Newcastle University

School of Computing Science

The Consistent Representation of Scientific Knowledge: Investigations into the Ontology of Karyotypes and Mitochondria.

Jennifer D. Warrender

Submitted for the degree of Doctor of Philosophy in the School of Computing Science, Newcastle University

June 2015

Abstract

Ontologies are widely used in life sciences to model scientific knowledge. The engineering of these ontologies is well-studied and there are a variety of methodologies and techniques, some of which have been re-purposed from software engineering methodologies and techniques. However, due to the complex nature of bio-ontologies, they are not resistant to errors and mistakes. This is especially true for more expressive and/or larger ontologies.

In order to improve on this issue, we explore a variety of software engineering techniques that were re-purposed in order to aid ontology engineering. This exploration is driven by the construction of two light-weight ontologies, The Mitochondrial Disease Ontology and The Karyotype Ontology. These ontologies have specific and useful computational goals, as well as providing exemplars for our methodology. This thesis discusses the modelling decisions undertaken as well as the overall success of each ontological model. Due to the added knowledge capture steps required for the mitochondrial knowledge, The Karyotype Ontology is further developed than The Mitochondrial Disease Ontology.

Specifically, this thesis explores the use of a pattern-driven and programmatic approach to bio-medical ontology engineering. During the engineering of our biomedical ontologies, we found many of the components of each model were similar in logical and textual definitions. This was especially true for The Karyotype Ontology. In software engineering a common technique to avoid replication is to abstract through the use of patterns. Therefore we utilised *localised patterns* to model these highly repetitive models. There are a variety of possible tools for the encoding of these patterns, but we found ontology development using Graphical User Interface (GUI) tools to be time-consuming due to the necessity of manual GUI interaction when the ontology needed updating. With the development of TawnyOWL, a programmatic tool for ontology construction, we are able to overcome this issue, with the added benefit of using a single syntax to express both simple and

patternised parts of the ontology.

Lastly, we briefly discuss how other methodologies and tools from software engineering, namely unit tests, diffing, version control and Continuous Integration (CI) were re-purposed and how they aided the engineering of our two domain ontologies.

Together, this knowledge increases our understanding in ontology engineering techniques. By re-purposing software engineering methodologies, we have aided construction, quality and maintainability of two novel ontologies, and have demonstrated their applicability more generally.

Declaration

I declare that this thesis is my own work unless otherwise stated. No part of this thesis has previously been submitted for a degree or any other qualification at Newcastle University or any other institution.

Jennifer D. Warrender

June 2015

Publications

Portions of the work within this thesis have been documented in the following publications:

Jennifer D. Warrender, Phillip Lord. The Karyotype Ontology: a computational representation for human cytogenetic patterns. *Bio-Ontologies 2013*, 2013.

Jennifer D. Warrender, Phillip Lord. A Pattern-driven Approach to Biomedical Ontology Engineering. *SWAT4LS 2013*, 2013.

Acknowledgements

First and foremost I would like to thank my main supervisor, Dr Phillip Lord, who provided me with the chance to complete my doctorate. With his continuous fortitude and endless guidance throughout this thesis, I was able to conclude this once-in-a-lifetime opportunity.

I would also like to thank my secondary supervisors Dr Simon J. Cockell and Dr Joanna L. Elson. Both supervisors were instrumental as sounding boards and reviewers of this work. Dr Joanna L. Elson especially was conducive in all of the mitochondrial work.

In addition to my supervisors, I am also grateful to Dr Michel Dumontier for his invaluable feedback on the work regarding localised patterns in The Semanticscience Integrated Ontology (SIO) (as described in Chapter 8). His feedback provided further confidence in the use of localised patterns in ontology engineering.

As part of the term capture process of The Mitochondrial Disease Ontology, as discussed in Section 7.2.2, I took part in a number of weekly The Mitochondrial Research Group (MRG) lab meetings. I would like to thank the individuals of MRG in allowing me to attend these meetings and share in their latest research.

For the duration of this thesis I was fortunate to be placed with some of the most generous individuals. I would like to thank all the individuals in the department and school for their friendship, support and humorous attitudes. I would especially like to thank Dr Michael Bell, whose optimism and LaTeX skills far outweigh my own. I would also like to thank Professor Anil Wipat and Professor Robert Stevens for their challenging discussions and constructive feedback.

Now that I am at the completion of my thesis I find myself forever indebted to my friends and family. To my friends, I thank you for your patience and unceasing faith. Lastly, to my parents, I thank you for being you with your continual encouragement and unwavering support throughout, despite the long and tough five years we have had.

Contents

1 Introduction **1**

1.1 Introduction . . . 2

1.2 Contributions of this thesis . . . 7

1.3 Thesis structure . . . 10

2 Background **13**

2.1 Scientific knowledge . . . 14

2.2 What is an Ontology? . . . 18

2.3 Technologies and Methodologies . . . 21

2.4 Summary . . . 24

3 Tawny-OWL **27**

3.1 Introduction . . . 28

3.2 The genesis of Tawny-OWL . . . 29

3.3 Tawny-OWL overview . . . 32

3.4 The many names of an entity . . . 40

3.5 Clojure overview . . . 43

3.6 Summary . . . 45

3.6.1 Presentation in this thesis . . . 45

4 Pattern-Driven Development **47**

4.1 Introduction . . . 48

4.2 Ontology Design Patterns (ODPs) implementation in Tawny-OWL . . . 50

4.3 Sources of data . . . 60

4.4 Localised patterns . . . 62

4.5 Summary . . . 65

5 Modelling Karyotypes **67**

5.1 Introduction . . . 68

5.1.1 Definition of terms . . . 69

5.2 What is an ISCN String . . . 72

5.2.1 Reviewing chromosome components . . . 82

5.2.2 Modelling requirements . . . 85
5.3 Design considerations . . . 88
5.3.1 Portions of reality . . . 88
5.3.2 A partonomic axiomitisation . . . 89
5.3.3 The event-based change axiomitisation . . . 92
5.4 Representing karyotypic knowledge . . . 95
5.4.1 Modelling chromosome components . . . 95
5.4.2 Modelling *normal* karyotypes . . . 96
5.4.3 Abnormality breakpoints . . . 97
5.4.4 Orientation of substitution segments . . . 99
5.4.5 Partial knowledge . . . 103
5.4.6 Modelling uncertainty . . . 103
5.4.7 Multiple copies of rearranged chromosomes . . . 104
5.4.8 Derivative chromosomes . . . 104
5.4.9 Abnormalities involving homologous chromosomes . . . 105
5.4.10 Constitutional anomalies . . . 107
5.4.11 Mosaic karyotypes . . . 107
5.4.12 Identifying the (near-)ploidy levels . . . 108
5.4.13 Defining sex . . . 109
5.5 Assessment . . . 111
5.6 Summary . . . 113

6 Scaling The Karyotype Ontology 115

6.1 Introduction . . . 116
6.2 Creating random ontologies . . . 117
6.3 Performance . . . 119
6.4 Scaling The Karyotype Ontology . . . 121
6.5 Incorporating `affects` restrictions . . . 122
6.5.1 The `affects` implementations . . . 122
6.5.2 Results . . . 127
6.6 Summary . . . 129

