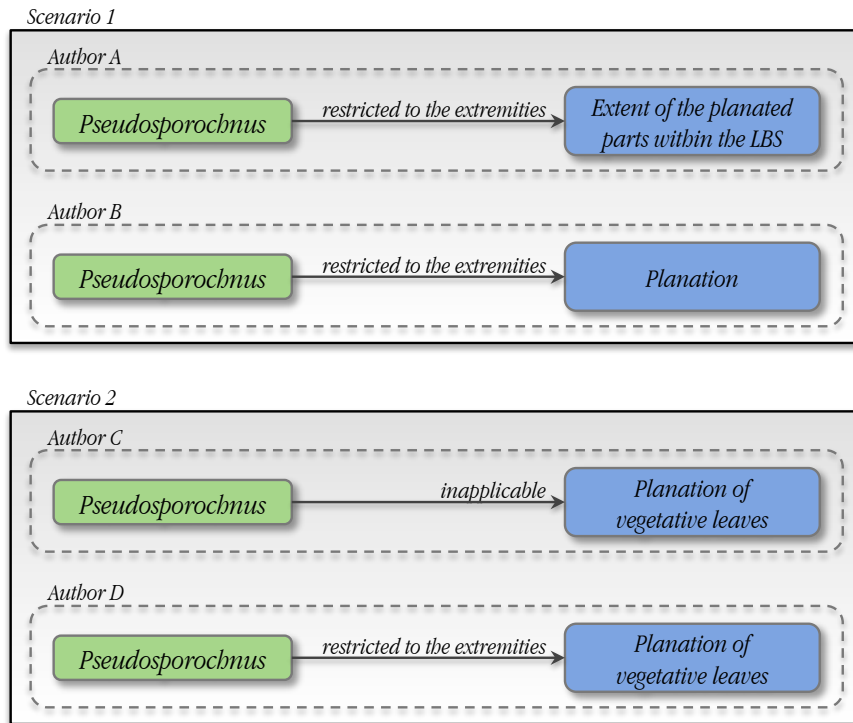


Figure 3.5: Practical Scenario

3.5.1 Similarity Index

In this respect, we are proposing a heuristic similarity measure that computes the similarity degree between two morphological character descriptions, which will represent how closely related they are. The similarity index (\mathcal{S}_i) is based on 2 weighted aspects. 25% of the index is calculated based on the taxa being described, i.e. it analyzes if two given characters (C_1 and C_2) describe the same taxa. The other 75% are based on the meaning of the character-states. This part checks if the state labels being used are the same. This heuristic is still a work in progress. The weights assigned to parts of the index are configurable and their values were calibrated based on observations.

Let $G = (V(G), E(G))$ be a directed graph with vertex-set $V(G) = \{v_1, \dots, v_n\}$ and edge-set $E(G) = \{e_1, \dots, e_m\} \subset \{(v_i, v_j) | v_i, v_j \in V(G)\}$. Let $C_1, C_2 \in V(G)$ be two distinct vertices of G . We define the following sets:

$$N_{C_1} = \{v_i \in V(G) \mid (v_i, C_1) \in E(G)\} \quad (3.1)$$

$$N_{C_2} = \{v_i \in V(G) \mid (v_i, C_2) \in E(G)\} \quad (3.2)$$

$$\mathcal{S}_1 = \frac{|N_{C_1} \cap N_{C_2}|}{\max\{|N_{C_1}|, |N_{C_2}|\}} \quad (3.3)$$

Let $f : E(G) \rightarrow \Upsilon$ be a labeling function, where Υ is a set of labels, and $f(e) \in \Upsilon$ is the label of edge $e \in E(G)$. We define the following sets:

$$L_{C_1} = \{e \mid e = f((v_i, C_1)) \in \Upsilon \text{ and } (v_i, C_1) \in E(G) \text{ and } v_i \in V(G)\} \quad (3.4)$$

$$L_{C_2} = \{e \mid e = f((v_i, C_2)) \in \Upsilon \text{ and } (v_i, C_2) \in E(G) \text{ and } v_i \in V(G)\} \quad (3.5)$$

$$S_2 = \frac{|L_{C_1} \cap L_{C_2}|}{\max\{|L_{C_1}|, |L_{C_2}|\}} \quad (3.6)$$

$$\textit{Similarity Index}(S_i) = 0.25 * S_1 + 0.75 * S_2 \quad (3.7)$$

S_1 defines a rate of common OTU vertexes with edges for two given characters C_1 and C_2 . The S_1 result lies between 0 (no common OTUs) and 1 (all OTUs are common). N_{C_1} is the subset of incoming adjacent vertexes of C_1 and N_{C_2} is the subset of incoming adjacent vertexes of C_2 . Incoming adjacent vertexes of both C_1 and C_2 are always OTU vertexes, as shown in Figure 3.4. S_2 defines a rate of common labels of the incoming edges (character-states) for the characters C_1 and C_2 . The S_2 result also lies between 0 (no common character-states) and 1 (all character states are common). L_{C_1} and L_{C_2} are the subset of incoming adjacent edge labels (character-states) of C_1 and C_2 respectively.

It is important to note that the character labels of C_1 and C_2 are not being taken into account in the S_i formula. This intends to avoid weighting in favor of two identical textual characters that do not have the same meaning, and to avoid weighting against two textual characters that are identical but do not have the same meaning. In practice, this will make the solution independent of the label and applicable for both presented scenarios (same label but different meanings and different labels and same meaning). Additionally, the symmetric property of equality is satisfied.

3.5.2 Practical Implementation of the Similarity Measure

In order to present our Similarity Index (S_i) working on top of the proposed graph data model (see Section 3.4), consider two studies of distinct authors. Author 1 worked with the fossils: *Denlongia*, *Equisetum*, *Pseudosporochnus*, *Archeopteris*, *Ibyka*, *Iridopteris*, *Ophioglossum* and *Polypodium*, describing them in terms of “*Cauline cladotaxy*”, “*Protoxylem position within the cauline stele*”, “*Development of the LBS*”, “*Organotaxy of the LBS*”, “*Presence of planated parts within the LBS*”, “*Extent of the planation*”, “*Xylem configuration in the rachis*”, “*Xylem configuration in the leaflets*” and “*Branchiness*”. Author 2 described three fossils also analyzed by author 1: *Equisetum*, *Ophioglossum* and *Polypodium*; plus described the fossils *Marattia*, *Botryopteris*, *Psaliyochlaena* and *Cyathea*. Author 2 described the fossils adopting four terms for characters equivalent to author 1: “*Cauline cladotaxy*”, “*Protoxylem position within the cauline stele*”, “*Xylem*

configuration in the rachis”, “*Xylem configuration in the leaflets*”; plus other character terms: “*Development of the foliar organ*”, “*Phyllotaxy*”, “*Planation*” and “*Branchiness of the leaf*”.

All of these data were managed in the Xper² tool, first exported to the SDD format and then imported to the graph. A software script was designed to calculate the S_i for all characters taken two by two. Figure 3.6 shows a screenshot of a visual tool still under development that creates an edge between each 2 characters with S_i greater or equal than 0.5. This is a simple but powerful visualization tool to present the similarity measure that could play a pivotal role in supporting biologists to understand and detect correlation between characters. Indeed, Figure 3.6 shows a graph clique among the characters “*Cauline cladotaxy*”, “*Organotaxy of the LBS*” and “*Phyllotaxy*”. All three characters refer to the insertion mode of an organ on a bearer structure: “*Cauline cladotaxy*” means the insertion mode of the stem ramifications on the main stem, “*Organotaxy of the LBS*” means the insertion mode of the LBS on the main axis, and “*Phyllotaxy*” means the insertion mode of the leaf on the main stem. All three characters share the same set of character-states, and even if they do not refer to the same entities (i.e. stem *versus* LBS *versus* leaf) and cannot be substituted for one another, they share a part of their meaning. Knowing that they are similar to some extent can encourage the biologist to suggest identical relationships among character-states for the foliar (or LBS) character and the cauline character, for the sake of consistency.

3.6 Conclusion

Linking together descriptive data around the dynamic Tree of Life model is a complex task because, although there are a lot of data available, these data are represented in many standards not often interconnectable. In this respect, the present paper explores this problem linking and coupling phylogenetic trees and phenotype descriptions through a graph database model. Our unified model enabled us to discover and make explicit the potential semantics raised by linking previously unconnected information. Our representation in a graph database is aligned with a RDF graph-based representation, which will be the next step to achieve the third layer. The challenge will be to map labels of character/character-states in RDF properties/values. The unification of characters and states, as shown on this preliminary work, is a first and high relevant step for this mapping. Since several ontologies related to phenotype descriptions are in OWL, the relations discovered in our graph can subsidize a better matching of labels and concepts in OWL ontologies by confronting relations. For example, to enhance the match of a character label (in the graph database) with an OWL property (a character being an OWL property), it is possible to consider the states allowed by the character, confronting them with

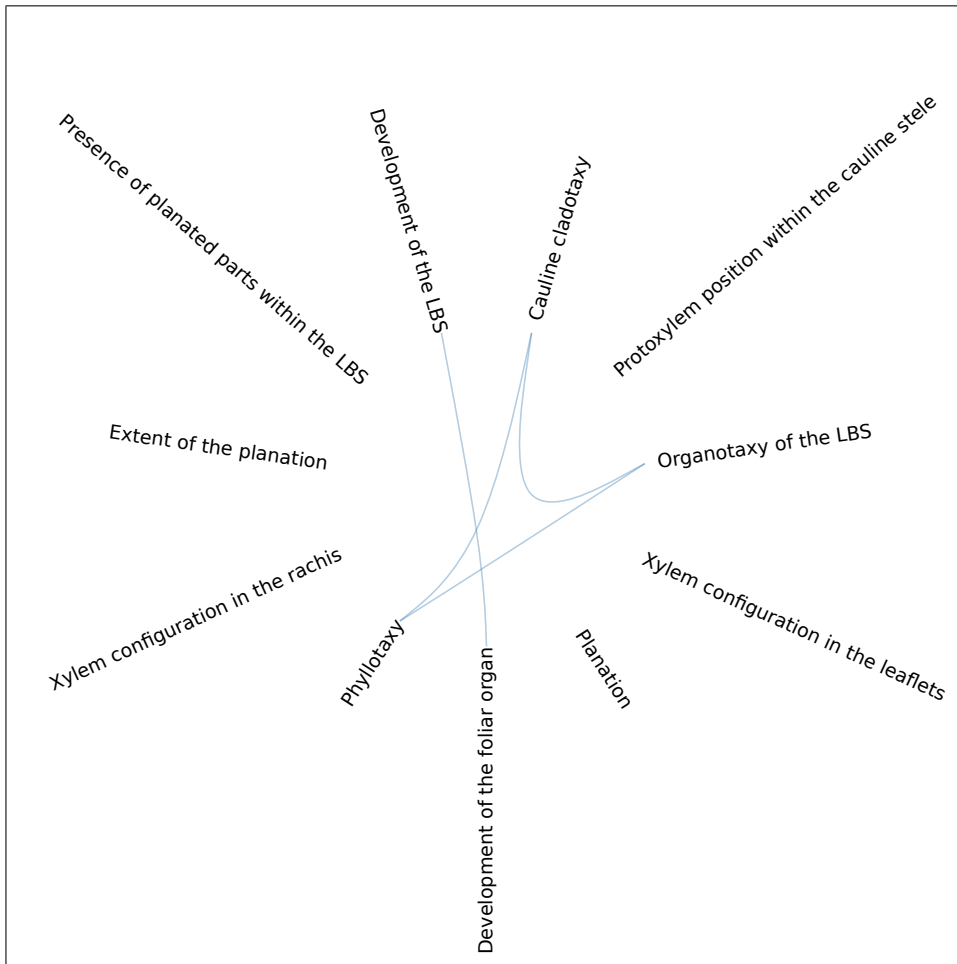


Figure 3.6: Practical Implementation

the property range (values allowed by the property).

The viability and the potential of our approach were tested by practical implementations in which 2 distinct author descriptions of fossils were inserted into a graph database and analyzed by the similarity measure method proposed in this paper. A visual tool to visualize how close related any two given characters are is being developed and some preliminary results are presented. This tool has the potential to demonstrate how the same characters recur in different studies and might support biologists to understand and detect correlation between characters. Having the capability of integrating knowledge around taxonomic units will enable, for instance, evolutionary biologists to generate new research questions, gain predictive insight or confront evolutionary hypotheses. Further developments will take into account the similarity among hierarchies of states for the character comparisons.

Chapter 4

Linked biology technical aspects – linking phenotypes and phylogenetic trees

4.1 Introduction

In 1859 Charles Darwin published *On the Origin of Species* which is considered the foundation of evolutionary biology. In his book, Darwin set forth the theory of evolution and natural selection. It argues that all life is related and has descended from a common ancestor. The Tree of Life is a metaphor to describe the relationships between living and extinct organisms through their common ancestors. More precisely, it is an abstract form to represent hypotheses about evolutionary relationships, in which all species that have ever existed are taken together with relationships among them, describing their evolutionary lineages. In this abstract representation, the taxa are the leaves of the tree and the internal nodes are common ancestors, or hypothetical taxa.

This huge and complex tree is split into smaller branches, which are investigated separately and then incorporated into the tree. Evolutionary biologists normally work in relatively small chunks of the tree, analyzing a very specific subset of species. A fundamental challenge in this scenario is the creation of a complete evolutionary Tree of Life [39], assembling genomic and morphological data so as to congregate the phylogenetic relationships among all known living or extinct organisms [15, 19, 33]. The integration of these data may contribute to better understand how a morphological trait became organized and evolved over time [29], how organisms interact and how life on Earth came to be.

The main goal of this research is to design and implement a linked biology approach to automatically connect and combine data from independent semi-structured resources of

phenotype descriptions and/or phylogenetic trees, exploiting their latent semantics. We propose a graph data model that plays a crucial role, since it is the basis of our linking discovery and combination process. It contributes assisting biologists in the exploration of existing biology assets related to phenotype descriptions and their latent semantics. The present work details algorithms, implementation aspects and the database model related to our research.

The text is organized as follows. Section 4.2 synthesizes basic concepts necessary for understanding the text. Section 4.3 discusses implementation details of our system and presents some results. Section 4.4 presents concluding remarks. In the Appendix the source code is provided with comments explaining its functionalities.

4.2 Basic concepts

In this section, we highlight basic concepts adopted in this text. Subsection 4.2.1 introduces some key elements of XML formats for phenotype description. Subsection 4.2.2 we details the Life Science Identifier which is one of the solutions for data interconnection. Subsection 4.2.3 presents an overview of our proposed graph model.

4.2.1 Standards for Phenotype Description

There is a wide variety of representation formats for phenotype descriptions adopted by information systems and open standards, which represent differently the same information. In [34] we analyze four of them – Xper², SDD, Nexus and NeXML – looking for a common denominator which is the foundation for our graph-based model. SDD, Nexus and NeXML are widely adopted open standards. Xper² (<http://lis-upmc.snv.jussieu.fr/lis/>) is a management system adopted by the systematist community, for storing, editing and analyzing phenotype descriptive data. It focuses mainly on taxonomic descriptions, allowing creation, sharing and comparison of identification keys [47, 48]. Xper² was developed in the Laboratoire Informatique & Systématique of the University Pierre et Marie Curie and this work is part of a bigger project in collaboration with this lab. Therefore, Xper² was adopted for our practical implementation.