7 The Mitochondrial Domain **131**

7.1 Introduction . . . 132

7.2 Stage 1 – Term Capture . . . 135

7.2.1 Term of the week . . . 135

7.2.2 Lab meetings . . . 136

7.2.3 Published papers . . . 136

7.2.4 Assessing the term capture techniques . . . 139

7.3 Stage 2 – Competency Questions . . . 140

7.4 Stage 3 – Refinement . . . 141

7.4.1 Canonicalising terms . . . 141

7.4.2 Identifying disease relevant terms . . . 142

7.5 Stage 4 – Construction . . . 143

7.5.1 Constructing The Mitochondrial Disease Ontology classes . . . 144

7.6 Stage 5 – Evaluation . . . 148

7.7 Summary . . . 149

8 Patternised Development of an Existing Ontology **151**

8.1 Introduction . . . 152

8.2 Non-patternised rendering of Tawny-SIO . . . 153

8.3 Patternised refactoring of Tawny-SIO . . . 157

8.4 Tawny-SIO errors . . . 163

8.5 Patterns for downstream usage . . . 166

8.6 Summary . . . 170

9 Pattern classification **171**

9.1 Introduction . . . 172

9.2 Classification by role . . . 174

9.2.1 Internal localised patterns . . . 174

9.2.2 External localised patterns . . . 175

9.3 Results . . . 176

9.4 Summary . . . 182

10 Discussion **183**

10.1 Introduction 184

10.2 Utilising a pattern-driven and programmatic approach 185

10.3 The Karyotype Ontology 188

10.4 The Mitochondrial Disease Ontology 191

10.5 Re-purposing software engineering 193

10.6 Improving the ontology engineering process 197

A Recast of The Pizza Ontology **199**

B Tawny-OWL: Supplementary Material **201**

B.1 Tawny-OWL restriction exemplars 202

B.2 Tawny-OWL entity exemplars 204

B.3 Tawny-OWL frames 206

B.4 Defining a namespace in Tawny-OWL 209

C Mitochondria: Supplementary Material **211**

D Classification: Supplementary Material **217**

E Summary of research questions **221**

E.1 Summary of research questions 222

References **227**

List of Figures

2.1 The Protégé interface 22

3.1 Entity names 41

5.1 Top-level structure 68
5.2 Visualising the chromosome complement 71
5.3 Example chromosomal abnormalities 76
5.4 Translocation example 77
5.5 Chromosome components 83
5.6 Visualising sub-bands 84
5.7 Inverted chromosome 1 91
5.8 Limitations of `hasBreakPoint` 98
5.9 An inverse duplication event 100
5.10 A direct and inverse event 101
5.11 Homologous chromosome events 106

6.1 Performance check 120
6.2 Mean reasoning time taken 121
6.3 Diagrammatic representation of the affects implementations 123
6.4 Reasoning times for each affects implementation 128

7.1 Methodology overview 133
7.2 Term Capture results 138
7.3 Top-level structure 147

8.1 SIO workflow 153

9.1 Classification of ODPs 172
9.2 Classification of patterns 174

List of Tables

2.1 Ontology reasoning time 19

3.1 Ontology syntax overview 29
3.2 Tawny-OWL entity mappings 37
3.3 The subset of Tawny-OWL function aliases 38

5.1 Modal numbers 74
5.2 Structural chromosome rearrangements 78
5.3 Chromosome component statistics 85
5.4 Base karyotypes 96
5.5 Class statistics for The Karyotype Ontology 112

7.1 In-scope and quarantined statistics 142
7.2 Statistics for each generic class 145
7.3 Class statistics for The Mitochondrial Disease Ontology 146

8.1 Tawny-SIO entity name replacements 154
8.2 SIO comparison 161
8.3 List of all SIO errors 164
8.4 Correcting missing/incorrect annotations 165
8.5 Downstream ontology comparison 169

9.1 Classifying Tawny-Karyotype and Tawny-Karyotype-Scaling localised patterns 177
9.2 Classifying Tawny-Mitochondria localised patterns 179
9.3 Classifying Tawny-SIO localised patterns 180

B.1 Tawny-OWL ontology frames 206
B.2 Tawny-OWL class frames 206
B.3 Tawny-OWL individual frames 207
B.4 Tawny-OWL object property frames 207
B.5 Tawny-OWL annotation property frames 208
B.6 Tawny-OWL datatype property frames 208

C.1 Mitochondrial paper statistics 212
C.2 Mitochondrial disease terms 214

D.1 The VP ODP classification 218
D.2 The ME ODP classification 219

Acronyms

AI Artificial Intelligence

AOD The Agile Ontology Development

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ATP Adenosine TriPhosphate

BFO The Basic Formal Ontology

BMI Body Mass Index

BP Biological Process

CARMEN Code Analysis, Repository & Modelling for E-Neuroscience

CC Cellular Component

CI Continuous Integration

CP Content Pattern

CSV Comma Separated Values

CV Controlled Vocabulary

CVS Concurrent Versions System

CWA Closed-World Assumption

CdCS Cri du Chat Syndrome

DAML DARPA Agent Markup Language

DC Dublin Core

DDC Dewey Decimal Classification

DL Description Logic

DNA DeoxyriboNucleic Acid

EFO The Experimental Factor Ontology

FISH Fluorescence In Situ Hybridization

FMA Foundational Model of Anatomy

GO The Gene Ontology

GOA Gene Ontology Annotation

GUI Graphical User Interface

IRI Internationalized Resource Identifier

IDE Integrated Development Environment

ISCN International System for human Cytogenetic Nomenclature

JVM Java Virtual Machine

KB Knowledge Base

M^2 Mapping Master

MELAS Mitochondrial Encephalomyopathy, Lactic Acidosis, and Stroke-like episodes

MF Molecular Function

MRG The Mitochondrial Research Group

mtDNA mitochondrial DNA

NOR Nucleolus Organiser Region

OBO Open Biomedical Ontologies

ODP Ontology Design Pattern

OIL Ontology Interface Language

OMIM Online Mendelian Inheritance in Man

OOPS! The OntOlogy Pitfall Service!

OPPL The Ontology Pre-Processing Language

OWA Open-World Assumption

OWL The Web Ontology Language

PATO The Phenotype And Trait Ontology

POM Project Object Model

PROV-O The PROV Ontology

RDF Resource Description Framework

REPL Read-Eval-Print Loop

RFC Request For Comments

RIO Regularities Inspector for Ontologies

RNA RiboNucleic Acid

SDP Software Design Patterns

SHU Scoville Heat Units

SIO The Semanticscience Integrated Ontology

SKOS The Simple Knowledge Organization System

SO The Sequence Ontology

SNOMED CT The Systematized Nomenclature Of MEDicine Clinical Terms

TA The Terminologia Anatomica

TCA TriCarboxylic Acid cycle

TPS The Pretty Turtle Syntax

UCS Universal Character Set

UMDF The United Mitochondrial Disease Foundation

UML Unified Modeling Language

UniProtKB The UniProt Knowledge Base

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML eXtensible Markup Language

1

Introduction

Contents

1.1 Introduction 2

1.2 Contributions of this thesis 7

1.3 Thesis structure 10

1.1 Introduction

Science is a knowledge-rich discipline, that as a process, involves the building, capturing and organisation of a vast amount of knowledge. The creation of knowledge is difficult, as is the storing and structuring of knowledge so that it can be sorted, retrieved and used. Historically, the successful structuring of knowledge can be pivotal to discovery: one well-known example occurred in the 18th Century, when Carl Linnaeus introduced the Linnaean taxonomy [68], an early biological classification of organisms. This taxonomy was fundamental to the biological community and enabled an enormous increase in our biological understanding.

The quantity of knowledge is continually increasing. For example in PubMed[1] the number of papers has grown from approximately 16 million to 24 million, over the past 10 years[2]. In general, with an increase in size, there is an increase in complexity, thus the whole process of structuring knowledge gets harder. Being able to manage and deal with this problem is one of the challenges for science in the 21st Century.

One way of making scientific knowledge easier to understand, define, quantify, visualise or simulate is by creating scientific models. A simple example of a scientific model used in biology is the modelling of a nucleic acid; using a succession of letters we[3] indicate the order of the nucleotides and represent information about a molecule. Within a DeoxyriboNucleic Acid (DNA) molecule the letters `A`, `C`, `G` and `T` represent the `Adenine`, `Cytosine`, `Guanine`, and `Thymine` nucleotides respectively [2].

However scientific knowledge can be complex, multi-scaled, and generated in a distributed and autonomous manner. There are a variety of ways we model this complex knowledge, including Petri Nets [10], networks [50] and differential equations [155]. These forms of modelling deal well with knowledge of a certain sort. In this thesis, the modelling technology that we will be using to model our biological domain is an *ontology*, which is well-known for the handling of categorical biological knowledge.

[1] `http://www.ncbi.nlm.nih.gov/pubmed`

[2] Identified using a search for papers published between 1908 and 2004 versus 1908 and present (2014).

[3] Throughout this thesis plurals are used; this does not indicate multiple authorship unless stated explicitly.

Originally used in philosophy, the term ontology was adopted by Artificial Intelligence (AI) researchers, who created ontologies as computational models and viewed them as a form of applied philosophy. Perhaps the most well-known definition of an ontology from a computer science perspective is attributed to Tom Gruber [49], who defines an ontology as an explicit and formal specification of a conceptualisation. In this thesis, we define an ontology as a model of objects or process in the real-world, that captures knowledge about a domain by describing these objects as well as the relationships between these objects. The benefits of ontologies include their ability to share a common understanding of the structure of knowledge, the ability to analyse the domain knowledge as well as keep the ontology in a form readable to both humans and machines.

One well-known ontology in biology is The Gene Ontology (GO) [161]. GO[4] is a structured Controlled Vocabulary (CV) of terms and relationships for the cataloguing of gene product properties. GO was originally made up of three non-overlapping sub-ontologies: Molecular Function (MF); Biological Process (BP); and Cellular Component (CC), where each ontology describes a particular aspect of a gene or gene product as well as the relations between these terms [5]. The ontology has since evolved to include cross-links between the three domains [6]. The success of GO [7] caused an explosion in the awareness and usage of ontologies in biology [60]. Many of these ontologies can be found in online ontology libraries such as NCBO BioPortal[5] [101], which at this time of writing (2014) contains 384 ontologies, whilst Gene Ontology Annotations (GOAs) [20] are commonplace within biological databases, such as The UniProt Knowledge Base (UniProtKB) [3].

Ontologies can be encoded in a variety of different ontology languages; one of the most popular is The Web Ontology Language (OWL) [161], a standard defined by the World Wide Web Consortium (W3C)[6] in 2004 [87] and subsequently updated in 2009 [160].

The development of ontologies is aided by the use of (bespoke) ontology editors.

[4] http://www.geneontology.org/
[5] http://bioportal.bioontology.org/
[6] http://www.w3.org/Consortium/

A popular OWL ontology editor is Protégé [161]. Protégé[7], is a free, open-source desktop platform that provides users with the ability to create and manipulate OWL ontologies. Built in Java, Protégé is an extensible framework with plugins that, for example, allow the visualisation and reasoning ontologies. Another editor available from the developers of Protégé, is WebProtégé [153], which has similar capabilities, but is tailored to aid the development of collaborative OWL ontologies on the Web. Though these editors are developed as OWL editors, both can support other ontology syntax such as the OBO flat file format [99]. Another editor, OBO-Edit was originally created for GO and later extended to handle other Open Biomedical Ontologies (OBO) ontologies [27]. Overall, one characteristic of all these tools is that they were built specifically for building ontologies.

Ontology engineering is well-studied and there are a variety of methodologies in existence. Some of these are analogous to software engineering methodologies. For example V-model [144], Spiral [51], Waterfall and Iterative-Incremental [147]. Furthermore, this parallel between software engineering and ontology engineering is not limited to methodologies. For example, originally popularised in the context of software engineering [40], SDP has been recast into Ontology Design Patterns (ODPs). We define ODPs as formal, reusable and successful modelling solutions to recurrent modelling problems that are used for creating and maintaining ontologies [35]. Many of these ODPs can be found in online ODP libraries such as ontologydesignpatterns.org [119] and the ODPs public catalog[8]. The main usage of ODPs is to help with the construction of ontologies while avoiding common mistakes [93]. However, even though ODPs are useful in ontology engineering, they are used sparingly in bio-medical ontologies [94]. This could be due to a lack of knowledge or appropriate tooling [95]. In addition, even with the use of patterns, due to the complex nature of bio-ontologies, they are not resistant to errors and mistakes. This is especially true for more expressive and/or larger ontologies.

Another example of a re-purposed software engineering technology is static code analysis, specifically through the use of *lints*. In software engineering, lints are

[7] `http://protege.stanford.edu/`
[8] `http://www.gong.manchester.ac.uk/odp/html/`

generally used to aid development and maintenance by flagging suspicious language usage, as well as identifying syntactic discrepancies. Generic lints are available to the ontology community by tools called validators. One well-known validator is the The OntOlogy Pitfall Service! (OOPS!) [118], which tests ontologies for generic pitfalls such as missing labels and comments. The efovalidator[9] is a more specific validator that can only be used for the validation of The Experimental Factor Ontology (EFO), an ontology that provides formal description of many experimental variables [79].

Using this software analogy, we ask the simple question: if patterns are useful for extending and manipulating existing ontologies built with tools such as Protégé, how would the process of ontology engineering change if we inverted the traditional use and (programmatically) built with patterns from the start. In addition, we actively pursue the notion that other parts of software engineering, in addition to patterns, can be re-purposed and recast for use within ontology engineering.

In this thesis, we explore the usage of pattern-driven and programmatic approach to ontology engineering, by developing two significant ontologies of biology; specifically in the karyotypes and mitochondria domain. We use this ontology development to answer a number of research questions:

RQ1 How can we build a computational representation of the International System for human Cytogenetic Nomenclature (ISCN) using a pattern-driven and programmatic approach?

RQ2 Can we apply this approach to model new areas of biology and produce useful computational artefacts?

RQ3 What are the advantages and benefits of this approach to ontology engineering?

The research in this thesis is largely driven by these two novel ontologies, which are useful in their own right, and also serve as real-world exemplars. We wish to more formally and unambiguously define the complex knowledge found in two developing

[9] `http://www.ebi.ac.uk/fgpt/sw/efovalidator/index.html`

research areas; mitochondrial disease and karyotypes, and potentially facilitate and support clinical decisions.

1.2 Contributions of this thesis

This thesis focuses on the ontological modelling of scientific knowledge, using a pattern-driven and programmatic approach. Investigations using this approach resulted in the development of two novel, light-weight ontologies: The Karyotype Ontology[10] and The Mitochondrial Disease Ontology[11].

The first ontology, The Karyotype Ontology, describes human chromosomes as seen under the microscope. This ontology was designed to be a replacement for the current nomenclature, which is based on semantically meaningful strings that do not have a formal interpretation. With this computational representation, cytogeneticists will potentially have the ability to transform collections of karyotypes to a form that is easy to query, validate and maintain. The second ontology, The Mitochondrial Disease Ontology, describes mitochondrial disease related terms. This ontology is the first step in building an ontology which will enable us to structure and organise knowledge about a small (and hopefully tractable) organelle, to increase our understanding of this domain. With The Mitochondrial Disease Ontology we would potentially have the ability to classify and clarify mitochondrial disease by their symptomatic and/or genomic definition.

During the engineering of our domain ontologies, we found many repetitive components having similar textual and/or logical definitions. In software engineering, patterns are a common method of avoiding repetition. Therefore, we have investigated the use of patterns within our ontology building, to model and abstract over these repetitive components. While ODPs have been described previously, here we discuss *localised patterns* that were built for a specific purpose, rather than generically for reuse in other ontologies. We found custom construction patterns that are specific to an ontology, to be more useful for our specific needs and easy to express. This is also true for specific patterns that aid potential downstream users of our ontologies. Lastly, we show that two existing ODP classifications are insufficient in the classification of localised patterns. Thus we introduce a novel classification of localised patterns (and their relation to existing ODP classifications) as well as

[10] `https://w3id.org/ontolink/karyotype/`
[11] `https://w3id.org/ontolink/mitochondria/`

provide basic statistics of the localised patterns used in this work in order to improve the ontological community's understanding of localised patterns .

While there are numerous existing tools for the encoding of patterns, the tool we used was a novel environment called Tawny-OWL[12], which was built in parallel with and motivated by the work described in this thesis. Throughout the software's agile life cycle we have evaluated the tool's fitness for purpose (as a Clojure library to build ontologies), provided minor fixes and documented the application of Tawny-OWL to three bio-ontologies.

As Tawny-OWL is built on a programming language (Clojure) we can use its programmatic nature to automatically and consistently (re-)generate ontologies easily and quickly in order to provide a novel means to explore modelling choices and determine how well they scale. We generated numerous test ontologies (of various size and axiomitisations) and calculated the mean time taken to reason them. We found that The Karyotype Ontology can comfortably scale to 10^5 karyotypes and that there are two viable ways modelling the `affects` restriction.

In addition, we have shown that by using this approach we enforce consistency and thus potentially identify any errors. By explicitly encoding the ISCN and its exemplars, we found documentation errors (e.g. missing bands). In recasting The Semanticscience Integrated Ontology (SIO) and its downstream patterns we found errors within the ontology itself and the documentation (i.e. the SIO wiki). Some of these errors have been sent to the authors of SIO, who have since updated the SIO wiki. The majority of the identified errors were found as we had to explicitly handle exceptions to our preconceived patterns.