In order to transform phenotype observations into digital records and generalize them – e.g., devising general characters and states observed in a genus of monitor lizards – the biologist may use a tool as Xper². Phenotype descriptions can be stored in the Xper² native format or can be exported to the SDD open format. The Structured Descriptive Data (SDD) (<http://wiki.tdwg.org/SDD>) is a platform and application-independent XML-based standard developed by the Biodiversity Information Standards (historic acronym: TDWG) for recording and exchanging descriptions of biological and biodiversity data of

any type [26]. SDD is adopted by several other phenotype description tools – e.g., Lucid Central (<http://www.lucidcentral.org>) and Linnaeus II (<http://www.eti.uva.nl/>).

We further introduce some key elements of the SDD format, which are recurrent in the formats confronted in [34]. A SDD description comprises, in a single file, a domain schema and its instances. Figure 4.1 shows a diagram with a fragment of a SDD file containing the description of a varanus lizard. A (C,CS) description in SDD has two main blocks: (i) defines the characters involved and their possible states – Figure 4.1 top; (ii) describes an Operational Taxonomic Unit (OTU) using the characters defined in (i) – Figure 4.1 bottom. OTU is a biology term which refers to a given taxon at the rank adopted to the study – e.g., a specimen, a species, a genus etc.

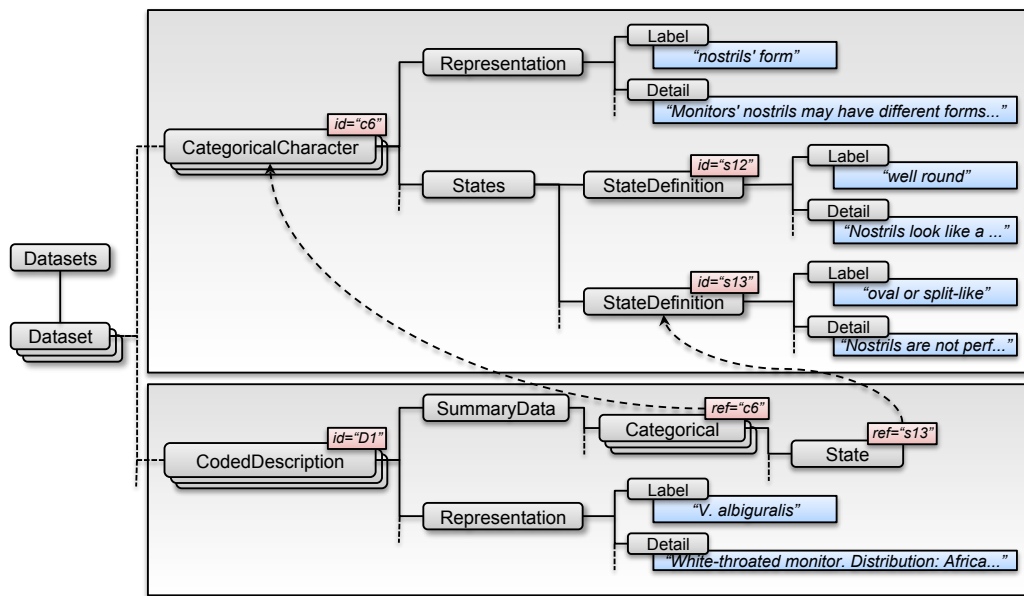


Figure 4.1: Fragment of SDD Schema with Instances

*<CategoricalCharacter>*s and their *<States>* (shown in Figure 4.1 top) are primitives to describe an OTU [26]. Each *<CategoricalCharacter>* has its *<Representation>* – comprising a label and a description as plain texts – and a set of *<StateDefinition>* elements with their possible states. *<CategoricalCharacter>* and *<StateDefinition>* elements defined here will be referred throughout the XML document by their ids. The *<CodedDescription>* (Figure 4.1 bottom) links the described OTU to *States* of each *<CategoricalCharacter>*. It has two essential items: (i) the described OTU, where its name and description are listed in natural language under *<Representation>*; (ii) a set of character and values (*<Categorical>* and *<State>*), which address the characters defined in the previous section through the *ref* attribute. It is possible and usual to assign multiple character-states for a given OTU (i.e. in case of polymorphism). A first integration,

problem observed here is that each character or OTU described does not have a global unique identification among documents. Therefore, the description can only be used by the document where it was declared and it is not possible to guarantee the equivalence of two or more *<CategoricalCharacters>*.

4.2.2 Life Science Identifiers (LSIDs)

One of the problems faced in life science is related to the identification of objects within and across repositories [38]. More precisely, an object may refer to a taxon, gene, anatomical feature, phenotypic description, geographical location etc. Integrating data from different sources is not straightforward and uniquely identifying these objects is undoubtedly a key point for the success of our proposed solution.

During the 18th century, Carolus Linnaeus introduced the binomial nomenclature for naming species that is the basis of modern classification [24]. This system basically concatenates 2 Latin words, where the first part identifies the species genera and the second one the species itself. The binomial nomenclature has been used for the last 250 years [24] and the biological information related to organisms is historically annotated by species names. Hence, the binomial name would appear to be a logical candidate to index information available about species. However, misspelling problems are often encountered [1, 43], moreover, taxonomic names are not unique identifiers [27, 40] because scientists may use (i) similar names to different species (homonyms) or (ii) multiple names for the same specie (synonyms) [38, 9].

Furthermore, each organization has its own means of defining a key, which makes the problem even harder to solve. For example, the species *Aotus ericoides* has the id 11479744 on the Catalogue of Life (CoL), id 42472 on the Australian Plant Name Index (APN), id 643314 on the Encyclopedia of Life (EoL), id 129761-3 on the The International Plant Names Index (IPNI), id 700844 on the Universal Biological Indexer and Organizer (uBio) etc.

In order to address this issue, some organizations – e.g., Universal Biological Indexer and Organizer (uBio), Integrated Taxonomic Information System (ITIS), Catalogue of Life (CoL), The International Plant Names Index (IPNI), National Center for Biotechnology Information (NCBI) etc. – incorporated into their projects the concept of Life Science Identifiers (LSIDs), proposed by the Object Management Group (OMG) (<http://www.omg.org/>). LSID is a persistent, location-independent resource identifier, whose purpose is to uniquely identify biological resources [16]. The persistent property refers to the fact that LSID identifiers are unique, can be assigned to only one object forever and they never expire. The location-independent property specifies that each authority locally creates LSIDs and they are the responsible to guaranteeing the uniqueness

of LSIDs.

4.2.3 The proposed graph data model

In this section we will present an overview of our proposed graph model. From the numerous graph data models proposed – see [3, 4, 44] for more details – the *property graph* model was adopted in the present work. In a *property graph*, nodes and relationships can maintain extra metadata as a set of key/value pairs. Moreover, relationships are typed, enabling to create multi-relational networks with heterogeneous sets of edges. Different from single-relational networks, in which edges are of the same type, multi-relational networks are more appropriate to represent complex domain models, due to the variety of relationship types in the same graph [45]. For example: relationships may either represent membership in a social group (family membership) or professional relationships (employer-worker relationship) simultaneously in the same network.

Figure 4.2 shows our graph data model. The tables below the nodes/edges represent their types and metadata. We mapped the SDD format to the graph model as follows: OTUs are entities (e.g., “*Varanus prasinus*”) and, therefore, were mapped to nodes. A future target of this project is to enrich our model by associating identifiable entities to ontology concepts. One may consider to map Characters and Characters States to key/value pairs, to be related to OTU nodes. However, we decided to map Characters to nodes, in order to unify in the same node equivalent characters observed in several OTUs and, in a future work, to relate the unified characters with ontologies. Finally, the Character-state makes a semantic bridge (relationship) between OTUs and Characters. Thus, a statement like “*Varanus gouldi ventral pattern is randomly scattered dark spots*” is represented in our model as *Varanus gouldi* (node) → *randomly scattered dark spots* (edge) → *ventral pattern* (node).

Our model comprises, in a single place, phenotype descriptions and phylogenetic trees. For this reason a new node called HTU (Hypothetical Taxonomic Unit) is present in this model. HTUs are internal nodes in phylogenetic trees that represent an inferred ancestral organism. HTUs are hypothetical common ancestors of OTUs nodes and, therefore, can only be connected to themselves (HTU → HTU) or to OTUs (HTU → OTU). For the sake of modeling simplicity, only the *TreeEdge* relationship is allowed between HTU → HTU and HTU → OTU. Finally, there is also a character-state relationship between HTU nodes and character nodes that are strictly created by some algorithms.

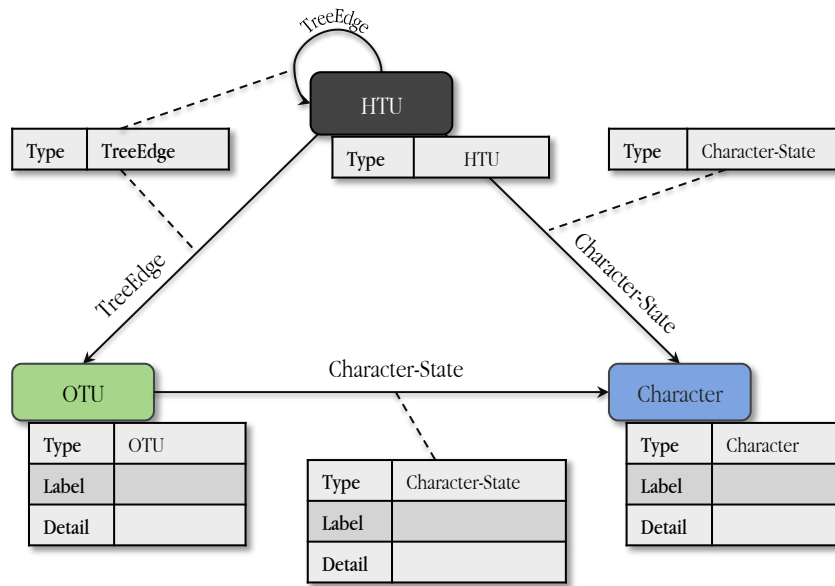


Figure 4.2: Property Graph Model

4.3 System Architecture and Implementation Details

In this section, we analyze the system architecture and its implementation details, in order to present its main functionalities and operational features. The text is presented progressively. The core functionalities are shown in the first subsections and the algorithms are presented later.

We have developed our platform on top of the Neo4j graph database (<http://www.neo4j.org/>), mainly due to its widespread adoption. Our implementation uses the Python programming language and Py2neo (<http://book.py2neo.org/>), which is an interface connecting Python and Neo4j via REST API. The adopted query language was *Cypher*, which is a declarative graph query language.

4.3.1 SDD Parser

Our SDD Parser has all functionalities to parse an SDD file (for implementation details see Appendix A.1) using the Python *xml.dom.minidom*, which is a minimal implementation of the Document Object Model interface. Listing 4.1 shows an SDD fragment of a Varanus knowledge base ¹, of which Figure 4.1 is a simplified abstraction. In addition, all main SDD structures presented in Figure 4.1 and Listing 4.1 – Representation, StateDefinition,

¹Knowledge base of the genus Varanus
(http://lis-upmc.snv.jussieu.fr/xper2/infosXper2Bases/details_base.php?id_base=86)

CategoricalCharacter, Categorical and CodedDescription – were processed to produce our graph.

Listing 4.1: Varanus.sdd.xml

```

1 <Characters>
2   ...
3   <CategoricalCharacter id="c6">
4     <Representation>
5       <Label>nostrils ' form</Label>
6       <Detail>Monitors' nostrils mayhave different forms.&lt;br&
          gt;Look at the head in side view or dorsal view in
          order to appreciate this characteristic.</Detail>
7       <MediaObject ref="m40"/>
8     </Representation>
9     <States>
10      <StateDefinition id="s12">
11        <Representation>
12          <Label>well round</Label>
13          <Detail>Nostrils look like a quite perfect circle.</
            Detail>
14        </Representation>
15      </StateDefinition>
16      <StateDefinition id="s13">
17        <Representation>
18          <Label>oval or split-like</Label>
19          <Detail>Nostrils are not perfectly round: they are oval
            or they present a split-like form.</Detail>
20        </Representation>
21      </StateDefinition>
22    </States>
23  </CategoricalCharacter>
24  ...
25 </Characters>
26 ...
27 <CodedDescriptions>
28 <CodedDescription id="D1">
29 <Representation>
30 <Label>V. albiguralis</Label>
31 <Detail>White-throated monitor&lt;br&gt;&lt;br&gt;

```

```

    Distribution: Africa (West and South).&lt;br&gt;&lt;br&gt;
    CITES: appendix II.</Detail>
32 <MediaObject ref="m1"/>
33 </Representation>
34 <SummaryData>
35     ...
36     <Categorical ref="c6">
37         <State ref="s13"/>
38     </Categorical>
39     ...
40 </SummaryData>
41 </CodedDescription>
42 </CodedDescriptions>

```

4.3.2 Tree Output

The present work also draws upon phylogenetic trees generated from LisBeth (<http://lis-upmc.snv.jussieu.fr/lis/>). LisBeth is a cladistics software for phylogenetics and biogeography [5] that implements the three-item analysis (3ia) method of phylogenetic inference [37]. It minimizes the conflictual relationships within a set of characters, or maximizes the compatible relationships so as to reconstruct one or several optimal tree(s). We implemented a *TreeOutput* class, which abstracts the functions of interacting with LisBeth output files (for implementation details see Appendix A.2). Listing 4.2 displays two fragments of a LisBeth output file, focusing in the elements processed in this work, i.e. taxons with their ids and the retained tree – newick tree which is a way to represent a tree in computer-readable form, using parentheses and commas. The *TreeOutput* main function combines the retained tree with the taxon names, retrieved in previous steps, and returns a root node to a tree that represents the retained tree. In this new tree, the internal nodes are renamed to *HTU* and the leaf nodes to its respective taxon names (see Figure 4.3).

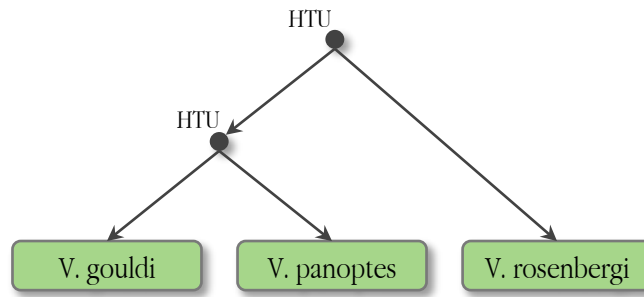


Figure 4.3: Retained Tree Example

Listing 4.2: LisBethOutput.3iz

```

1 ...
2 <D02>
3 ...
4 Taxa (3) :
5 .           3 V. gouldii
6 .           7 V. panoptes
7 .          12 V. rosenbergi
8 <F02>
9 ...
10 <D06>
11 ...
12
13 Retained trees : 1
14 .   1: ((3 7) 12)
15 <F06>
16 ...

```

4.3.3 Global Names Resolver (GNR)

In order to find a valid LSID, we adopted the Global Names Resolver (GNR) web service (<http://resolver.globalnames.org/>) that executes exact or fuzzy matching against canonical forms of scientific names in 170 distinct data sources. The Canonical form (cf) is the simplest, most complete and unambiguous form of a name. The Canonical form of scientific names consists of the genus and species – when applied – with no authorship, rank, nomenclatural annotation or subgenus.