More generally, we show that the re-purposing of many programmatic tools and methodologies has additional advantages for ontology development. For example, with the use of an explicit abstraction, we are able to cleanly separate out the knowledge from the axiomitisation. This means that we can utilise an agile approach to manipulate OWL entities and axioms rapidly, even if these changes affect many OWL objects. Unit tests frameworks for software engineering can be used for the same purpose with ontologies. This is desirable and we have implemented this into

[12]Developed by Dr Phillip Lord, Newcastle University.

our repositories. With the use of Continuous Integration (CI), we were able to automate the running of our unit tests. Lastly, with the use of version control, we were able to develop ontologies in a distributed and non-linear manner while efficiently tracking ontology changes, which is otherwise difficult with OWL.

1.3 Thesis structure

This thesis is divided into the following chapters:

- In Chapter 2 we broadly introduce ontologies and their usage in the bio-medical domain. We extend this background to review ontology engineering methodologies and technologies, as well as the use of agile software engineering techniques to aid ontology engineering.

- In Chapter 3 we introduce the Tawny-OWL library, the mechanism that was used for the construction of the ontologies, and was developed as a result of the work described in this thesis. We show how OWL entities and axioms are defined in Tawny-OWL, by introducing Clojure syntax and the frame-based syntax of Manchester Syntax. These exemplar OWL entity definitions in Tawny-OWL are taken from the exemplar ontology; The Pizza Ontology.

- In Chapter 4 we show how generic and parameterisable patterns, known as localised patterns, are encoded in Tawny-OWL. Similar to Chapter 3, examples are taken from the exemplar ontology; The Pizza Ontology, and the Tawny-OWL library itself.

- In Chapter 5 we present how karyotypes are currently represented and how they can be represented ontologically. First, we discuss the karyotype components and then present the motivating problem; the current string representation of karyotypes is not computationally amenable, therefore karyotypes of diagnostic importance, can be hard to parse, validate and query. Here, we discuss the modelling decisions made for the ontological modelling of karyotypes with numerous examples taken from the ISCN2013, our handbook on karyotypes.

- In Chapter 6 we investigate the scaling performance of The Karyotype Ontology and test three different representations of the `affects` restriction by generating multiple versions of The Karyotype Ontology. Using the HermiT reasoner [137], we found that the ontology can comfortably scale and that

the best way to model the `affects` relation is dependent on the number of karyotypes.

- In Chapter 7 we discuss the five stage methodology used to build a computational model of mitochondrial disease. The first three stages, collectively known as the knowledge acquisition stage, is used to identify related terms and competency question. After, we discuss the modelling decisions made for the first iteration of the ontology. This includes the incorporation of existing structured data from databases as well as the refined terms manually obtained from a corpus of papers.

- In Chapter 8 we investigate the application of a pattern-driven and programmatic approach to ontology engineering on an existing bio-ontology, SIO.

- In Chapter 9 we introduce a novel classification of localised patterns as existing classifications of ODPs are insufficient in the classification of localised patterns. In addition, we provide basic statistics of localised pattern usage in our three Tawny-OWL projects.

- In Chapter 10 we summarise the work presented in chapters 2 to 9, specifically highlighting their relation to the original research questions as well as highlighting the limitations and potential extensions of each model. This chapter ends with a discussion as to how this work will aid the ontology community.

2

Background

Contents

2.1 Scientific knowledge 14
2.2 What is an Ontology? 18
2.3 Technologies and Methodologies 21
2.4 Summary 24

2.1 Scientific knowledge

Bioinformatics is the application of computing technology to biology, i.e. the study of life and living organisms. Biology, like all sub-disciplines of science, is a knowledge-rich subject that involves the management of knowledge. In the past, biological datasets were relatively small in size, but represented extremely complex knowledge. However the quantity of this data is continuously increasing, largely due to new experimental technologies, thus requiring biologists to join "the big-data club" [86]. In this thesis we will focus on two areas of biology: mitochondria and karyotypes.

The first biological domain is mitochondria (mitochondrion singl.): complex organelles found in most eukaryotic cells. Mitochondria are an important part of human biology due to their prominent role in the production of Adenosine TriPhosphate (ATP) through respiration which provides usable energy for the cell. Similar to other organelles mitochondria have their own independent genome, which is known as mitochondrial DNA (mtDNA), and is cytoplasmically inherited; any abnormalities in the mtDNA can result in mitochondrial disease, such as Mitochondrial Encephalomyopathy, Lactic Acidosis, and Stroke-like episodes (MELAS), which was first classified in 1984 [114]. MELAS syndrome symptoms can include muscle weakness, migraines, loss of appetite, unexplained vomiting, seizures and so on [29]. This syndrome is caused by mtDNA point mutations where the most common is m.3243A>G [47]. Approximately 236 in 100,000 people have MELAS with this mutation [85]. The mutations in the mtDNA impairs the ability of the mitochondria to produce proteins and continue its natural functions. Like all mitochondrial diseases, MELAS is currently incurable, however the symptoms can be treated. The link between mitochondria and disease is a major research area as mitochondrial diseases can give rise to a wide range of symptoms, in almost any organ or tissue, at any age, with any mode of inheritance [22]. As a result, there is currently no standard and simple way to diagnose and differentiate between mitochondrial diseases.

The mitochondrial disease domain knowledge is a classic example of most biological knowledge. Firstly, the knowledge can be found in numerous sources (e.g. published papers and online databases) and in a variety of formats (e.g. "free text" and

Comma Separated Values (CSV)). Furthermore, the community's understanding of mitochondria and mitochondrial disease is incomplete; research is actively being conducted throughout the world, so the quantity of biological knowledge is continuously growing. Lastly, the knowledge is multi-scaled (i.e. can exist of a range of granularity) according to its biological organisation of life (e.g. atoms and molecules), which increases complexity.

The second biological domain we analyse is the karyotype; this is a description of the chromosomes present in a cell, describing their number and the presence of abnormalities (if any). Karyotypes are important for their diagnostic application, as they describe chromosomal abnormalities which can cause a variety of genetic disorders.

One such disorder, Tuners Syndrome (aka Ullrich-Turner Syndrome), was first classified in 1938 [154] and occurs approximately 1 in 2500 people [146]. This syndrome is caused by monosomy X, i.e. the absence of one chromosome X in female individuals [38] (see Figure 5.2a). The symptoms of Tuners Syndrome include short stature, neck webbing, low hairline and so on [30]. Like MELAS, there is no cure for Tuners Syndrome though the symptoms can be treated [91].

Human karyotypes are normally represented using a string, as defined by the International System for human Cytogenetic Nomenclature (ISCN) [135] (see Section 5.2). ISCN Strings state: the number of chromosomes; the sex chromosomes; and any abnormalities, such as inversions, deletions, that occur in the chromosomes. For example, a female with Tuners Syndrome (and no other abnormalities) is represented as *"45,X"*. However these strings can be hard to parse, validate and query as they are not computationally amenable. For further background on karyotypes and current representation, see Section 5.2.

Unlike mitochondria, the biological knowledge of karyotypes is bounded, because the existing interpretation of the domain is already mature; the ISCN has undergone numerous, rigorous and laborious revisions to become the standard manual for representing karyotypes. However the number of karyotype instances is continuously growing, such that the karyotypic knowledge is outstripping our ability to maintain and analyse these karyotypes; we need some way of enabling this.

One way of making biological knowledge easier to understand, define, quantify, visualise or simulate is through the creation of scientific models. There are numerous modelling technologies we could use to model this complex knowledge; in this thesis, we will be using ontologies to model our two biological domains.