Our system used three of the six types of matching offered by the GNR resolver: (i) exact matching; (ii) exact matching of canonical forms – this process reduces a given name to its canonical form and checks it for an exact match; (iii) fuzzy matching of canonical forms – uses a modified version of the TaxaMatch algorithm [43] and intends to work around misspellings errors. It does a fuzzy match of the canonical form of a given name – even with mistakes – against spellings considered correct. The GNR resolver reports the matching quality (“*confidence score*”) for each match. The other three remaining matching types are: (iv) exact matching of specific parts of names, (v) fuzzy matching of specific parts of names and (vi) exact matching of genus part of names. They were not adopted because we focused in complete names in their canonical form.

Our algorithm extracts all plain text taxon entities present in the SDD file and, for each one, it uses the GNR to transform the taxon name to its canonical form. Only those taxons with confidence score above of 0.988 are considered. After that, the algorithm makes use of the GNR resolver to search for its LSID (for implementation details see Appendix A.3) – only exact matches are considered. The GNR results have the output field “*local id*” which, in the case of uBio, is the LSID. Moreover, we prioritized the uBio LSID, since it indexes and organizes until now more than 11 million names. But there are cases in which the GNR resolver does not retrieve any result from the uBio. In these cases, the algorithm makes use of the Integrated Taxonomic Information System (ITIS) web services (<http://www.usgovxml.com/DataService.aspx?ds=ITIS>), in order to obtain the LSID (for implementation details see Appendix A.5). ITIS is a reliable taxonomic base for species, with more than 740 thousand common names and scientific names indexed. If none of the services return a valid LSID, we also implemented a class to interact with the CoL web service (<http://www.catalogueoflife.org/col/webservice>), attempting to obtain a valid LSID (for implementation details see Appendix A.6).

4.3.4 Graph Importer

Graph Importer is an object class written in Python that is responsible for coupling the phylogenetic trees and phenotype descriptions into the graph database. The insertion process follows the sequence: (1) Starts parsing the SDD XML file and the LisBeth output file – see Listing 4.1 and 4.2 respectively. (2) Creates a taxon node for each taxon present in the SDD file – see Figure 4.1 bottom, tag *<Representation>*. In this process, it searches for a valid LSID for each taxon node, using the GNR web service, ITIS web service or CoL web service. If the LSID is not found, it creates a taxon node without LSID. (3) Joins the taxon nodes to the tree structure, extracted from the LisBeth output file. (4) A node is created for each character in the SDD file – see Figure 4.1 top, tag *<Representation>*. (5) The taxon nodes are linked to the character nodes by their character-states – see Figure 4.1 top,

tag `<States>/<StateDefinition>`. It will exist character-state relationships where exists a pair `<SummaryData>/<Categorical>` and `<SummaryData>/<Categorical>/<State>` – see Figure 4.1 bottom, tag `<SummaryData>`. For implementation details see Appendix A.7. Figure 4.4 shows a visual representation of the retained tree combined with the taxon nodes provided in Listing 4.2. The figure shows that the edges depart from taxon nodes toward character nodes.

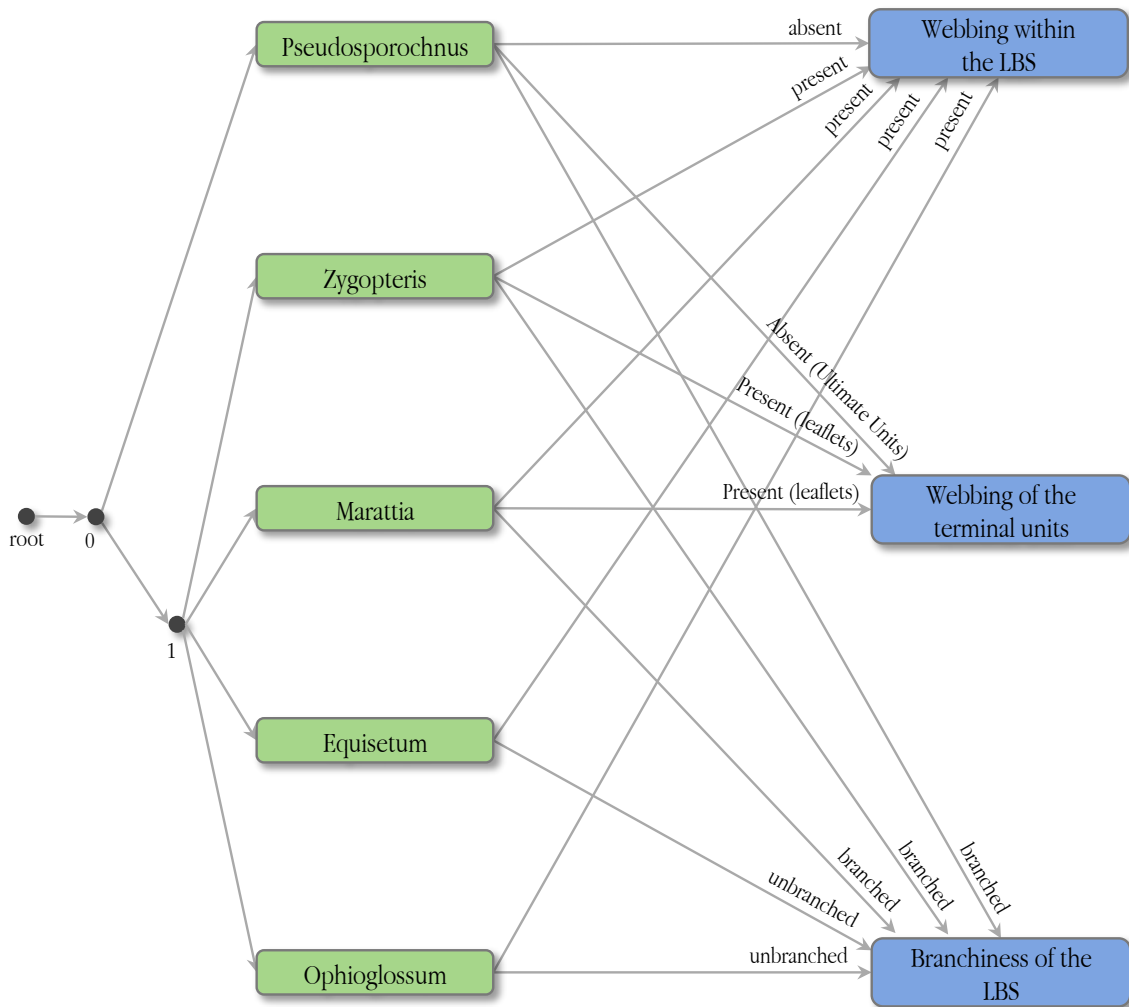


Figure 4.4: Real Example

4.3.5 Graph Database

We implemented a GraphDB class, which abstracts and centralizes all database operations. We describe each function header, followed by a short description of the main

Cypher queries used in the system.

```

1  getNodeByLSID( LSID ):
2  // Returns a node for the supplied LSID.
3  START n=node( * )
4  WHERE n.lsid = 'LSID'
5  RETURN n
6
7  getOutgoingAdjacentNodes( GivenNode ):
8  // Returns all nodes to which the given node points to.
9  START n=node( GivenNode.id )
10 MATCH (n)-->(c)
11 RETURN DISTINCT c
12
13 getIncomingAdjacentNodes( GivenNode ):
14 // Returns all nodes that points to the given node.
15 START n=node( GivenNode.id )
16 MATCH (c)-->(n)
17 RETURN DISTINCT c
18
19 getIncomingAdjacentRelationships( GivenNode ):
20 // Returns all relationships incoming to a given node.
21 START n=node( GivenNode.id )
22 MATCH ()-[r]->(n)
23 RETURN r
24
25 getIncomingAdjacentNodesWithRelationshipInBetween( GivenNode,
26     GivenRelationship ):
27 // Returns all nodes, ordered by their label, that points to
28     a given node with a given relationship in between.
29 START n=node( GivenNode.id )
30 MATCH (c)-[:GivenRelationship.label]->(n)
31 RETURN c
32 ORDER BY c.label
33
34 getOutgoingRelationships( GivenNode ):
35 // Returns all relationships outgoing from a given node.
36 START n=node( GivenNode.id )

```

```

35 MATCH (n)-[r]->()
36 RETURN r
37
38 getDistinctRelationshipsInBetween( GivenNodeA, GivenNodeB ):
39 // Returns all distinct relationships that exists between
   nodes A and B.
40 START a=node( GivenNodeA.id ), b=node( GivenNodeB.id )
41 MATCH (a)-[r]-(b)
42 WITH COLLECT( DISTINCT TYPE( r ) ) as rels
43 RETURN rels
44
45 getDescriptionNodesOfATree( TreeRoot )
46 // Returns all distinct description nodes id, character or
   character-states depending on the schema, that are
   conected to a given tree.
47 START root=node( TreeRoot.id )
48 MATCH (root)-[*..]->(d)
49 WHERE d.type = 'description'
50 RETURN DISTINCT ID(d)
51
52 deleteNodeRelationshipsExceptLabel( GivenNode,
   RelationshipLabel ):
53 // Deletes all node relationships except for a given
   relationship label.
54 START n=node( GivenNode.id )
55 MATCH n-[r]->()
56 WHERE NOT( r.label = 'RelationshipLabel' ) AND NOT( r.type =
   'TreeEdge' )
57 DELETE r
58
59 deleteRelationshipsTypeFromNode( GivenNode, RelationshipType
   ):
60 // Deletes all node relationships of a given type.
61 START n=node( GivenNode.id )
62 MATCH n-[r]->()
63 WHERE r.type = 'RelationshipType'
64 DELETE r

```

4.3.6 Similarity Index

We are proposing a heuristic similarity measure that computes the similarity degree between two morphological character descriptions. This measure will represent how closely related they are. The similarity index (S_i) is based on 2 weighted aspects. 25% of the index is calculated based on the taxa being described, i.e. it analyzes if two given characters (C_1 and C_2) describe the same taxa. The other 75% are based on the meaning of the character-states. It checks if the state labels being used are the same. This heuristic is still a work in progress. The weights assigned to parts of the index are configurable and their values were calibrated based on observations.

Let $G = (V(G), E(G))$ be a directed graph with vertex-set $V(G) = \{v_1, \dots, v_n\}$ and edge-set $E(G) = \{e_1, \dots, e_m\} \subset \{(v_i, v_j) | v_i, v_j \in V(G)\}$. Let $C_1, C_2 \in V(G)$ be two distinct vertices of G . We define the following sets:

$$N_{C_1} = \{v_i \in V(G) \mid (v_i, C_1) \in E(G)\} \quad (4.1)$$

$$N_{C_2} = \{v_i \in V(G) \mid (v_i, C_2) \in E(G)\} \quad (4.2)$$

$$S_1 = \frac{|N_{C_1} \cap N_{C_2}|}{\max\{|N_{C_1}|, |N_{C_2}|\}} \quad (4.3)$$

Let $f : E(G) \rightarrow \Upsilon$ be a labeling function, where Υ is a set of labels, and $f(e) \in \Upsilon$ is the label of edge $e \in E(G)$. We define the following sets:

$$L_{C_1} = \{e \mid e = f((v_i, C_1)) \in \Upsilon \text{ and } (v_i, C_1) \in E(G) \text{ and } v_i \in V(G)\} \quad (4.4)$$

$$L_{C_2} = \{e \mid e = f((v_i, C_2)) \in \Upsilon \text{ and } (v_i, C_2) \in E(G) \text{ and } v_i \in V(G)\} \quad (4.5)$$

$$S_2 = \frac{|L_{C_1} \cap L_{C_2}|}{\max\{|L_{C_1}|, |L_{C_2}|\}} \quad (4.6)$$

$$\textit{Similarity Index}(S_i) = 0.25 * S_1 + 0.75 * S_2 \quad (4.7)$$

S_1 defines a rate of common OTU vertices with edges for two given characters C_1 and C_2 . The S_1 result lies between 0 (no common OTUs) and 1 (all OTUs are common). N_{C_1} is the subset of incoming adjacent vertexes of C_1 and N_{C_2} is the subset of incoming adjacent vertexes of C_2 . Incoming adjacent vertexes of both C_1 and C_2 are always OTU vertexes, as shown in Figure 4.2. S_2 defines a rate of common labels of the incoming edges (character-states) for the characters C_1 and C_2 . The S_2 result also lies between 0 (no common character-states) and 1 (all character states are common). L_{C_1} and L_{C_2} are the subset of incoming adjacent edge labels (character-states) of C_1 and C_2 respectively.

It is important to note that the character labels of C_1 and C_2 are not being taken into account in the S_i formula. This intends to avoid weighting in favor of two identical

textual characters that do not have the same meaning, and to avoid weighting against two textual characters that are identical but do not have the same meaning. In practice, this will make the solution independent of the label and applicable for both presented scenarios (same label but different meanings and different labels and same meaning). Additionally, the symmetric property of equality is satisfied.

Practical Implementation of the Similarity Measure

Our system is able to draw a chart as illustrated in Figure 4.5, whose algorithm is inspired by the hierarchical edge bundling example (<http://mbostock.github.io/d3/talk/20111116/bundle.html>) of D3.js (<http://d3js.org/>) library. D3.js is a JavaScript library for manipulating documents and it has a wide variety of powerful visualization components. In the case of the hierarchical edge bundling example, it is necessary to provide only a “name” for each node and, inside a related “imports” sentence, the node name to where an edge must be created to. Listing 4.3 shows the JSON file that encodes the data used to generate Figure 4.5 (for implementation details see Appendix A.8).