The reasons why we are using an ontology is that they [102]:

- Are already widely used in Life Sciences (e.g. The Gene Ontology (GO) [5] and The Experimental Factor Ontology (EFO) [79]). As introduced In Section 1.1, BioPortal[1] [101], which at this time of writing (2014) contains 384 ontologies. In addition, Gene Ontology Annotations (GOAs) [20] are commonplace within biological databases, such as The UniProt Knowledge Base (UniProtKB) [3]

- Can handle multi-scale definitions and in fact are being built and used in a multi-scale manner [130]. This will be helpful in the defining of mitochondrial disease where the knowledge can be defined according to its biological organisation of life (e.g. atoms and molecules)

- Can handle partial knowledge i.e. incomplete knowledge. This will be of interest to the karyotypes domains as there are situations when were are unable to determine the origin segment or an exact breakpoint loci (see Section 5.4.5)

- Enable common shared understanding of the structure of knowledge, in a form readable to both humans and machines [143]. Resources that have the same underlying ontological model can be aggregated then queried

- Can be built with standard technology e.g. The Web Ontology Language (OWL), a subset of first order logic, as declared by World Wide Web Consortium (W3C) in 2004 [87] (see Section 2.3). As well as ensuring compatibility with other ontologies, it encourages development in the area

- Enable reuse of domain knowledge. Rather than "reinventing the wheel", developers can import other existing ontologies to extend their ontology

[1] `http://bioportal.bioontology.org/`

- Make domain assumptions explicit. As discussed in this section, there is no standard and simple way to diagnose and differentiate between mitochondrial diseases. In explicitly defining these we hope to classify and clarify mitochondrial disease by their symptomatic and/or genomic definition

- Separate domain knowledge from operational knowledge. A biologist does not necessarily need to understand the computational interface that encapsulates the ontology

- Analyse domain knowledge. Using an ontology for a specific task has been proven to be successful when classifying phosphatase proteins [166], or comparing annotation similarity with sequence similarity [69]

- Have shown impressive progress over the years, as shown in Table 2.1. Something that was slow and simple in 1995 has become reasonably acceptable in more recent times. This is due to the improvement in computers, ontologies and reasoners (this is discussed in more detail, in Section 2.2)

In this section, we have described the scientific knowledge involved in two biological domains: mitochondria (specifically mitochondrial disease) and karyotypes. In order to model these two disciplines, we aim to use ontologies, thus in the next section, we provide the fundamentals of ontologies.

2.2 What is an Ontology?

Originally used in philosophy, the term ontology was adopted by Artificial Intelligence (AI) researchers, who created ontologies as computational models and viewed them as a form of applied philosophy. Perhaps the most well-known definition of an ontology from a computer science perspective is attributed to Tom Gruber, who defines an ontology as an explicit and formal specification of a conceptualisation [49]. In this thesis, we define an ontology as a model of objects or processes in the real-world that captures knowledge about a domain by describing these objects as well as the relationships between these objects.

These objects and relationships can be formally represented in an ontology by using three main components [70]:

1. Individuals (or Instances) which represent the objects in the domain of interest (e.g. `ThisThesis` and `Jennifer`)
2. Properties (or Slots) which represent the link between two individuals (e.g. `Thesis hasAuthor Jennifer`)
3. Classes (or Concepts) that represent sets containing the individuals that share common features (e.g. `Document` and `Person`)

Based on the logical structure of an OWL ontology, the model will allow the use of (semantic) reasoners for inference. Two common example reasoners are ELK [170] and HermiT [137]. As discussed in [132] we use reasoners to:

1. check consistency i.e. there exists a model of the ontology `O` that satisfies all the axioms of `O`
2. check satisfiability [131] i.e. for each named class `N` in ontology `O` there exists a model of `O` that satisfies all the axioms of `O` and has an instance of `N`
3. automatically build the ontology hierarchy i.e. for any two named classes `A` and `B` of ontology `O`, there exists a model of `O` that satisfies all the axioms of `O` and an instance of `A` is also an instance `B`

In addition, they can be also used for basic querying of the ontology through the use of defined classes.

Nowadays, reasoners apply two main approaches to reasoning: consequence-driven and tableau-based. Consequence-driven reasoning uses deductive rules in order to infer logical consequences (or entailments) using the axioms of an ontology and other derived axioms. The ELK reasoner is an example reasoner that uses the consequence-based approach. In contrast, tableau-based reasoning uses completion rules to individuals to construct and extend a model that satisfies all of the axioms in the ontology. They were developed for more expressive profiles, while ELK was built for the OWL EL profile (see Section 2.3). The HermiT reasoner is an example of tableau-based reasoner.

In general, reasoners perform well, as shown in Table 2.1 [59]; through the years, the time taken to reason over the ontology has generally decreased. Despite this, reasoning time can still be rather unpredictable and can increase dramatically, particularly if the ontology is computationally complex.

Table 2.1: Ontology Reasoning Time - Taken from Oxford Research Overview (2012) Ian Horrocks, UK Ontology Network 2012

Year	Size	Time(s)
1995	3,000	>>109
1998	3,000	300
2005	30,000	30
2008	30,000	<1
2008	400,000	45
2011	400,000	4

Ontologies can be classified in a variety of ways; the common classification focuses on the scope or domain granularity [128]. *Upper level* [55] (or Top Level) ontologies (e.g. The Basic Formal Ontology (BFO)[2], Dublin Core (DC) ontology[3] and The Semanticscience Integrated Ontology (SIO) [33]) are generic ontologies that are encapsulate various domains. Their main purpose is to allow the semantic interoperability of ontologies. *Reference* ontologies (e.g. Foundational Model of Anatomy (FMA) [127],

[2] `http://ifomis.uni-saarland.de/bfo/`
[3] `http://dublincore.org/`

The Systematized Nomenclature Of MEDicine Clinical Terms (SNOMED CT)[4] and NCI Thesaurus[5]) are ontologies that encapsulate community knowledge about a domain. In Chapter 7, we discuss the steps taken to build a reference ontology of mitochondria from the knowledge source of published papers. Similarly, *domain* ontologies (e.g. ChEBI [52]), also model the knowledge about a domain, but the knowledge contained is user specific rather than community orientated. However despite this difference, the term *domain ontology* has also been known to mean any ontology that focuses on one domain, regardless of whether the knowledge is community or user-specific. Lastly, *application* [80] (or Local) ontologies (e.g. EFO [79]) are ontologies that are built for a specific task or purpose, such as annotation. In Chapter 5, we discuss the construction of The Karyotype Ontology, an ontology built to replace the current representation of karyotypes.

Ontologies are used widely in life sciences for many purposes, including instance classification, schema reconciliation, or as a controlled vocabulary [145]. Of these, classification is perhaps of most interest to us. An example of this is demonstrated by GO in [4]. Further, with the use of a reasoner, we can find biological significance in conjunction with our built ontology. An example of this is demonstrated in work undertaken in [166] on the PhosphaBase Ontology [167, 168]. Here, an ontology was used to explore novel proteins and in the process refine an existing classification.

In this section, we have briefly described the history, components, types and usage of ontologies. In the next section, we discuss Semantic Web technologies and methodologies related to ontology engineering.

[4] `http://www.ihtsdo.org/snomed-ct`
[5] `http://ncit.nci.nih.gov/`

2.3 Technologies and Methodologies

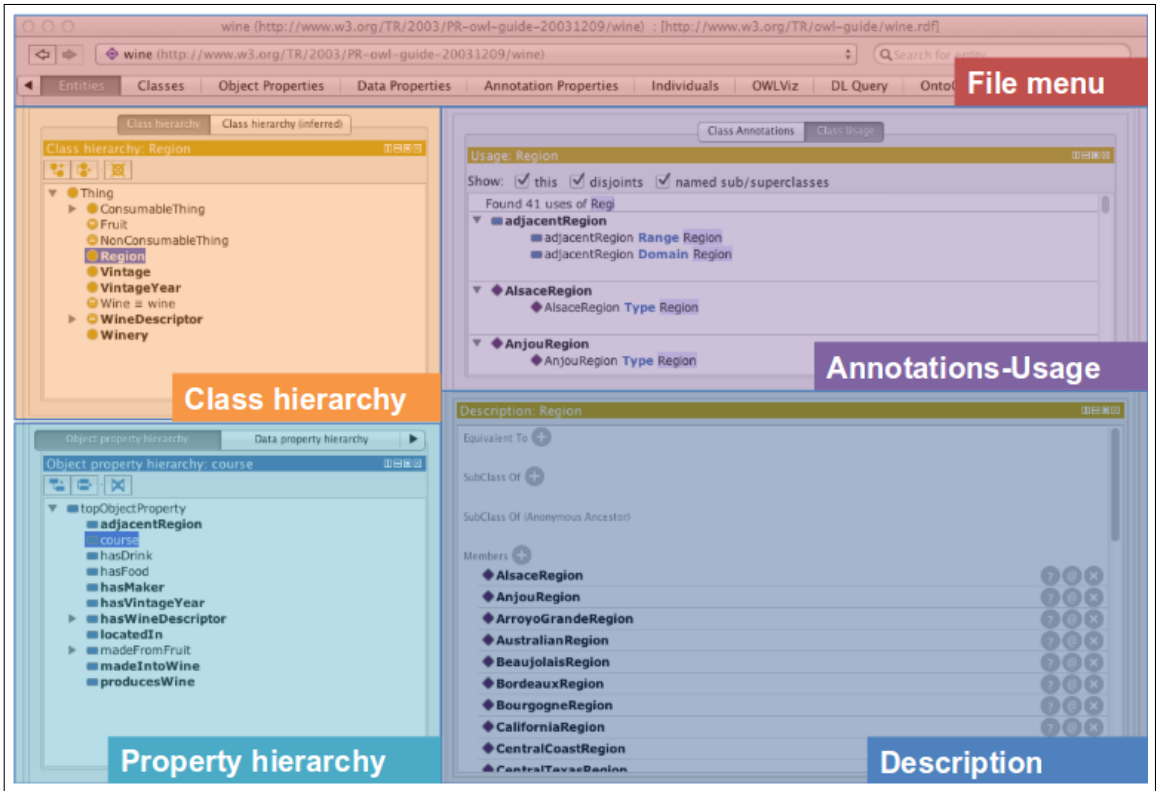
While there have historically been many ontology languages (e.g. DAML+OIL [141]), in bioinformatics the two most common are OWL and the OBO flat file format. OWL is a standard for the Semantic Web since 2004 [87], as recommended by the W3C[6], the international body which defines web standards. OWL1 has since been revised to version OWL2 and was released in 2009 [160]. There are a number of variants of OWL that vary in the use of the expressiveness of their language. Perhaps the most common is to use just subsumption and existential restrictions (i.e. the OWL EL profile) of which GO is a well-known example, while The Simple Knowledge Organization System (SKOS)[7] is an example of using the OWL Full profile. In addition, OWL supports a variety of syntax, such as OWL/XML [96] and Manchester Syntax [58]. For more information on the different OWL syntaxes, see Section 3.2.

There are various bespoke ontology editors available for the building of OWL ontologies, however the most popular is the integrated ontology development tool, Protégé [161]. Protégé[8], is a free, open-source desktop platform that provides users the ability to create and manipulate OWL ontologies. This popularity is most likely due to its user friendly interface and visualisation capabilities that encourage the natural exploration ontology (see Figure 2.1 [157]). Though this editor is tailored to OWL ontology development, Protégé also supports the OBO flat file format.

The OBO flat file format, first developed in 1999 for the construction of GO[9], is an ontology language which supports a subset of the expressivity of OWL and has been extended to support meta-data modelling [99]. The motivation of the OBO flat file format was to provide an extensible, easy to read and parse syntax that contained little redundancy. A popular ontology editor for Open Biomedical Ontologies (OBO) is OBO-Edit. Originally created for the editing of GO, OBO-Edit [27] has since been extended to handle other OBO.

In the past, there was a significant community divide between those using the OBO

[6] http://www.w3.org/Consortium/
[7] http://www.w3.org/2009/08/skos-reference/skos.rdf
[8] http://protege.stanford.edu/
[9] http://www.geneontology.org/



©Taken from Constructing Conceptual Knowledge Artefacts: Activity Patterns in the Ontology Authoring Process by Markel Vigo, Caroline Jay and Robert Stevens[157]

Figure 2.1: Figure showing the core areas of the Protégé interface.

flat file format and those using OWL [43]. However, over time, a desire for interoperability has meant that a standard mapping between the two has been developed [150].

While there is no standard way of engineering ontologies, there are a variety of methodologies that exist, some of which are analogous to software engineering methodologies. For example V-model [144], Spiral [51], Waterfall and Iterative-Incremental [147]. Furthermore, this parallel between software engineering and ontology engineering is not limited to methodologies. For example, originally popularised in the context of software engineering [40], SDP has been recast into Ontology Design Patterns (ODPs): formal, reusable and successful solutions to recurrent ontology modelling problems [35]. The main usage of ODPs is to help with the construction of ontologies while avoiding common mistakes [93]. Two well-known ODPs are value partition [121] and the sequence pattern [31]. ODPs have been supported by environments like Protégé through the use of graphical wizards[10].

As well as ODPs, which address generic concerns cross-cutting a number of domains,

[10] `http://protegewiki.stanford.edu/wiki/Protege_Wizards`

a need has also been recognised for patterns within a single ontology. One mechanism for expressing these patterns is The Ontology Pre-Processing Language (OPPL), a pre-processing language, also available as a Protégé plugin, which can be used to automate addition or transformation of ontology terms derived by a declarative rule language [36]. A similar idea is found with "Safe Macros", where patterns are expressed as annotation properties within the ontology, which may be expanded to logical axioms with a post-processor [100]. Other tools use patterns to leverage alternative data entry environments, generally spreadsheets. For example, RightField [169] and Populous [62] enable constrained data entry using an Excel spreadsheet, and then use OPPL to expand this data into OWL expressions. Similarly, Quick term templates [126] uses spreadsheets and the mapping language, Mapping Master (M^2) [105].

However, while this Graphical User Interface (GUI) based interaction is of benefit to some users (e.g. biologists that lack sufficient knowledge to code), some users (e.g. programmers) find these tools to be time-consuming and error-prone, especially when there are major modelling changes. Therefore, a number of text based/programming environment tools for ontology construction have been developed. Examples include Thea-OWL [156] (Prolog) and InfixOWL [106] (Python), both of which can generate OWL. These tools enable the development of ontologies within a programmatic environment. These also provide a mechanism for the use of patterns thus repetitive ontology construction tasks can be automated.

In this section, we have briefly described various ontology languages and ontology editors. In the next section, we discuss what we have learned from our research and highlight the gaps in this research and their relation to the research questions.

2.4 Summary

This section provides an analytical review of the existing ontological and scientific research discussed in this chapter; we use this review to identify three research questions that this thesis should answer.

As discussed in Section 2.1, the current string representation of karyotypes (as defined by the ISCN) is not computational amenable. Specifically, when representing homologous chromosomes the ISCN specification underlines the chromosome number and this underlining cannot be represented in American Standard Code for Information Interchange (ASCII). Therefore these karyotypes of diagnostic importance can be hard to parse, validate and query. Thus, we aim to build an ontology that will represent the underlying biological knowledge contained in the ISCN.

In Section 2.3, we discuss a variety of technologies and methodologies that could be used for the building the ontology. In addition, we have seen that with the use of patterns we can enforce consistency throughout ontology thus potentially decrease the number of errors. However, we would like to potentially improve on this by programmatically applying these patterns. Thus we identify the first research question to be "How can we build a computational representation of the ISCN using a pattern-driven and programmatic approach?" (RQ1).

In Section 2.1, we discuss the biological knowledge contained in two biological domains; karyotypes and mitochondria[11]. From this we learn that the karyotypic biological knowledge is a bounded and mature, while the mitochondrial biological knowledge is distributed, incomplete and multi-scaled. Therefore, while the application of this approach to build a computational representation of the ISCN is useful (RQ1), we must also show that the approach can be used to model mitochondrial biological knowledge. By producing useful computational artefacts for karyotypes and mitochondrial domains we provide evidence that the approach is potentially applicable to other domains. Thus our second research question asks "Can we apply this approach to model new areas of biology and produce useful computational artefacts?" (RQ2).

[11]To our knowledge, neither of these domains have an existing computational representation.

As discussed in Section 2.3, the use of a programmatic environment to build ontologies (with or without patterns) is not novel. In addition, the benefits of using of patterns to build ontologies have been discussed previously [17, 18]. However, to our knowledge there has been no research that comprehensively explores the advantages and benefits of using a pattern-driven and programmatic approach to ontology engineering. Thus our last research question asks the question "What are the advantages and benefits of this approach to ontology engineering?" (RQ3).

We address these questions through the use of a novel environment Tawny-OWL, which was built in parallel with and motivated by the work described in this thesis.

In the next chapter, we briefly introduce Tawny-OWL, Clojure (which Tawny-OWL is built with) and the idea of using a programmatic environment for constructing ontologies, by focusing on exemplars from The Pizza Ontology.