Listing 4.3: RealExample.json

```

1  [
2  {"name": "root.Cauline cladotaxy" , "imports": ["root.Cauline
   cladotaxy" , "root.Phyllotaxy"]},
3  {"name": "root.Protoxylem position within the cauline stele"
   , "imports": ["root.Protoxylem position within the cauline
   stele"]},
4  {"name": "root.Organotaxy of the LBS" , "imports": ["root.
   Cauline cladotaxy" , "root.Phyllotaxy"]},
5  {"name": "root.Xylem configuration in the leaflets" , "
   imports": []},
6  {"name": "root.Planation" , "imports": []},
7  {"name": "root.Development of the foliar organ" , "imports":
   []},
8  {"name": "root.Phyllotaxy" , "imports": []},
9  {"name": "root.Xylem configuration in the rachis" , "imports"
   : []},
10 {"name": "root.Cauline cladotaxy" , "imports": []},
11 {"name": "root.Protoxylem position within the cauline stele"
   , "imports": []},
12 {"name": "root.Xylem configuration in the rachis" , "imports"
   : []},

```

```

13 { "name": "root.Extent of the planation" , "imports": [] } ,
14 { "name": "root.Presence of planated parts within the LBS" , "
    imports": [] } ,
15 { "name": "root.Xylem configuration in the leaflets" , "
    imports": [] } ,
16 { "name": "root.Development of the LBS" , "imports": [ "root.
    Development of the foliar organ" ] }
17 ]

```

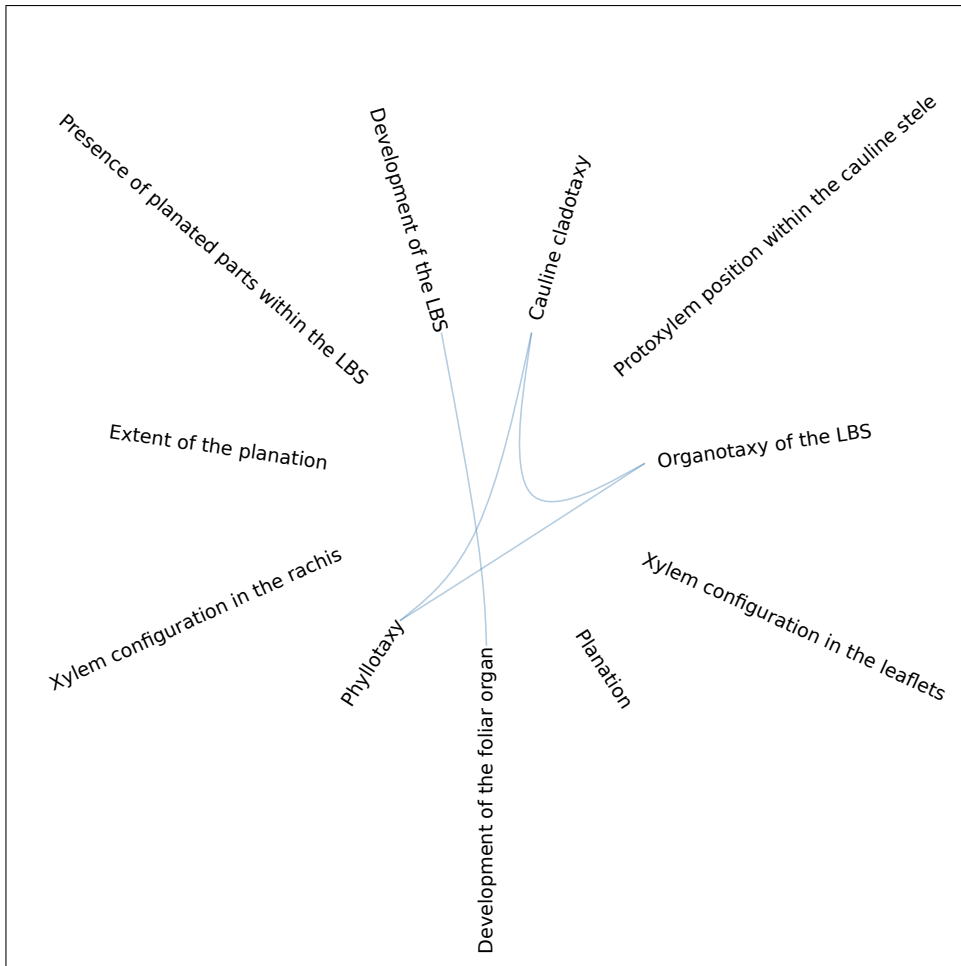


Figure 4.5: Practical Implementation

4.3.7 Tracing the Evolutionary History

The *TraceEvolutionaryHistory* class abstracts an important algorithm that traces a phylogenetic history of traits changes (for implementation details see Appendix A.9). This algorithm was built on top of our graph data model. It searches in a given tree for traits (characters) that might be the “responsible” for a tree branching, in which branching is considered as any division from a particular ancestor. For example, Figure 4.4 has two Hypothetical Taxonomic Units (HTU), in which the least nested one after the root has the *Pseudosporochnus* node and another HTU node as children. A typical question that motivated us to create such an algorithm was: What differentiates *Pseudosporochnus* from the other nodes?

The algorithm is divided into two recursive methods that are invoked in sequence. The first one *BottomUpAggregation* starts from a given point in the tree and goes down until it reaches Operational Taxonomic Unit (OTU) nodes. At this point, the method retrieves all outgoing relationships from the OTU node and starts going back towards the root. While the method is traversing internal HTU nodes ($current_{HTU}$) from the leaves back towards the root, it performs an union operation with the outgoing relationships of all children nodes – one occurrence for each type of relationship – and then, for each type of relationship of the resulting union, the method creates an edge departing from the current HTU ($current_{HTU}$) towards the original ending point of the relationship. In the end, the method returns all relationships outgoing from all nodes, including the intermediary HTU nodes ($current_{HTU}$). Figure 4.6 shows the result of *BottomUpAggregation* method being applied on the graph of Figure 4.4.

The second part of the algorithm is called *TopDownRefining*. This method is triggered after the *BottomUpAggregation* method, going to the same starting node provided in the *BottomUpAggregation* method. It starts from a given node ($node_n$) traversing down the tree and, in every HTU it reaches, it subtracts the set of character-states that starts in its children nodes ($node_{children}$) and points to a given character ($node_{character}$), from the set of character-states starting from itself ($node_n$) pointing to the same character node ($node_{character}$).

For example, in Figure 4.6, consider the least nested node ($node_0$), just after the root and linked to the *Webbing within the LBS* character node ($node_{webbingLBS}$). There are two edges connecting the $node_0$ and the $node_{webbingLBS}$ with values *present* and *absent*. The *present* edge comes from the most nested part of the tree, composed of the nodes *Zygopteris*, *Marattia*, *Equisetum* and *Ophioglossum*, nested by node 1 ($node_1$) – see Figure 4.4. The *absent* comes from *Pseudosporochnus* node – see Figure 4.4.

When the algorithm reaches $node_0$ it will subtracts the set of character-states (edges) outgoing from *Pseudosporochnus* toward $node_{webbingLBS}$ from the set of outgoing character-states (edges) outgoing from $node_0$ toward $node_{webbingLBS}$. This set subtraction will be

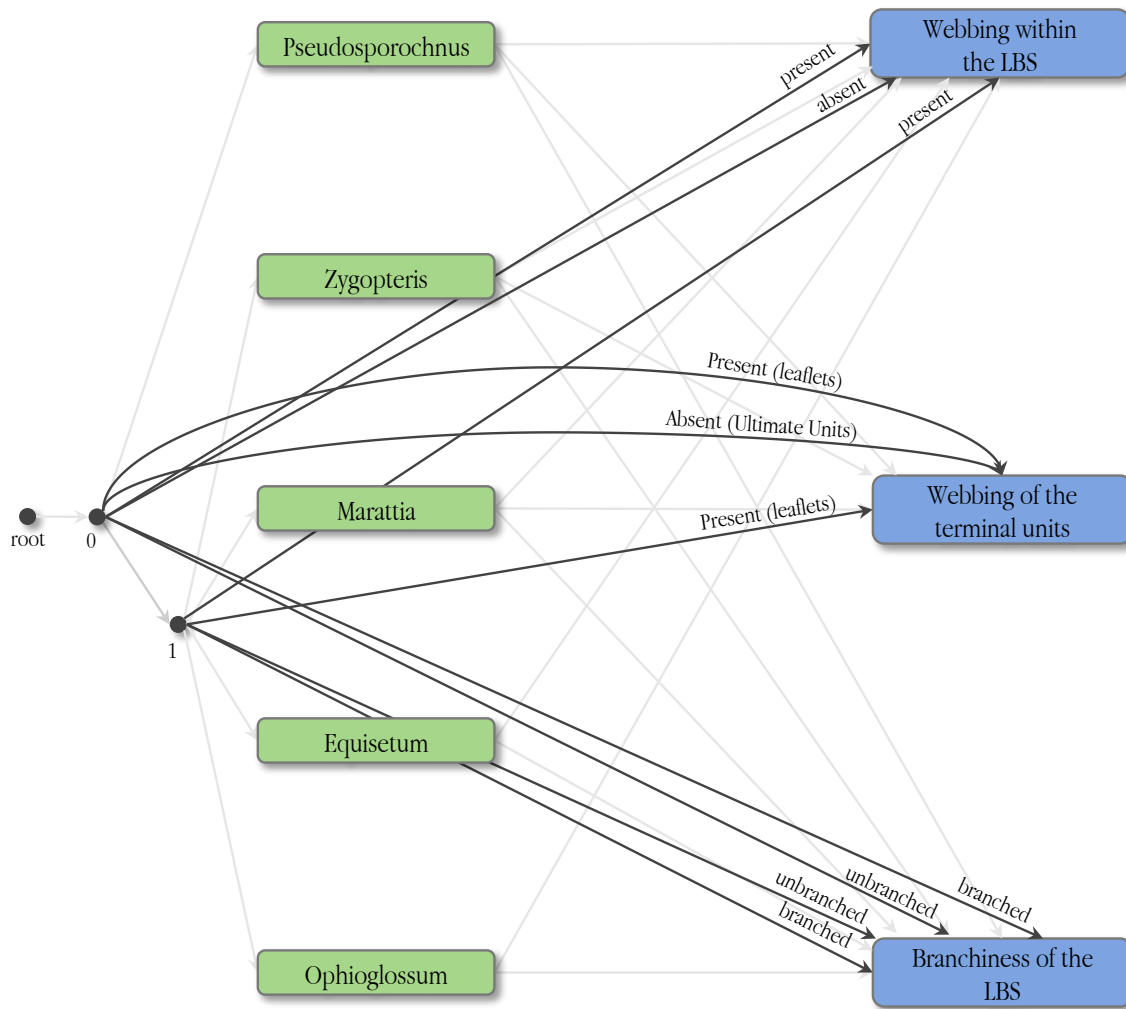


Figure 4.6: Bottom Up Aggregation

$\{present, absent\} - \{absent\} = \{present\}$. If the set subtraction result is not empty, it creates an edge called “*EvolvedTrait*” from itself ($node_0$) toward the character ($node_{webbingLBS}$) as shown in Figure 4.7.

Also, the algorithm will subtracts the set of character-states (edges) outgoing from $node_1$ toward $node_{webbingLBS}$ from the set of character-states (edges) outgoing from $node_0$ toward $node_{webbingLBS}$. This set subtraction will also not be empty ($\{present, absent\} - \{present\} = \{absent\}$) but the “*EvolvedTrait*” edge is created only once between $node_0$ and $node_{webbingLBS}$.

In a second iteration, the algorithm will reach $node_1$ and it will individually subtracts $node_1$ children nodes (*Zygopteris*, *Marattia*, *Equisetum* and *Ophioglossum*) outgoing character-states toward $node_{webbingLBS}$ from the set of character-states outgoing from

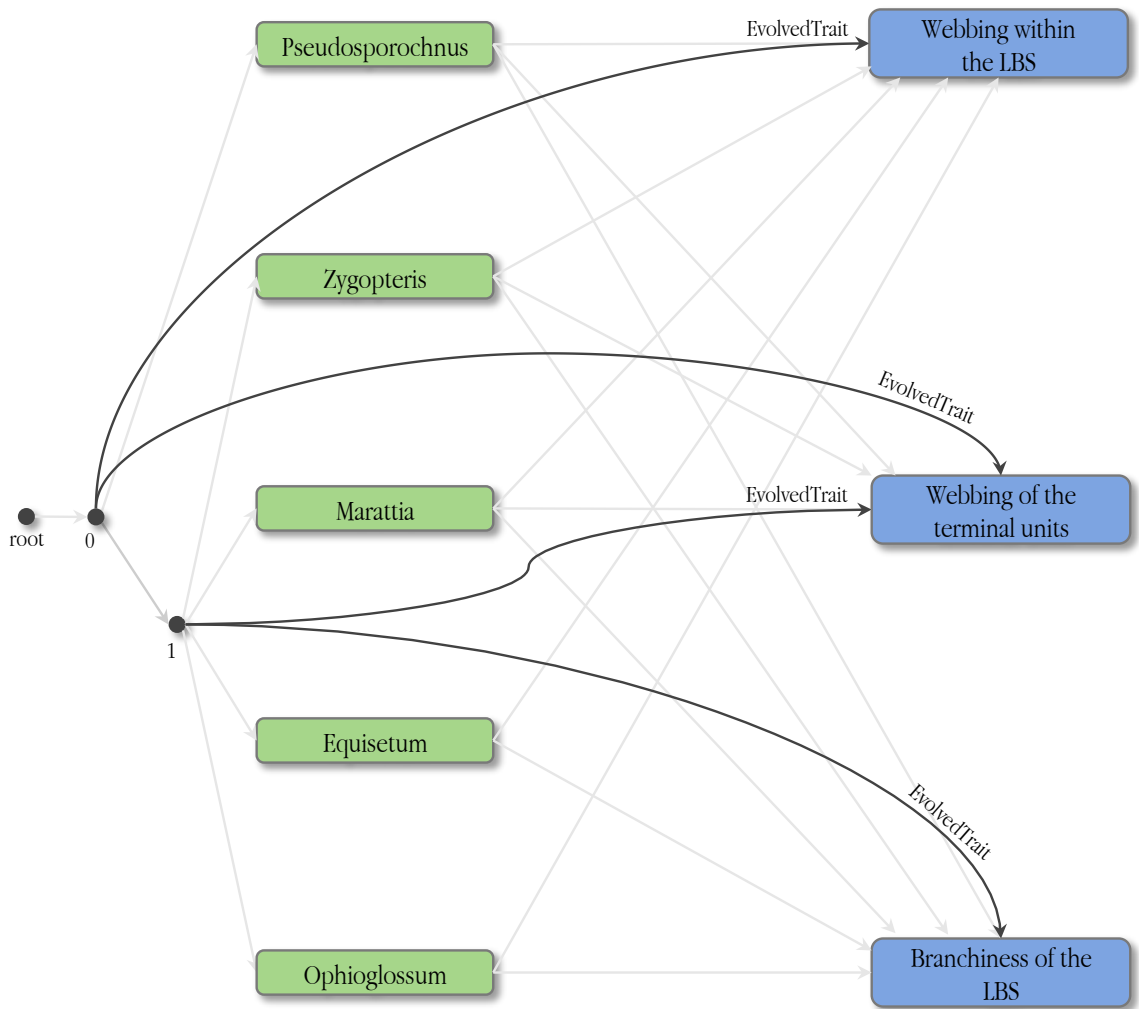


Figure 4.7: Top Down Refining

$node_1$ toward $node_{webbingLBS}$. All those set subtractions will be $\{present\} - \{present\} = \emptyset$. In such a case (empty set, \emptyset), no “*EvolvedTrait*” edge is created, as can be seen in Figure 4.7.

Finally there is a visual tool that presents to the user the tree structure with all characters flagged with the “*EvolvedTrait*” edge, i.e. the characters that the algorithm “suspect” of being responsible for the branching. Figure 4.8 is a screenshot of our visual tool.

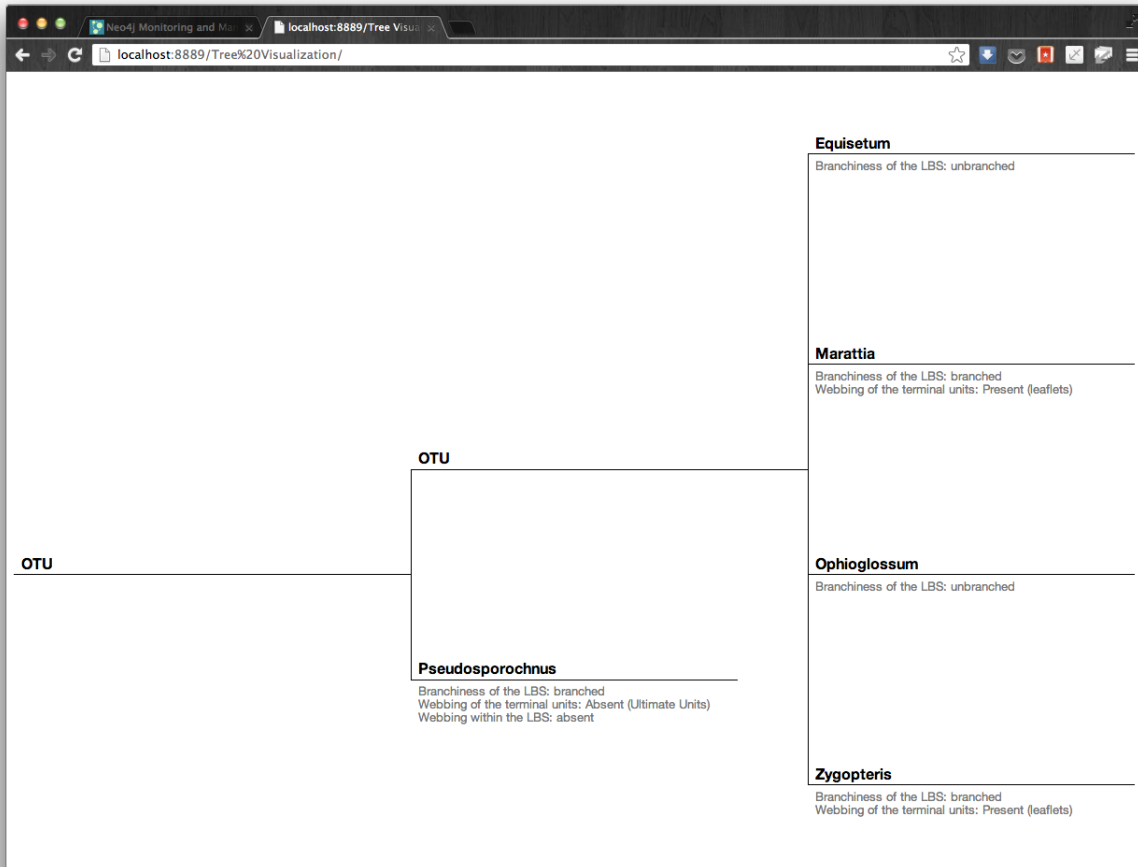


Figure 4.8: Evolved Traits Visualization

4.4 Conclusion

In this technical report we showed the main functionalities and operational features of the system. We mapped the SDD format to the graph model, remodeling semi-structured descriptions to a graph abstraction, in which the data are linked enabling coupling phylogenetic trees and phenotype descriptions. We drilled down the interconnection process through LSID unification, showing the required steps to obtain a valid LSID and implementation details of the services used in this process. We presented details regarding a visualization tool implemented on top of the D3.js, to visualize our proposed similarity measure. Such a solution will not only help discovering characters similarity, but will be very important in the next stage of this project, which is the mapping from the graph towards ontologies. Furthermore, an algorithm to trace the phylogenetic history of traits

changes has been shown. Finally, *Cypher* database queries and the main classes and methods of the system were provided with detailed comments for each method.

Chapter 5

Conclusions and Extensions

5.1 Contributions

This work is a starting point to understand and address the broader problem of integrating biological knowledge, in the context of phenotype description and phylogenetic tree reconstruction, which are heterogeneous in model and representation. We argue that an intermediate step between semi-structured data and ontologies, based on graph databases, can be exploited to emphasize relations among data elements.

We proposed a graph data model to congregate phenotype descriptions and phylogenetic trees. This model is a central part of this work. On top of it, we presented two approaches to progressively reflect the integration process via the graph structure. The first approach integrates knowledge around taxonomic units, the second one suggests correlations among characters. The correspondingly algorithms have the potential to simplify mappings to ontologies, as they support linking correlated terms.

The feasibility and potential of our approach were tested by practical implementations. In the first implementation, we showed the integration around the taxonomic units, in which states for a given character were unified across different description files. In the second implementation, two distinct descriptions of fossils were inserted into the graph and analyzed by the similarity measure proposed.

There are several extensions for this work including:

- The incorporation of morphological descriptions stored in other knowledge bases, e.g., MorphoBank (<http://morphobank.org/>), TreeBASE (<http://treebase.org/>) or Dryad (<http://datadryad.org/>). We consider that the integration with these bases can enhance the algorithms to find correlations, providing better insights.
- Further investigations around the similarity measure. A possible extension would be to consider the similarity among hierarchies of states.

- For the next stage of this project – which involves mapping the graph towards ontologies – the analysis of correlations can be extended to the relation between character nodes and ontology terms.
- This approach can be integrated with related work [2] concerning exploiting social knowledge to enrich ontologies.

Bibliography

- [1] Peter H Adler and Roger W Crosskey. World blackflies (diptera: Simuliidae): a comprehensive revision of the taxonomic and geographical inventory [2013], 2013. Accessed on July 08 2013.
- [2] Hugo Alves and André Santanchè. Folksonomized Ontology and the 3E Steps Technique to Support Ontology Evolvment. *Journal of Web Semantics*, 18(1):19–30, 2013.
- [3] R. Angles. A comparison of current graph database models. In *Data Engineering Workshops (ICDEW), 2012 IEEE 28th International Conference on*, pages 171–177, 2012.
- [4] Renzo Angles and Claudio Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40(1):1, 2008.
- [5] René Zaragüeta Bagils, Visotheary Ung, Anaïs Grand, Régine Vignes-Lebbe, Nathanaël Cao, and Jacques Ducasse. Lisbeth: New cladistics for phylogenetics and biogeography. *Comptes Rendus Palevol*, 11(8):563 – 566, 2012.
- [6] James P. Balhoff, Wasila M. Dahdul, Cartik R. Kothari, Hilmar Lapp, John G. Lundberg, Paula Mabee, Peter E. Midford, Monte Westerfield, and Todd J. Vision. Phenex: Ontological annotation of phenotypic diversity. *PLoS ONE*, 5(5):e10500, 05 2010.
- [7] Jonathan BL Bard and Seung Y Rhee. Ontologies in biology: design, applications and future challenges. *Nature Reviews Genetics*, 5(3):213–222, 2004.
- [8] Gordon Bell, Tony Hey, and Alex Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.
- [9] F.A. Bisby. The quiet revolution: biodiversity informatics and the internet. *Science*, 289(5488):2309–2312, 2000.

- [10] WJ Bock. Comparative morphology in systematics. *Systematic biology*, 411:441–448, 1969.
- [11] Daniel R Brooks. Hennig’s parasitological method: A proposed solution. *Systematic Biology*, 30(3):229–249, 1981.
- [12] Daniel R Brooks. Historical ecology: a new approach to studying the evolution of ecological associations. *Annals of the Missouri Botanical Garden*, pages 660–680, 1985.
- [13] Daniel R. Brooks, Marco G. P. Van Veller, and Deborah A. McLennan. How to do bpa, really. *Journal of Biogeography*, 28(3):345–358, 2001.
- [14] Nathanaël Cao, R Zaragüeta Bagils, Régine Vignes-Lebbe, et al. Hierarchical representation of hypotheses of homology. *Geodiversitas*, 29(1):5–15, 2007.
- [15] Francesca D Ciccarelli, Tobias Doerks, Christian Von Mering, Christopher J Creevey, Berend Snel, and Peer Bork. Toward automatic reconstruction of a highly resolved tree of life. *Science*, 311(5765):1283–1287, 2006.
- [16] T. Clark, S. Martin, and T. Liefeld. Globally distributed object identification for biological knowledgebases. *Briefings in bioinformatics*, 5(1):59–70, 2004.
- [17] Adèle Corvez, Véronique Barriol, and Jean-Yves Dubuisson. Diversity and evolution of the megaphyll in euphyllophytes: Phylogenetic hypotheses and the problem of foliar organ definition. *Comptes Rendus Palevol*, 11(6):403–418, 2012.
- [18] Wasila M. Dahdul, James P. Balhoff, Jeffrey Engeman, Terry Grande, Eric J. Hilton, Cartik Kothari, Hilmar Lapp, John G. Lundberg, Peter E. Midford, Todd J. Vision, Monte Westerfield, and Paula M. Mabee. Evolutionary characters, phenotypes and ontologies: Curating data from the systematic biology literature. *PLoS ONE*, 5(5):e10708, 05 2010.
- [19] Frédéric Delsuc, Henner Brinkmann, and Hervé Philippe. Phylogenomics and the reconstruction of the tree of life. *Nature Reviews Genetics*, 6(5):361–375, 2005.
- [20] Malte C Ebach, Christopher J Humphries, and David M Williams. Phylogenetic biogeography deconstructed. *Journal of Biogeography*, 30(9):1285–1296, 2003.
- [21] Michael Franklin, Alon Halevy, and David Maier. From databases to dataspace: a new abstraction for information management. *SIGMOD Rec.*, 34(4):27–33, December 2005.

- [22] Cynthia Gibas and Per Jambeck. *Developing bioinformatics computer skills*. O'Reilly Media, Inc., 2001.
- [23] Georgios Gkoutos, Eain Green, Ann-Marie Mallon, John Hancock, and Duncan Davidson. Using ontologies to describe mouse phenotypes. *Genome Biology*, 6(1):R8, 2004.
- [24] H.C.J. Godfray et al. Challenges for taxonomy. *Nature*, 417(6884):17–19, 2002.
- [25] A Grand, LM Duque, Velez, A Corvez, and M Laurin. Data from: Phylogenetic inference using discrete characters: performance of ordered and unordered parsimony and of three-item statements. *Biological Journal of the Linnean Society*, 2013.
- [26] Gregor Hagedorn. *Structuring Descriptive Data of Organisms – Requirement Analysis and Information Models*. PhD thesis, Universität Bayreuth, Fakultät für Biologie, Chemie und Geowissenschaften, 11 2007.
- [27] J. Kennedy, R. Kukla, and T. Paterson. Scientific names are ambiguous as identifiers for biological taxa: Their context and definition are required for accurate data integration. In *2nd Intl. Workshop on Data Integration in the Life Sciences (DILS)*, LNCS 3615, pages 80–95, July 2005.
- [28] Maurizio Lenzerini. Data integration: A theoretical perspective. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246. ACM, 2002.
- [29] Paula M Mabee. Integrating evolution and development: the need for bioinformatics in evo-devo. *BioScience*, 56(4):301–309, 2006.
- [30] Paula M Mabee, Michael Ashburner, Quentin Cronk, Georgios V Gkoutos, Melissa Haendel, Erik Segerdell, Chris Mungall, and Monte Westerfield. Phenotype ontologies: the bridge between genomics and evolution. *Trends in ecology & evolution*, 22(7):345–350, 2007.
- [31] David R. Maddison, David L. Swofford, and Wayne P. Maddison. Nexus: An extensible file format for systematic information. *Systematic Biology*, 46(4):590–621, 1997.
- [32] F Manola and E Miller. RDF Primer – W3C Recommendation. Technical report, W3C, 2004.

- [33] Mark A Miller, Wayne Pfeiffer, and Terri Schwartz. Creating the cipres science gateway for inference of large phylogenetic trees. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1–8. IEEE, 2010.
- [34] Eduardo Miranda and André Santanchè. Unifying phenotypes to support semantic descriptions. In *Proceedings of the 6th Seminar on Ontology Research in Brazil*, volume 1041, pages 154–165, 09 2013.
- [35] Gareth Nelson. Homology and systematics. *Homology: the hierarchical basis of comparative biology*, pages 101–149, 1994.
- [36] Gareth Nelson and Norman I Platnick. Three-taxon statements: A more precise use of parsimony? *Cladistics*, 7(4):351–366, 1991.
- [37] Gareth Nelson and Norman I. Platnick. Three-taxon statements: A more precise use of parsimony? *Cladistics*, 7(4):351–366, 1991.
- [38] R.D.M. Page. Biodiversity informatics: the challenge of linking data and the role of shared identifiers. *Briefings in Bioinformatics*, 9(5):345–354, 2008.
- [39] Cynthia S Parr, Robert Guralnick, Nico Cellinese, and Roderic DM Page. Evolutionary informatics: unifying knowledge about the diversity of life. *Trends in ecology & evolution*, 27(2):94–103, 2012.
- [40] D.J. Patterson, J. Cooper, PM Kirk, RL Pyle, and D.P. Remsen. Names are key to the big new biology. *Trends in ecology & evolution*, 25(12):686–691, 2010.
- [41] Richard A. Pimentel and Rhonda Riggins. The nature of cladistic data. *Cladistics*, 3(3):201–209, 1987.
- [42] Dennis Quan. Improving life sciences information retrieval using semantic web technology. *Briefings in bioinformatics*, 8(3):172–182, 2007.
- [43] Tony Rees. Taxamatch, a "fuzzy" matching algorithm for taxon names, and potential applications in taxonomic databases. In Anna Weitzman and Lee Belbin, editors, *Provisional Abstracts of the 2008 Annual Conference of the Taxonomic Databases Working Group*, Fremantle, Australia, 2008. Biodiversity Information Standards (TDWG) and the Missouri Botanical Garden.
- [44] Ian Robinson, Jim Webber, and Emil Eifrem. *Graph Databases*. O'Reilly Media, Inc., 2013.

- [45] Marko A. Rodriguez and Joshua Shinavier. Exposing multi-relational networks to single-relational network analysis algorithms. *Journal of Informetrics*, 4(1):29 – 41, 2010.
- [46] Joseph B Slowinski. “unordered” versus “ordered” characters. *Systematic Biology*, 42(2):155–165, 1993.
- [47] Visotheary Ung, Florian Causse, and Régine Vignes Lebbe. Xper²: managing descriptive data from their collection to e-monographs. 2010.
- [48] Visotheary Ung, Guillaume Dubus, René Zaragüeta-Bagils, and Régine Vignes-Lebbe. Xper2: introducing e-taxonomy. *Bioinformatics*, 26(5):703–704, 2010.
- [49] Rutger A Vos, James P Balhoff, Jason A Caravas, Mark T Holder, Hilmar Lapp, Wayne P Maddison, Peter E Midford, Anurag Priyam, Jeet Sukumaran, Xuhua Xia, et al. Nexml: rich, extensible, and verifiable representation of comparative data and metadata. *Systematic Biology*, 61(4):675–689, 2012.
- [50] Myrna E Watanabe. Assembling an online tree of life of two million species. *BioScience*, 63(1):64, 2013.
- [51] Mark Wilkinson. Ordered versus unordered characters. *Cladistics*, 8(4):375–385, 1992.
- [52] David M Williams and Malte C Ebach. The data matrix. *Geodiversitas*, 28(3):409–420, 2006.
- [53] René Zaragüeta-Bagils and Estelle Bourdon. Three-item analysis: Hierarchical representation and treatment of missing and inapplicable data. *Comptes Rendus Palevol*, 6(6):527–534, 2007.

Appendix A

Demonstration

In this section we present the source code of the system, according to the graph data model presented in previous sections. The code is modularized in files and each file has a class with methods, all with comments explaining their functionality.