3

Tawny-OWL

Contents

3.1 Introduction 28
3.2 The genesis of Tawny-OWL 29
3.3 Tawny-OWL overview 32
3.4 The many names of an entity 40
3.5 Clojure overview 43
3.6 Summary 45
3.6.1 Presentation in this thesis 45

3.1 Introduction

As the main tool used in this thesis, the majority of the examples will be in Tawny-OWL format, unless specified otherwise. The syntax of Tawny-OWL [74] is based on the Manchester Syntax [58] and therefore should be relatively straight-forward to both ontologists and non-programmers. However to further aid the reader's understanding of these Tawny-OWL exemplars, this chapter is used to describe basic functionality and highlight important aspects of Tawny-OWL. This means that there is no research in this chapter; instead we present further necessary thesis background information.

Tawny-OWL[1] is a library written in Clojure, a dialect of Lisp, that wraps the OWL API. The basic syntax of Tawny-OWL is a frame-based syntax such that each entity is defined by a Clojure function and a frame is defined by a Clojure keyword. The tool development was motivated by the karyotype work described in Chapter 5, but is not specific to this domain and can be used in other domains. Tawny-OWL is used as a textual interface for ontology construction and enables a fully programmatic way of building an ontology, such that entities can be generated from simple data structures and patterns in Clojure (see Chapter 4). It also has the ability to interoperate with external reasoners and make use of unit testing.

In order to describe different aspects of Tawny-OWL, we will focus on exemplars from The Pizza Ontology, a well-known ontology used by bioinformaticians and users of Protégé [122]. However this chapter is not comprehensive user documentation of Tawny-OWL. Instead, this chapter is used to discuss the basic functions and highlight important points required to understand the exemplars found in this thesis. For a more detailed discussion on Tawny-OWL and its features please refer to the project repository[2] and associated paper [74].

[1] Developed by Dr Phillip Lord, Newcastle University.
[2] `https://github.com/phillord/tawny-owl/`

3.2 The genesis of Tawny-OWL

In this section, we discuss the ideology of Tawny-OWL. We briefly review four ontology syntaxes to see which syntax is beneficial for the quick editing of OWL ontologies, thus highlighting why Tawny-OWL is based on the frame-based Manchester Syntax.

While RDF/XML [41] is the standard OWL2 Syntax [140], the complexity of this syntax means that the quick editing of OWL ontologies can be problematic. Other syntaxes are available such as OWL/XML [96], Functional Syntax [97] and Manchester Syntax [58]; see Table 3.1 for a description and example of each syntax.

Table 3.1: Table showing a variety of syntaxes available to define ontologies. Examples for each syntax are shown in Listings 3.1 to 3.4; in each we define an ontology and a class. The preambles and prefix definitions have been elided.

Name of syntax	Example	Brief description
RDF/XML	Listing 3.1	The W3C recommended syntax for storing OWL2 ontologies is the RDF/XML Syntax that represents the RDF graph as an XML serialisation.
OWL/XML	Listing 3.2	As the name suggests, this syntax is an XML-based syntax that represents OWL information as a regular XML serialisation. This syntax is designed to support processing with (off-the-shelf) XML tools.
Functional Syntax	Listing 3.3	This syntax is a high-level syntax which closely follows the formal structure of an OWL ontology. This syntax evolved from the Abstract Syntax and Concrete Abstract Syntax.
Manchester Syntax	Listing 3.4	The Manchester Syntax is a human readable compact DL syntax, that is generally easier for non-logicians to read.

The two XML-based syntaxes are not convenient for a human to write, because of their relative verbosity as well as the necessity for balancing open and close tags.

The functional syntax is more convenient to type, although it has a potentially deeply-nested parenthetical structure; the use of parentheses for scoping an ontology means that the open and close parentheses can be separated by many screens.

```
<rdf:RDF ...>
    <Ontology rdf:about="http://www.ncl.ac.uk/pizza"/>
    <Class rdf:about="&piz;Pizza"/>
</rdf:RDF>
```

Listing 3.1: RDF/XML Syntax.

```
<Ontology ontologyIRI="http://www.ncl.ac.uk/pizza" ...>
  <Declaration>
    <Class IRI="#Pizza"/>
  </Declaration>
</Ontology>
```

Listing 3.2: OWL/XML Syntax.

```
Ontology(<http://www.ncl.ac.uk/pizza>
Declaration(Class(piz:Pizza))
)
```

Listing 3.3: Functional Syntax.

```
Ontology: <http://www.ncl.ac.uk/pizza>
Class: piz:Pizza
```

Listing 3.4: Manchester Syntax.

Manchester Syntax, on the other hand, was explicitly designed to be user-friendly and for presentation to users.

The Manchester Syntax, developed at The University of Manchester, was originally released in 2006 for OWL1 ontologies [56]. Since then there have been revisions to allow the construction of OWL1.1 [57] and OWL2 [58] ontologies. The motivation of Manchester Syntax was to provide a less verbose OWL syntax to users who did not have a Description Logic (DL) background. This was accomplished by blending principles from the OWL Abstract Syntax [113] and the compact German DL Syntax. The Manchester Syntax, like the Abstract Syntax, is frame-based such that all information about an entity is grouped into a single construct; this differs, for instance, from the eXtensible Markup Language (XML) syntaxes which use groupings based on the underlying axioms. Generally a frame-based syntax follows the format shown in Listing 3.5.

```
Entity: IRI
   Frame:
      Value(s)
```

Listing 3.5: Basic frame-based format of Manchester Syntax.

Interestingly, each entity is represented in a stanza, something Manchester Syntax shares with the OBO flat file format (see Listing 3.6); the latter was also designed for direct editing. As the syntax of Tawny-OWL also needs to be user-friendly and easily readable, Tawny-OWL incorporates a variant of Manchester Syntax frame syntax.

```
id_space: pizza http://www.ncl.ac.uk/pizza#
[Term]
id: Pizza
```

Listing 3.6: The OBO flat file format style.

Tawny-OWL is a Clojure library, which is a dialect of Lisp and its syntax also reflects this. Clojure syntax consists of parenthesis delimited lists[3], which define *expressions*[4] that contain *elements*. Elements of Clojure can be literals (e.g. strings, numbers and keywords), symbols, or collections (e.g. lists, vectors). Literals evaluate to themselves while symbols evaluate to their values. The first element of an expression is usually a function. For example in Listing 3.7, the `+` symbol is a `clojure.core` function which returns the sum of numbers. Using a Read-Eval-Print Loop (REPL), the expression is evaluated to return the number value 3. For more information on Clojure syntax, see Section 3.5. In the next section we see how we have combined the two.

```
user=> (+ 1 2)
3
```

Listing 3.7: An example of basic Clojure usage.

[3]Earlier, we criticised the deep nesting of parenthesis with the functional syntax, something which would appear to also apply to Tawny-OWL. However, Tawny-OWL has been designed to avoid parenthesis where possible. Additionally, as it uses a derivative of Lisp syntax, we can reuse Lisp editing tools such as Par Edit which largely manages a balancing and nesting of parenthesis automatically.

[4]Also called s-expressions, sexprs or sexps in the Lisp literature.

3.3 Tawny-OWL overview

Generally, the basic syntax of Tawny-OWL is an extension of Clojure that blends the frame-based Manchester Syntax with Clojure such that an entity is defined by a Clojure function, a frame by a Clojure keyword, and a value by a Clojure expression or element. Continuing with our syntax exemplars (see Listings 3.1 to Listing 3.4), the equivalent Tawny-OWL expressions to define an ontology and a class is shown in Listing 3.8.

```
(defontology pizzaontology
  :iri "http://www.ncl.ac.uk/pizza"
  :prefix "piz:")
(defclass Pizza)
```

Listing 3.8: Tawny-OWL syntax.

Each of the two expressions is a list (parenthesis delimited). The "def" entity functions are Tawny-OWL functions that create a new symbol that allows us to refer to the associated OWL entities later[5].

More specifically, the `defontology` function builds on the `ontology` function, such that `ontology` creates the OWL API `OWLOntology` object and `defontology` creates a symbol, in this case `pizzaontology`. Subsequent use of this symbol will evaluate to the ontology object. The `defclass` function binds an `OWLClass` object to the `Pizza` symbol (this is discussed in more detail, later in this section).