A.1 SDDParser.py

```
1 import os,sys
2
3 from xml.dom import minidom
4 from collections import OrderedDict
5
6 from Representation import *
7 from StateDefinition import *
8 from CategoricalCharacter import *
9 from Categorical import *
10 from CodedDescription import *
11
12 class SDDParser:
13
14     def __init__(self, SDDFile):
15
16         self.CategoricalCharacters = self.
17             __parseCategoricalCharacter( SDDFile )
18         self.CodedDescriptions = self.__parseCodedDescription(
19             SDDFile )
```

```

18
19 def __parseRepresentation(self, Repr) :
20     """
21     Representation is a plain text label and description block
22     found inside CategoricalCharacter, StateDefinition and
23     CodedDescription blocks.
24     Args: A XML Representation block and its content.
25     Returns: A SDD Representation object.
26     """
27
28     label = ''
29     detail = ''
30
31     if Repr :
32
33         if 0 < Repr.getElementsByTagName('Label').length :
34             label = Repr.getElementsByTagName('Label')[0].childNodes
35                 [0].nodeValue.strip()
36
37         if 0 < Repr.getElementsByTagName('Detail').length :
38             detail = Repr.getElementsByTagName('Detail')[0].
39                 childNodes[0].nodeValue.strip()
40
41     return Representation( label, detail )
42
43 def __parseStateDefinitions(self, StateDefinitions):
44     """
45     StateDefinition has its own id and a Representation block.
46     It is define inside the CategoricalCharacter/States
47     block in which the States groups together all possible
48     states (StateDefinition) observed at a given
49     Categorical Character.
50     Args: All XML StateDefinition blocks of a particular
51     CategoricalCharacter/States block.
52     Returns: A dictionary of StateDefinition object.
53     """

```

```

47 # Dictionary with all state definition nodes
48 SStateDefinitionsDictionary = {}
49
50 for State in StateDefinitions :
51
52     Id = State.getAttributeNode('id').nodeValue
53
54     Repr = State.getElementsByTagName('Representation')[0]
55
56     Representation = self.__parseRepresentation( Repr )
57
58 # Add node to Dictionary
59 SStateDefinitionsDictionary[Id] = StateDefinition( Id ,
60     Representation )
61
62 return SStateDefinitionsDictionary
63
64 def __parseStates(self, States):
65     """
66     State is define inside CodedDescription/SummaryData/
67     Categorical and it links a taxon CategoricalCharacter
68     to its possible States through the ref parameters.
69     Args: All XML State blocks of a particular
70     CodedDescription/SummaryData/Categorical block.
71     Returns: An array with StateDefinitions references.
72     """
73
74     # Array with references to StateDefinitions
75     StatesDictionary = []
76
77     for state in States :
78
79         ref = state.getAttributeNode('ref').nodeValue
80         StatesDictionary.append( ref )
81
82     return StatesDictionary

```

```

81
82 def __parseSummaryData(self, Categoricals):
83     """
84     Categorical is a reference to a CategoricalCharacter
85     object and is composed by a list of references to
86     possible states that a given taxon can take.
87     Args: All XML Categorical blocks of a particular
88     CodedDescription/SummaryData block.
89     Returns: A dictionary of Categorical objects.
90     """
91
92     # Dictionary of Categorical objects
93     SummaryDataDictionary = {}
94
95     for c in Categoricals :
96
97         ref = c.getAttributeNode('ref').nodeValue
98
99         s = c.getElementsByTagName('State')
100
101         States = self.__parseStates( s )
102
103         SummaryDataDictionary[ref] = Categorical( ref , States )
104
105     return SummaryDataDictionary
106
107 def __parseCategoricalCharacter(self, SDDFile):
108     """
109     CategoricalCharacter has its own id, a Representation
110     block and a States block.
111     Args: A SDD file name.
112     Returns: A dictionary with all CategoricalCharacters
113     objects in the given file.
114     """
115
116     CC = SDDFile.getElementsByTagName('CategoricalCharacter')

```



```

114 # Dictionary with all CategoricalCharacters objects
115 CategoricalCharacters = {}
116
117 for Character in CC:
118
119     Id = Character.getAttributeNode('id').nodeValue
120
121     States = Character.getElementsByTagName('StateDefinition'
122     )
123
124     Repr = Character.getElementsByTagName('Representation')[
125     0]
126
127     Representation = self.__parseRepresentation( Repr )
128     SStateDefinitionsDictionary = self.
129     __parseStateDefinitions( States )
130
131     CategoricalCharacters[ Id ] = CategoricalCharacter( Id ,
132     SStateDefinitionsDictionary, Representation )
133
134 return CategoricalCharacters
135
136
137 def __parseCodedDescription(self, SDDFile):
138     """
139     CodedDescription has its own id, a Representation block
140     and a SummaryData block.
141     Args: A SDD file name.
142     Returns: A dictionary with all CodedDescription objects in
143     the given file.
144     """
145
146     CD = SDDFile.getElementsByTagName('CodedDescription')
147
148     # Dictionary with all CodedDescriptions objects
149     CodedDescriptions = {}
150
151     for Description in CD:

```

```

146     Id = Description.getAttributeNode('id').nodeValue
147
148     SD = Description.getElementsByTagName('Categorical')
149     Repr = Description.getElementsByTagName('Representation')
150         [0]
151
152     Representation = self.__parseRepresentation( Repr )
153     SummaryDataDictionary = self.__parseSummaryData( SD )
154
155     CodedDescriptions[ Id ] = CodedDescription( Id ,
156         SummaryDataDictionary, Representation )
157
158     return CodedDescriptions
159
160 def getAllStates(self):
161     """
162     Returns a dictionary of all 'StateDefinitions' elements.
163     """
164
165     States = {}
166
167     for key, CategoricalCharacter in self.
168         CategoricalCharacters.iteritems():
169
170         States.update( CategoricalCharacter.States )
171
172     OrderedStates = OrderedDict( sorted( States.items() ) )
173
174     return OrderedStates
175
176 def getAllTaxons(self):
177     """
178     Returns a list of all taxons elements.
179     """
180
181     Taxons = []

```

```
181
182     for key, CodedDescription in self.CodedDescriptions.
183         iteritems():
184         Taxons.append( CodedDescription.Representation )
185
186     return Taxons
187
188
189 def getAllCharacters(self):
190     """
191     Returns a dictionary of all 'CategoricalCharacter'
192     elements.
193     """
194     Characters = {}
195
196     for key, CategoricalCharacter in self.
197         CategoricalCharacters.iteritems():
198         Characters[ CategoricalCharacter.id ] = (
199             CategoricalCharacter.Representation )
200
201     OrderedCharacters = OrderedDict( sorted( Characters.items
202         ( ) ) )
203
204     return OrderedCharacters
```

A.2 TeeOutput.py

```

1 import re
2 import shlex
3 import mmap
4 import sys
5
6 from TreeNode import *
7 from NodeTypes import *
8
9 class TreeOutput:
10
11     def __init__(self, _TreeOutputFile ):
12
13         self.TreeOutputFile = _TreeOutputFile
14
15
16     def __parseNewickTree( self, NewickTree , parentNode ) :
17         """
18         Newick tree format (New Hampshire tree format) is a way of
19         representing trees in computer-readable form using
20         parentheses and commas.
21         Args:
22         NewickTree: A NewickTree string. For example: ((((((12
23             18) 22) 13) 3) 7) 30) (23 25))
24         parentNode: A node to where NewickTree tree will be
25             attached to.
26         """
27
28         opened = False
29         substring = NewickTree
30
31         i = j = begin = end = 0
32
33         for c in NewickTree :
34
35             if c == '(' :
36                 i += 1

```

```
33
34     if not opened :
35         begin = j
36
37         opened = True
38
39     elif c == ')' :
40         i -= 1
41
42     if opened and i == 0 :
43         # (opened and i == 0) means that opening round bracket
44         # '(' and the corresponding closing round bracket ')'
45         # was found.
46         # It will recursively call __parseNewickTree with
47         # brackets content. Also, it will remove parentheses
48         # block and content from NewickTree.
49
50         opened = False
51
52         childrenWithBrackets = NewickTree[ begin : j + 1 ]
53         childrenWithNoBrackets = NewickTree[ begin + 1 : j ]
54
55         child = TreeNode(None)
56         parentNode.appendChild( child )
57
58         self.__parseNewickTree( childrenWithNoBrackets , child )
59
60         substring = substring.replace( childrenWithBrackets, ""
61         )
62
63     j += 1
64
65     if "(" not in substring:
66         # When this condition is satisfied , it means that
67         # substring will only have leaves nodes or it is empty.
68
69         my_splitter = shlex.shlex(substring, posix = True )
70         my_splitter.whitespace += ','
```

```

65     my_splitter.whitespace_split = True
66
67     for n in my_splitter :
68         parentNode.appendChild( TreeNode(n) )
69
70
71 def getNewickTree( self ) :
72     """
73     This method looks into the file in search of the Newick
74     Tree and returns it.
75     The process is pretty straightforward:
76     1. Set the file's current position to the o occurrence of
77     'Retained trees'
78     2. Reads this line and discards it
79     3. Reads the next line, which supposedly should contain
80     the Newick Tree
81     4. Get the Newick Tree
82     """
83
84     _file = open( self.TreeOutputFile )
85     memorymap = mmap.mmap( _file.fileno(), 0, access = mmap.
86         ACCESS_READ )
87
88     RetainedTreesPosition = memorymap.find("Retained trees")
89     memorymap.seek( RetainedTreesPosition )
90     memorymap.readline()
91     FirstRetainedTreeLine = memorymap.readline()
92     memorymap.close()
93
94     # First occurrence of ')'
95     begin = FirstRetainedTreeLine.find('(')
96
97     # Last occurrence of '('
98     end = FirstRetainedTreeLine.rfind(')')
99
100    NewickTree = FirstRetainedTreeLine[ begin : end + 1 ]
101
102    return NewickTree

```

```
99
100
101 def getTaxons( self ) :
102     """
103     Get all taxons listed right bellow 'Taxa (# taxons)'
104     inside <D02> block and return all those taxons.
105     """
106     _file = open( self.TreeOutputFile )
107     memorymap = mmap.mmap( _file.fileno(), 0, access = mmap.
108         ACCESS_READ )
109
110     BlockBegin = memorymap.find("<D02>")
111     BlockEnd   = memorymap.find("<F02>")
112
113     TaxaPosition = memorymap.find( "Taxa" , BlockBegin ,
114         BlockEnd )
115
116     memorymap.seek( TaxaPosition )
117     TaxaLine = memorymap.readline()
118     TotalTaxa = int( re.search( re.escape( '(' ) + "(.*?)" +
119         re.escape( ')' ) , TaxaLine ).group(1) )
120
121     TaxonsDictionary = {}
122
123     for i in range( TotalTaxa ):
124
125         line = memorymap.readline()
126
127         index = line[ 1 : 21 ].strip()
128         taxon = line[ 22 : ].strip()
129
130         TaxonsDictionary[ index ] = taxon
131
132     memorymap.close()
133
134     return TaxonsDictionary
```

```

133
134 def __RenameTreeNodes( self , subTree , TaxonsDictionary ):
135
136     """
137     In a rooted phylogenetic tree , each node is called a
138     taxonomic unit . Internal nodes are generally called
139     hypothetical taxonomic units (HTUs) as they cannot be
140     directly observed .
141
142     Args:
143     subTree: Is a branch of the tree .
144     TaxonsDictionary: A list of taxons present in the 3iz
145     file .
146     """
147
148     if subTree.nodes:
149
150         subTree.value = str( NodeType.HTU )
151
152         for n in subTree.nodes:
153             self.__RenameTreeNodes( n , TaxonsDictionary )
154
155     else:
156         subTree.value = TaxonsDictionary[ subTree.value ]
157
158
159 def getTaxonsTreeStructure( self ):
160     """
161     It parse the NewickTree string into a tree structure with
162     Hypothetical Taxonomic Units as internal nodes and the
163     correct Taxon name as the leaves .
164     """
165
166     NewickTree = self.getNewickTree()
167     TaxonsDictionary = self.getTaxons()
168
169     root = TreeNode( None )
170     self.__parseNewickTree( NewickTree , root )

```



```
165     self.__RenameTreeNodes( root , TaxonsDictionary )
166
167     return root
```

A.3 GlobalNamesResolver.py

```
1 from bs4 import BeautifulSoup
2 from GNRResultObject import *
3 import urllib2
4 from enumerator import *
5
6 class GlobalNamesResolver:
7
8     def __init__(self) :
9         self.url = 'http://resolver.globalnames.org/name_resolvers
10             .xml?names='
11
12         # Names Data Sources <http://resolver.globalnames.org/
13             data_sources>
14         # ID Source
15         # 169 uBio NameBank
16         # 1 Catalogue of Life
17         # 3 ITIS
18         self.DataSources = enum( CatalogueOfLife = 1, ITIS = 3,
19             uBioNameBank = 169 )
20
21         self.DataSourceIds = [self.DataSources.CatalogueOfLife,
22             self.DataSources.ITIS, self.DataSources.uBioNameBank ]
23
24     def getResultsObjects( self, ScientificName ):
25
26         ScientificName = ScientificName.replace(' ', '%20')
27
28         url = self.url + ScientificName
29
30         if len( self.DataSourceIds ) > 0 :
31             url = url + '&data_source_ids='
32
33             for _id in self.DataSourceIds :
34                 url = url + str( _id ) + '|'
35
36         return urllib2.urlopen( url ).read()
```

```
33     try:
34         GNRServiceUrlResponse = urllib2.urlopen( url ).read()
35
36     except urllib2.HTTPError, e:
37         print "HTTP error: %d" % e.code
38     except urllib2.URLError, e:
39         print "Network error: %s" % e.reason.args[1]
40
41     SoupGNRRResponse = BeautifulSoup( GNRServiceUrlResponse )
42
43     results = SoupGNRRResponse.findAll('result')
44
45     GNRResultObjects = []
46
47     for result in results :
48
49         DataSourceId      = result.find('data-source-id', {'type':
50             'integer'} )
51         DataSourceTitle   = result.find('data-source-title')
52         gniUUID           = result.find('gni-uuid')
53         NameString        = result.find('name-string')
54         CanonicalForm     = result.find('canonical-form')
55         TaxonId           = result.find('taxon-id')
56         LocalId           = result.find('local-id')
57         MatchType         = result.find('match-type', {'type': '
58             integer'} )
59         Prescore          = result.find('prescore')
60         Score             = result.find('score', {'type': 'float'})
61
62         DataSourceId      = DataSourceId.contents[0]      if
63             DataSourceId      else ""
64         DataSourceTitle   = DataSourceTitle.contents[0]  if
65             DataSourceTitle   else ""
66         gniUUID           = gniUUID.contents[0]          if gniUUID
67             else ""
68         NameString        = NameString.contents[0]       if
69             NameString         else ""
```

```

64 CanonicalForm = CanonicalForm.contents[0] if
    CanonicalForm else ""
65 TaxonId       = TaxonId.contents[0]   if TaxonId
    else ""
66 LocalId      = LocalId.contents[0]   if LocalId
    else ""
67 MatchType    = MatchType.contents[0]  if
    MatchType    else ""
68 Prescore     = Prescore.contents[0]   if Prescore
    else ""
69 Score        = Score.contents[0]      if Score
    else ""
70
71 obj = GNRResultObject( DataSourceId, DataSourceTitle,
    gniUUID, NameString, CanonicalForm, TaxonId, LocalId,
    MatchType , Prescore , Score )
72
73 GNRResultObjects.append( obj )
74
75 return GNRResultObjects
76
77
78 def getCanonicalForm( self, ScientificName ) :
79     """
80     Returns the canonical forms of a given scientific name.
81     """
82
83     objects = self.getResultsObjects( ScientificName )
84
85     CanonicalForms = set( [ ] )
86
87     for obj in objects :
88
89         match = int(obj.MatchType)
90
91         # 1 - Exact match
92         # 2 - Exact match by canonical form
93         # 3 - Fuzzy match by canonical form

```

```
94     if match == 1 or match == 2 or match == 3 :
95
96         if 0.988 <= float(obj.MatchType) :
97
98             # Add canonical form to the set
99             CanonicalForms = CanonicalForms | set( [ obj.
100                 CanonicalForm ] )
101
102         if 1 == len( CanonicalForms ):
103
104             return sorted(CanonicalForms)[0]
105
106         return None
107
108 def getLSIDFromCanonicalForm( self, CanonicalForm ) :
109     """
110     Returns the LSID of a given Canonical Form. Only uBio
111     NameBank LSID are retrieved and still only if a exact
112     match occur.
113     """
114
115     ResultsObjects = self.getResultsObjects( CanonicalForm )
116
117     for obj in ResultsObjects :
118
119         if int(obj.MatchType) == 1:
120
121             if int(obj.DataSourceId) == self.DataSources.
122                 uBioNameBank:
123
124                 return obj.LocalId
125
126     return None
```

A.4 GNRResultObject.py

```
1 class GNRResultObject:
2
3 def __init__(self, _DataSourceId, _DataSourceTitle, _gniUUID
4     , _NameString, _CanonicalForm, _TaxonId, _LocalId,
5     _MatchType, _Prescore, _Score ):
6
7     # The id of the data source where a name was found.
8     self.DataSourceId = _DataSourceId
9
10    # The data source title where a name was found.
11    self.DataSourceTitle = _DataSourceTitle
12
13    # An identifier for the found name string used in Global
14    # Names.
15    self.gniUUID = _gniUUID
16
17    # The name string found in this data source.
18    self.NameString = _NameString
19
20    # A "canonical" version of the name generated by the Global
21    # Names parser
22    self.CanonicalForm = _CanonicalForm
23
24    # Tree path to the root if a name string was found within a
25    # data source classification.
26    # self.ClassificationPath
27
28    # self.ClassificationPathRanks
29
30    # Same tree path using taxon_ids
31    # self.ClassificationPathIds
32
33    # An identifier supplied in the source Darwin Core Archive
34    # for the name string record
35    self.TaxonId = _TaxonId
```

```
31 | # Shows id local to the data source (if provided by the
    | data source manager)
32 | self.LocalId = _LocalId
33 |
34 | # Explains how resolver found the name. If the resolver
    | cannot find names corresponding to the entire queried
    | name string, it sequentially removes terminal portions
    | of the name string until a match is found.
35 | # 1 - Exact match
36 | # 2 - Exact match by canonical form of a name
37 | # 3 - Fuzzy match by canonical form
38 | # 4 - Partial exact match by species part of canonical form
39 | # 5 - Partial fuzzy match by species part of canonical form
40 | # 6 - Exact match by genus part of a canonical form
41 | self.MatchType = _MatchType
42 |
43 | # Displays points used to calculate the score delimited by
    | '|' — "Match points|Author match points|Context points
    | ". Negative points decrease the final result.
44 | self.Prescore = _Prescore
45 |
46 | # A confidence score calculated for the match.
47 | # 0.5 means an uncertain result that will require
    | investigation.
48 | # Results higher than 0.9 correspond to 'good' matches.
49 | # Results between 0.5 and 0.9 should be taken with caution.
50 | # Results less than 0.5 are likely poor matches.
51 | # The scoring is described in more details on http://
    | resolver.globalnames.org/about
52 | self.Score = _Score
```

A.5 ITISServices.py

```
1 import suds
2
3 class ITISServices:
4
5     url = "http://www.itis.gov/ITISWebService.xml"
6     client = None
7
8     def __init__(self):
9         self.client = suds.client.Client( self.url )
10
11
12     def getTSNfromScientificName(self, ScientificName ):
13         """
14         Taxonomic Serial Number (TSN) which is the primary key for
15         the scientific name. This method returns a TSN if the
16         provided ScientificName is found and None otherwise.
17         """
18
19         self.client.service.searchByScientificName( ScientificName
20             )
21
22         ScientificNamesResponse = self.client.last_received().
23             getChild("soapenv:Envelope").getChild("soapenv:Body").
24             getChild("ns:searchByScientificNameResponse").getChild("
25                 ns:return").getChildren("ax21:scientificNames")
26
27         for sn in ScientificNamesResponse:
28
29             tsn = sn.getChild("ax21:tsn")
30
31             if tsn != None :
32                 return tsn.getText()
33
34         return None
```



```
31 def getLSIDfromTSN(self, tsn ):
32     """
33     Given a TSN this method returns a LSID if found and None
34     otherwise.
35     """
36     self.client.service.getLSIDFromTSN( tsn )
37
38     LSID = self.client.last_received().getChild("soapenv:
39         Envelope").getChild("soapenv:Body").getChild("ns:
40         getLSIDFromTSNResponse").getChild("ns:return").getText()
41
42     if LSID:
43         return LSID
44
45     return None
```

A.6 CoLServices.py

```
1 from BeautifulSoup import BeautifulSoup
2 import urllib2
3
4 class CoLServices:
5     """
6     This class contains the main methods to interact with the
7     CoL web service.
8     """
9     def getCoLUrl( self, ScientificName ):
10        """
11        This method uses a XML scraping technique to get the URL
12        of the given Scientific Name from the webservice
13        response.
14        """
15
16        url = 'http://www.catalogueoflife.org/col/webservice?name=
17            ,
18
19        ScientificName = ScientificName.replace(' ', '%20')
20
21        try:
22            CoLWebServiceUrlResponse = urllib2.urlopen(url +
23                ScientificName).read()
24        except urllib2.HTTPError, e:
25            print "HTTP error: %d" % e.code
26        except urllib2.URLError, e:
27            print "Network error: %s" % e.reason.args[1]
28
29        SoupCoLWebServiceResponse = BeautifulSoup(
30            CoLWebServiceUrlResponse)
```

```
31     if CoUrl:
32         return CoUrl
33
34
35 def getCoLSpecieID( self, ScientificName ):
36     """
37     This method uses a XML scraping technique to get the ID of
38     the given Scientific Name from the webservice response
39     .
40     """
41
42     url = 'http://www.catalogueoflife.org/testcol/webservice?
43         name='
44
45     ScientificName = ScientificName.replace(' ', '%20')
46
47     try:
48         CoWebServiceUrlResponse = urllib2.urlopen(url +
49             ScientificName).read()
50     except urllib2.HTTPError, e:
51         print "HTTP error: %d" % e.code
52     except urllib2.URLError, e:
53         print "Network error: %s" % e.reason.args[1]
54
55     SoupCoWebServiceResponse = BeautifulSoup(
56         CoWebServiceUrlResponse)
57
58     result = SoupCoWebServiceResponse.find('result')
59
60     if result:
61         findID = result.find('id')
62
63         if findID:
64             SpecieID = findID.contents[0]
65
66         if SpecieID:
67             return SpecieID
68
```

```
64     return None
65
66
67 def getLSIDfromSpecieID( self, SpecieID ):
68     """
69     This method uses a HTML screen-scraping technique to get
70     the LSID of the given SpecieID.
71     """
72     url = 'http://www.catalogueoflife.org/testcol/details/
73         species/id/'
74
75     try:
76         SpecieDetailsCoUrlResponse = urllib2.urlopen(url +
77             SpecieID).read()
78     except urllib2.HTTPError, e:
79         print "HTTP error: %d" % e.code
80     except urllib2.URLError, e:
81         print "Network error: %s" % e.reason.args[1]
82
83     SoupSpecieDetailsCoUrlResponse = BeautifulSoup(
84         SpecieDetailsCoUrlResponse)
85
86     LSID = SoupSpecieDetailsCoUrlResponse.find('span', {'
87         class': 'lsid'}).contents[0]
88
89     return LSID
90
91
92
93 def getLSIDfromSpecieUrl( self, SpecieUrl ):
94     """
95     This method uses a HTML screen-scraping technique to get
96     the LSID of the given SpecieUrl.
97     """
98
99     try:
100         SpecieDetailsCoUrlResponse = urllib2.urlopen(SpecieUrl).
101             read()
```

```
95     except urllib2.HTTPError, e:
96         print "HTTP error: %d" % e.code
97     except urllib2.URLError, e:
98         print "Network error: %s" % e.reason.args[1]
99
100     SoupSpecieDetailsCoUrlResponse = BeautifulSoup(
101         SpecieDetailsCoUrlResponse)
102
103     LSID = SoupSpecieDetailsCoUrlResponse.find('span', {'
104         class': 'lsid'}).contents[0]
```

A.7 GraphImporter.py

```

1  from py2neo import rest, neo4j, cypher
2
3  from SDDParser import *
4  from TreeOutput import *
5  from GlobalNamesResolver import *
6  from GraphDB import *
7  from NodeTypes import *
8  from RelationshipTypes import *
9  from ITIServices import *
10 from CoLServices import *
11
12 class GraphImporter:
13
14     SDDFilename = None
15     TreeFilename = None
16
17     def __init__(self, _SDDFilename, _TreeFilename,
18                 _IgnoreTreeFilename ):
19
20         self.SDDFilename = _SDDFilename
21         self.TreeFilename = _TreeFilename
22         self.IgnoreTreeFilename = _IgnoreTreeFilename
23
24     def __CreateTaxonsNodes(self, CodedDescriptions ) :
25         """
26         Add to the Graph DB all taxons elements as nodes. In case
27         the taxon node already exists, it uses the node in
28         GraphDB rather than create a new one.
29         Args: CodedDescriptions: A list of all Coded Descriptions
30             elements.
31         Returns: A dict mapping keys to the corresponding added
32             nodes. Each tuple is represented as (Taxon Name , node)
33             where the first element of the tuple is the taxon name
34             and the last one is the node itself.
35         Example:

```

```
30     {u'Equisetum' : [Node('http://localhost:7474/db/data/
31         node/142')],
32     u'Marattia' : [Node('http://localhost:7474/db/data/node
33         /131')],
34     u'Botryopteris': [Node('http://localhost:7474/db/data/
35         node/222')]}
36     """
37
38     gdb = GraphDB()
39     GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
40
41     if GDBConn is not None:
42
43         # Dictionary for all taxons nodes
44         TaxonsNodes = {}
45
46         GNR = GlobalNamesResolver()
47         ITIS = ITIServices()
48         CoL = CoLServices()
49
50         for key, CodedDescription in CodedDescriptions.iteritems
51             ():
52
53             node = None
54
55             taxonName = CodedDescription.Representation.label
56             taxonNameCF = GNR.getCanonicalForm( taxonName )
57
58             lsid = GNR.getLSIDFromCanonicalForm( taxonNameCF )
59
60             if lsid == None :
61                 tsn = ITIS.getTSNfromScientificName( taxonNameCF )
62                 lsid = ITIS.getLSIDfromTSN( tsn )
63
64             if lsid == None :
65                 SpecieID = CoL.getCoLSpecieID( taxonNameCF )
66                 lsid = CoL.getLSIDfromSpecieID( SpecieID )
```

```

64     n = gdb.getNodeByLSID( lsid )
65
66     if n is None:
67
68         # Create taxon node
69         node = GDBConn.create( { 'label' : taxonNameCF ,
70                                 'detail' : CodedDescription.Representation.
71                                     detail ,
72                                 'sourceId' : CodedDescription.id ,
73                                 'type' : str( NodeType.OTU ) ,
74                                 'LSID' : lsid } )
75     else :
76         node = n
77
78     # Add node to Dictionary
79     TaxonsNodes[ CodedDescription.Representation.label ] =
80         node
81
82     return TaxonsNodes
83
84 else:
85     print msg
86     return None
87
88 def __CreateStateDefinitionNodes( self, StateDefinitions ) :
89     """
90     Add to the Graph DB all state definition elements as nodes
91     .
92     Args:
93     StateDefinitions: A dictionary of all 'StateDefinitions'
94         elements.
95     Returns:A dict mapping keys to the corresponding added
96         nodes. Each tuple is represented as (Id , node) where
97         the first element of the tuple, Id (For example: s54)
98         is the SDD.XML StateDefinition ID and the last one is
99         the node itself.
100     Example:

```



```

94     {u's54': [Node('http://localhost:7474/db/data/node/142')
95             ],
96     u's43': [Node('http://localhost:7474/db/data/node/131')
97             ],
98     u's46': [Node('http://localhost:7474/db/data/node/222')
99             ]}
100 """
101
102 gdb = GraphDB()
103 GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
104
105 if GDBConn is not None:
106
107     # Dictionary for all state definition nodes
108     StateDefinitionsNodes = {}
109
110     for key, State in StateDefinitions.iteritems():
111
112         # Create state definition node
113         node = GDBConn.create({ 'label' : State.Representation.
114                                label ,
115                                'detail' : State.Representation.detail ,
116                                'sourceId' : State.id ,
117                                'type' : str( NodeTypes.description ) })
118
119         # Add node to Dictionary
120         StateDefinitionsNodes[ State.id ] = node
121
122     return StateDefinitionsNodes
123
124 else:
125     print msg
126     return None
127
128 def __CreateCharacterNodes( self, Characters ) :
129     """
130     Add to the Graph DB all characters elements as nodes.

```

```

128 Args:
129     Characters: A dictionary of all 'Characters' elements.
130 Returns: A dict mapping keys to the corresponding added
        nodes. Each tuple is represented as (Id , node) where
        the first element of the tuple , Id (For example: c19)
        is the SDD.XML CategoricalCharacter ID and the last one
        is the node itself.
131 Example:
132 {u'c19 ': [Node('http://localhost:7474/db/data/node/396')
        ],
133     u'c18 ': [Node('http://localhost:7474/db/data/node/395')
        ],
134     u'c5 ' : [Node('http://localhost:7474/db/data/node/400')
        ]}
135 """
136
137 gdb = GraphDB()
138 GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
139
140 if GDBConn is not None:
141
142     # Dictionary for all characters nodes
143     CharactersNodes = {}
144
145     for ID, Character in Characters.iteritems():
146
147         # Create state definition node
148         node = GDBConn.create({ 'label' : Character.label ,
149                                'detail' : Character.detail ,
150                                'sourceId' : ID ,
151                                'type' : str( NodeTypes.description ) })
152
153         # Add node to Dictionary
154         CharactersNodes[ ID ] = node
155
156     return CharactersNodes
157
158 else:

```

```
159     print msg
160     return None
161
162
163 def __JoinTaxonsNodesTreeStructureRecursion(self,
164     TaxonsNodes , subTree , parentNode ):
165
166     gdb = GraphDB()
167     GDBConn , msg = gdb.getPy2neoGraphDatabaseService()
168
169     if GDBConn is not None:
170
171         if subTree.nodes:
172
173             # Create Hypothetical Taxonomic Unit node
174             htuNode , = GDBConn.create({ 'label' : str( NodeTypes.HTU
175                 ) ,
176                 'type' : str( NodeTypes.HTU ) })
177
178             # Join Hypothetical Taxonomic Unit node to its parent
179             node
180             parentNode.create_relationship_to( htuNode , str(
181                 RelationshipTypes.TreeEdge ) , { "type" : str(
182                 RelationshipTypes.TreeEdge ) } )
183
184             for n in subTree.nodes:
185                 self.__JoinTaxonsNodesTreeStructureRecursion(
186                     TaxonsNodes , n , htuNode )
187
188         else:
189             # Get Taxonomic Unit (taxon name) already created ,
190             # passed through TaxonsNodes dictionary
191             tuNode = TaxonsNodes[ subTree.value ][0]
192
193             # Join Taxonomic Unit node to its parent node
194             parentNode.create_relationship_to( tuNode , str(
195                 RelationshipTypes.TreeEdge ) , { "type" : str(
196                 RelationshipTypes.TreeEdge ) } )
```

```

188
189     else:
190         print msg
191         return None
192
193
194     def __JoinTaxonsNodesTreeStructure(self, TaxonsNodes , Tree
195         ):
196         """
197         Join taxons nodes with the Newick tree structure.
198         """
199         gdb = GraphDB()
200         GDBConn, msg = gdb.getPy2neoGraphDatabaseService()
201
202         if GDBConn is not None:
203
204             self.__JoinTaxonsNodesTreeStructureRecursion( TaxonsNodes
205                 , Tree, gdb.getRootNode() )
206
207         else:
208             print msg
209             return None
210
211     def ImportUsingTaxonCharacterStateSchema( self ) :
212         """
213         Schema : Taxon(Node) -> CategoricalCharacter (Edge) ->
214                 StateDefinition (Node)
215         """
216         # Parse the SDD-XML file
217         SDDFile = minidom.parse( self.SDDFilename )
218
219         SDD = SDDParser( SDDFile )
220
221         CategoricalCharacters = SDD.CategoricalCharacters
222         CodedDescriptions = SDD.CodedDescriptions

```

```
223
224 # Create Taxons nodes in the Graph DB
225 TaxonsNodes = self.__CreateTaxonsNodes( SDD.
      CodedDescriptions )
226
227 # Join Taxons nodes in a tree structure
228 treeOutput = TreeOutput( self.TreeFilename )
229 tree = treeOutput.getTaxonsTreeStructure()
230 self.__JoinTaxonsNodesTreeStructure( TaxonsNodes, tree )
231
232 # Create State Definition nodes in the Graph DB
233 StateDefinitionsNodes = self.__CreateStateDefinitionNodes(
      SDD.getAllSates() )
234
235 for key, CodedDescription in CodedDescriptions.iteritems()
      :
236
237     # Check if the given key exists in the dictionary.
      Otherwise does not proceed by creating the
      relationship
238     if CodedDescription.Representation.label in TaxonsNodes:
239
240         for key, SummaryData in CodedDescription.SummaryData.
            iteritems():
241
242             States = CategoricalCharacters[SummaryData.ref].States
243
244             for StateRef in SummaryData.States :
245
246                 # Check if the given key exists in the dictionary.
                    Otherwise does not proceed by creating the
                    relationship
247                 if StateRef in StateDefinitionsNodes:
248
249                     taxonNode = TaxonsNodes[ CodedDescription.
                        Representation.label ][0]
250                     StateDefinitionsNode = StateDefinitionsNodes[
                        StateRef ][0]
```

```

251
252     CategoricalCharacter      = CategoricalCharacters [
253         SummaryData.ref ].Representation
254     CategoricalCharacterDetail = CategoricalCharacter.
255         detail if CategoricalCharacter.detail else ""
256     relationshipType          = CategoricalCharacter.label.
257         replace(' ', '_')
258
259 # Join Taxon nodes to State Definition node using
260     CategoricalCharacter.label as relationship
261     taxonNode.create_relationship_to(
262         StateDefinitionsNode , relationshipType , { "label
263             " : relationshipType ,
264             "type" : str(
265                 RelationshipTypes.descriptor
266             ) ,
267             "Detail" :
268                 CategoricalCharacterDetail }
269         )
270
271 def ImportUsingTaxonStateCharacterSchema( self ) :
272     """
273     Schema : Taxon (Node) -> StateDefinition (Edge) ->
274             CategoricalCharacter (Node)
275     """
276
277 # Parse the SDD-XML file
278     SDDFile = minidom.parse( self.SDDFilename )
279
280     SDD = SDDParser( SDDFile )
281
282     CategoricalCharacters = SDD.CategoricalCharacters
283     CodedDescriptions     = SDD.CodedDescriptions
284
285 # Create Taxons nodes in the Graph DB

```

```
277     TaxonsNodes = self.__CreateTaxonsNodes( SDD.  
278         CodedDescriptions )  
279  
280     # Join Taxons nodes in a tree structure  
281     treeOutput = TreeOutput( self.TreeFilename )  
282     tree = treeOutput.getTaxonsTreeStructure()  
283     self.__JoinTaxonsNodesTreeStructure( TaxonsNodes, tree )  
284  
285     # Create Characters nodes in the Graph DB  
286     CharactersNodes = self.__CreateCharacterNodes( SDD.  
287         getAllCharacters() )  
288  
289     for key, CodedDescription in CodedDescriptions.iteritems()  
290         :  
291  
292         # Check if the given key exists in the dictionary.  
293         # Otherwise does not proceed by creating the  
294         # relationship.  
295         if CodedDescription.Representation.label in TaxonsNodes:  
296  
297             for key, SummaryData in CodedDescription.SummaryData.  
298                 iteritems():  
299  
300                 # Check if the given key exists in the dictionary.  
301                 # Otherwise does not proceed by creating the  
302                 # relationship.  
303                 if SummaryData.ref in CharactersNodes:  
304  
305                     States = CategoricalCharacters[ SummaryData.ref ].  
306                         States  
307  
308                     for StateRef in SummaryData.States :  
309  
310                         taxonNode = TaxonsNodes[ CodedDescription.  
311                             Representation.label ][0]  
312                         CharacterNode = CharactersNodes[ SummaryData.ref ][0]  
313  
314                         StateDefinition = States[ StateRef ].Representation
```

```
305     StateDefinitionDetail = StateDefinition.detail if
        StateDefinition.detail else ""
306     relationshipType = StateDefinition.label.replace(' ',
        , '_')
307
308     # Join Taxon nodes to Categorical Character node
        using StateDefinition.label as relationship
309     taxonNode.create_relationship_to( CharacterNode ,
        relationshipType , { "label" : relationshipType ,
310                            "type" : str( RelationshipTypes.
                                descriptor ) ,
311                            "Detail" : StateDefinitionDetail }
        )
```


A.8 SimilarityIndex.py

```

1  from __future__ import division
2  import codecs
3  from py2neo import rest, neo4j, cypher
4  from GraphDB import *
5  from NodeAndRelationshipTypes import *
6
7  class SimilarityIndex :
8
9      def CalculateIndex(self, gdb, n1, n2 ) :
10
11         TAaux = gdb.getIncomingAdjacentNodes( n1 )
12         TBaux = gdb.getIncomingAdjacentNodes( n2 )
13
14         TA = []
15         for n in TAaux : TA.append( n[0] )
16
17         TB = []
18         for n in TBaux : TB.append( n[0] )
19
20         setTA = set( TA )
21         setTB = set( TB )
22
23         S1 = len( setTA & setTB ) / max( len( setTA ) , len( setTB
24             ) )
25
26         TE1aux = gdb.getIncomingAdjacentRelationships( n1 )
27         TE2aux = gdb.getIncomingAdjacentRelationships( n2 )
28
29         TE1 = []
30         for r in TE1aux: TE1.append( r[0]["label"] )
31
32         TE2 = []
33         for r in TE2aux: TE2.append( r[0]["label"] )
34
35         setTE1 = set( TE1 )
36         setTE2 = set( TE2 )

```

```

36
37 S2 = len( setTE1 & setTE2 ) / max( len( setTE1 ) , len(
    setTE2 ) )
38
39 SI = ( 0.25 * S1 + 0.75 * S2 )
40
41 return SI
42
43
44 def CompareStudies(self, TreeRootStudyA, TreeRootStudyB,
    LowerBoundary, JSONFilename ):
45     """
46     It calculates the Similarity Index for all characters
    between two studies taking them two by two. Only SI
    greater or equal to LowerBoundary are exported into the
    given Json file.
47     Args:
48     TreeRootStudyA: Study A tree root.
49     TreeRootStudyB: Study B tree root.
50     LowerBoundary: Lower Boundary condition.
51     JSONFilename: Filename where the JSON data should be saved.
52     """
53
54     gdb = GraphDB()
55
56     rangeA = gdb.getDescriptionNodesOfATree( TreeRootStudyA )
57     rangeB = gdb.getDescriptionNodesOfATree( TreeRootStudyB )
58
59     Similarity = SimilarityIndex()
60
61     JSON = "["
62
63     for i in rangeA :
64
65         ni = gdb.getNode( i )
66
67         JSON = JSON + "\n" + '{' + "\"name\": \"{0}\" , \"imports
            \": [\".format( "root." + ni["label"] )

```

```
68
69     imports = False
70
71     for j in rangeB :
72
73         nj = gdb.getNode( j )
74
75         SI = Similarity.CalculateIndex( gdb, ni, nj )
76
77         if LowerBoundary <= SI :
78             JSON = JSON + "\{0}\", ".format( "root." + nj["label"] )
79             imports = True
80
81         if imports :
82             # Remove the last comma
83             JSON = JSON[:-2]
84
85             JSON = JSON + "]},,"
86
87         for j in rangeB :
88             nj = gdb.getNode( j )
89             JSON = JSON + "\n" + '{' + "\"name\": \"{0}\" , \"imports
90                 \": []".format( "root." + nj["label"] ) + ',,'
91
92         # Remove the last comma
93         JSON = JSON[:-1]
94
95         JSON = JSON + "\n]"
96
97     text_file = open( JSONFilename, "w" )
98     text_file.write( JSON )
99     text_file.close()
```

A.9 TraceEvolutionaryHistory.py

```

1 import codecs
2
3 from py2neo import rest, neo4j, cypher
4 from GraphDB import *
5 from NodeAndRelationshipTypes import *
6
7 class TraceEvolutionaryHistory:
8
9     def BottomUpAggregation( self, gdb, node ):
10         """
11         This method starts from anywhere in the tree and goes down
12         until reach Operational Taxonomic Unit (OTU) nodes.
13         When it happens, the method basically retrieves all
14         outgoing relationships from the reached OTU node and
15         start going back toward the root. When the method is
16         traversing internal nodes (Hypothetical Taxonomic Units
17         ) from the leaves back toward the root it performs an
18         union operation with all children nodes outgoing
19         relationships – i.e., relationships of the same type
20         are ignored – and then for each relationship in the
21         union the method creates a relationship of the same
22         type changing the starting node to itself and the end
23         node remains the same. In the end, the method returns
24         all relationships outgoing from the given node.
25         Returns: Outgoing relationships of the given node. In case
26         the given node is an OTU, it returns only the
27         character–states relationships from the given node to
28         character nodes.
29         In case the given node is an HTU, the method returns all
30         outgoing relationships resulted from the union of its
31         children nodes outgoing relationships.
32         """
33
34         if node["type"] != NodeTypes.OTU and node["type"] !=
35             NodeTypes.description :

```

```
18     NeighborsNodes = gdb.getOutgoingAdjacentNodes( node )
19
20     relationships = []
21
22     for neighbor in NeighborsNodes:
23
24         rels = self.BottomUpAggregation( gdb, neighbor[0] )
25
26         relationships.append( rels )
27
28     # At this point we have all children nodes relationships.
29     # In such a case, we can implement the first part of
30     # the algorithm which is duplicate all relationships (
31     # union of children nodes relationships) in the given
32     # node.
33
34     for rels in relationships:
35
36         if rels is not None:
37
38             for rel in rels :
39
40                 if rel[0]["type"] == str( RelationshipTypes.
41                     descriptor ) :
42
43                     relType = rel[0].type.encode('ascii', 'ignore')
44
45                     startNode = node
46                     endNode   = rel[0].end_node
47
48                     # creating new relationships only where necessary
49                     gdb.getPy2neoGraphDatabaseService()[0].
50                         get_or_create_relationships( ( startNode, relType
51                             , endNode, { "type" : str( RelationshipTypes.
52                                 descriptor ) } ) )
53
54     return gdb.getOutgoingRelationships( node )
```

```

48
49
50 def TopDownRefining( self, gdb, node ):
51     """
52     This method essentially should be called just after the
53     BottomUpAggregation method passing the same starting
54     node provided in BottomUpAggregation method. It starts
55     from the given node (gn) back down the tree and in
56     every HTU it traverses it compare the character-states
57     starting from itself (gn) and pointing to a given
58     character (chaN) with every character-states that
59     starts in its children nodes (chiN) and points to the
60     same character node (chaN) for all character nodes it (
61     gn) points to. In case the comparation result is not
62     empty - i.e. the set difference between the chatacter-
63     states starting from the given node (gn) and the set of
64     character-states starting from the children node (chiN
65     ) is not empty - it creates a edge called 'EvolvedTrait
66     ' from itself (gn) to the given character (chaN).
67     """
68
69     if node["type"] != NodeType.OTU and node["type"] !=
70         NodeType.description :
71
72         NeighborNodes = gdb.getOutgoingAdjacentNodes( node )
73
74         tuNeighborNodes = []
75         descriptionNeighborNodes = []
76
77         for n in NeighborNodes:
78
79             if n[0]["type"] == NodeType.HTU or n[0]["type"] ==
80                 NodeType.OTU :
81
82                 tuNeighborNodes.append( n )
83
84             elif n[0]["type"] == NodeType.description :

```



```

96
97
98     gdb.deleteRelationshipsTypeFromNode( node, str(
99         RelationshipTypes.descriptor ) )
100
101     for tu in tuNeighborNodes:
102         self.TopDownRefining( gdb, tu[0] )
103
104
105 def __JSONencodingRecursion( self, gdb, node, TraitNodes,
106     nesting ):
107     """
108     It is part of the JSONencoding method.
109     Args:
110     node: Given node.
111     TraitNodes: Is the list of character nodes that node's
112     parent has a 'EvolvedTrait' edge pointing to.
113     nesting: Is the space (padding) on the left.
114     Returns: JSON string.
115     """
116
117     if node["type"] != NodeType.OTU and node["type"] !=
118         NodeType.description :
119
120         EvolvedTraitNodes = gdb.
121             getIncomingAdjacentNodesWithRelationshipInBetween(
122                 node, str( RelationshipTypes.EvolvedTrait ) )
123
124         NeighborNodes = gdb.
125             getIncomingAdjacentNodesWithRelationshipInBetween(
126                 node, str( RelationshipTypes.TreeEdge ) )
127
128     json = ''
129     json = json + "\n" + ' '.ljust( nesting ) + "{"
130
131     json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}\"
132         : \"{1}\"", ".format( "otu" , NodeType.OTU )

```



```

125     json = json + "\n" + ' '.ljust( nesting ) + "\"parents\"
        : ["
126
127     for nn in NeighborNodes:
128
129         result = self.__JSONencodingRecursion( gdb, nn[0],
            EvolvedTraitNodes , nesting + 2 )
130
131         json = json + result + ","
132
133     # Remove the last comma
134     json = json[:-1]
135
136     json = json + "\n" + ' '.ljust( nesting ) + "]"
137     json = json + "\n" + ' '.ljust( nesting ) + "}"
138
139     return json
140
141 elif node["type"] == NodeType.OTU:
142
143     json = ''
144     json = json + "\n" + ' '.ljust( nesting ) + "{"
145     json = json + "\n" + ' '.ljust( nesting + 2 ) + "\"{0}\"
        : \"{1}\"", ".format( "otu" , node["label"] )
146
147     i = 0
148     for trait in TraitNodes:
149
150         descriptions = gdb.getDistinctRelationshipsInBetween (
            node , trait[0] )
151
152         for desc in descriptions:
153
154             json = json + "\n" + ' '.ljust( nesting + 2 ) + "
                \"{0}{1}\" : \"{2}\"", ".format( RelationshipTypes.
                descriptor , str(i) , trait[0]["label"].encode('
                ascii', 'ignore') )

```

```

155     json = json + "\n" + ' '.ljust( nesting + 2 ) + "
        \"{0}{1}\" : \"{2}\"".format( NodeType.description
            , str(i), desc.encode('ascii', 'ignore').replace("_
            , " ") )
156     i = i + 1
157
158     # Remove the last comma
159     json = json[:-1]
160
161     json = json + "\n" + ' '.ljust( nesting ) + "}"
162
163     return json
164
165
166 def JSONencoding( self, JSONFilename, startNode ):
167     """
168     It exports to a JSON format the tree structure with all
        characters the algorithm flagged with 'EvolvedTrait'
        edge.
169     Args:
170     JSONFilename: Filename where the JSON data should be
        saved.
171     startNode: Node from where the data start being collected
        .
172
173     """
174
175     gdb = GraphDB()
176
177     json = self.__JSONencodingRecursion( gdb, startNode, [] ,
        0 )
178
179     text_file = open( JSONFilename, "w" )
180     text_file.write( json )
181     text_file.close()

```