In this exemplar we introduce two ontology frames; the `:iri` and `:prefix` keywords. The `:iri` ontology frame is used to set the Internationalized Resource Identifier (IRI) for the ontology. This IRI is saved and used as the base IRI for all entities of the ontology. This frame value is mandatory such that if no IRI value is provided then Tawny-OWL automatically generates a random IRI. The `:prefix` frame is used to set the prefix of the ontology. This frame value is also mandatory such that if none is provided the prefix is set to the name of the ontology. Unlike the IRI, the prefix has no semantic value. Using these frames we have set the IRI of our ontology to `"http://www.ncl.ac.uk/pizza"` and prefix to `"piz:"`. In Tawny-OWL, the frames

[5]They may also generate what we call a Tawny-Name annotation, which we will go into detail in Section 3.4

are not explicitly defined, instead frames are terminated by the existence of another frame or closing bracket [72].

Typically each Clojure namespace has at most one ontology defined in it, though more are possible. However multiple usage of the `defontology` function with the same symbol name, in the same namespace, causes the new ontology to overwrite the old ontology.

As briefly discussed, generally the basic syntax of Tawny-OWL is a blend of Manchester Syntax and Clojure such that an entity is defined by a Clojure function, a frame by a Clojure keyword, and a value by a Clojure expression or element. The frames were initially the same as Manchester Syntax, however these have evolved with some differences (see Section B.3).

An `OWLClass` object is defined using the `defclass` function; for an example basic class definition see Listing 3.9. The `defclass` function builds on the `owl-class` function, such that `owl-class` creates an anonymous OWL API `OWLClass` object and `defclass` creates a symbol, in this case `Pizza`. Subsequent use of this symbol will resolve to the class object.

```
(defclass Pizza
  :annotation
  (annotation label-property (literal "Pizza" "en")))
```

Listing 3.9: An example basic class definition.

The `:annotation` frame is used to add an annotation to the entity object. The `literal` and `annotation` functions are used to create an `OWLLiteral` object and an `OWLAnnotationAxiom` object, which uses the generated `OWLLiteral` object, respectively. In this example, we are simply adding an annotation with the English string value `"Pizza"` to our `Pizza` class.

In Tawny-OWL, there are various constructions for the same semantic and syntactic representation. For example, in Listing 3.9, we used the `annotation` and `literal` functions to generate the appropriate label annotation axiom, using the `rdfs:label` annotation property. This can be simplified using the `label` shortcut function. The equivalent class definition for a `Pizza` using the `label` function is shown in Listing 3.10. By default, if no language argument for a string value, Tawny-OWL

will assume that the value is English.

```
(defclass Pizza
  :annotation
  (label "Pizza"))
```

Listing 3.10: An example usage of the `label` function.

Similarly the `comment` shortcut function can be used to generate the appropriate comment annotation axiom, using the `rdfs:comment` annotation property. More examples of construction variety (using tawny broadcasting) is discussed later in this section.

So far we have seen that Tawny-OWL has the same frames as Manchester Syntax. For example the `:annotation` Tawny-OWL frame refers to the `Annotations:` Manchester Syntax frame. Similarly the `:equivalent` Tawny-OWL frame refers to the `EquivalentTo:` Manchester Syntax frame (see Listing 3.14) and the `:disjoint` Tawny-OWL frame refers to the `DisjointWith:` Manchester Syntax frame (see Listing 3.19). However there are few exceptions to the rule.

The first exception is what we term *shortcut frames*. In the construction of ontologies, it is good practice for each entity to have a natural language label and definition, modelled using the `rdfs:label` and `rdfs:comment` annotation property respectively. This is encouraged in Tawny-OWL by providing shortcut frames to generate and add the appropriate `OWLAnnotationAxiom` to an entity. These shortcut frames (i.e. `:label` and `:comment`) can be used in place of the `:annotation` frame. For example, in Listing 3.9, we used the `:annotation` frame to add a label annotation axiom to an entity, using the `rdfs:label` annotation property; this can be simplified using the `:label` shortcut frame. The equivalent class definition for a `Pizza` using the `:label` shortcut frame is shown in Listing 3.11. Similarly the `:comment` shortcut frame can be used to add a comment annotation axiom to an entity, using the `rdfs:comment` annotation property. Both of these shortcut frames are available for any entity declaration.

```
(defclass Pizza
  :label "Pizza")
```

Listing 3.11: An example usage of the `:label` frame.

The second exception regards `OWLSubClassOfAxioms`. In Manchester Syntax, super-

classes are defined using the `SubClassOf:` frame, while in Tawny-OWL, after much exploration [76], we find that we can shorten and reverse the natural language semantics while still retaining the logical semantics [76] by using the `:super` Tawny-OWL frame (see Listing 3.12). A `:sub` Tawny-OWL frame is also available, such that we can add one or more subclass(es) to the class. The `:super` and `:sub` frames are more intelligent than initially realised; depending on the entity the `:sub` and `:super` frames can be used to add `OWLSubClassOfAxiom`s to class object or `OWLSubPropertyOfAxiom`s to properties. All available class frames are shown in Table B.3.

```
(defclass MargheritaPizza
          :super Pizza)
```

Listing 3.12: An example basic class definition.

In ontologies there are a variety of restrictions available, such as existential and universal. The most commonly used restriction in ontologies is the existential restriction. In Tawny-OWL, existential restrictions are declared using the `some` function; an example existential restriction can be seen in Listing 3.13[6]. In this example, we introduce the fact that it is possible to have more than one value for a frame. This is also true for the Manchester Syntax. Here, we show that the `:super` frame has two restriction values; these restrictions are used to ensure that each `Pizza` has at least one `PizzaTopping` and at least one `PizzaBase`.

```
(defclass Pizza
  :label "Pizza"
  :super
  (some hasTopping PizzaTopping)
  (some hasBase PizzaBase))
```

Listing 3.13: Example existential restriction definitions.

In Tawny-OWL universal restrictions are declared using the `only` function. An example universal restriction can be seen in Listing 3.14.

In OWL there are three Boolean operators available: intersection; union and complement. In Tawny-OWL these are defined using functions such as `and`, `or` and `not` respectively. In OWL ontologies these operators can be used to define complex nested definitions. This is also possible in Tawny-OWL as the underlying Lisp heritage naturally supports this by the use of nested function calls. The com-

[6]In order to run these exemplars, the `tawny.english` library must be imported.

plete class definition in Listing 3.14 utilises all three Boolean operators. Here, a `VegetarianPizza2` defined class is used to find all entities that are `Pizza`s that (and) do not have either a `MeatTopping` or `FishTopping`. Other exemplars of Tawny-OWL restriction definitions are shown in Section B.1.

```
(defclass VegetarianPizza2
  :equivalent
  (and Pizza
       (only hasTopping
             (not (or MeatTopping FishTopping)))))
```

Listing 3.14: An example universal restriction definition.

Previously in this section, we briefly identified that there are various constructions with the same semantic and syntactic representation of knowledge that can be accessed using shortcut functions, such as `label` or `comment`. Similar to Manchester Syntax, Tawny-OWL is compact and hides some of the complex OWL axiomitisations; in Tawny-OWL we call this functionality "broadcasting". Broadcasting in Tawny-OWL is the variadic flattening of Clojure data structures. Most Tawny-OWL functions have this functionality, and it is also supported by entity frames.

This means that the simple example in Listing 3.15, which contains two class expressions, is semantically and syntactically similar to the exemplars in Listings 3.16, 3.17 and 3.18.

```
(defclass C :super A)
(class C :super B)
```

Listing 3.15: Broadcasting expansion of the example basic class defined in Listings 3.16, 3.17 and 3.18.

```
(defclass C :super A B)
```

Listing 3.16: Broadcasting example with two entity arguments.

```
(defclass C :super [A B])
```

Listing 3.17: Broadcasting example with one vector argument.

```
(defclass C :super A [B])
```

Listing 3.18: Broadcasting example with one entity and one vector argument.

Tawny-OWL offers a number of def entity variants that are used to define OWL entities. Table 3.2 shows an overview of available Tawny-OWL functions that define an OWL entity, and their relation to the OWL API. Example definitions of the first

two entities (i.e. ontologies and classes) have been discussed in this chapter, while the latter four entities are shown in Section B.2. All available keyword frames for each entity are shown in Section B.3.

Table 3.2: Table showing the variety of entities available to define in Tawny-OWL. Data taken from the Tawny-OWL project repository.

OWL object	Tawny-OWL function	Def form
`OWLOntology`	`ontology`	`defontology`
`OWLClass`	`owl-class`	`defclass`
`OWLIndividual`	`individual`	`defindividual`
`OWLObjectProperty`	`object-property`	`defoproperty`
`OWLAnnotationProperty`	`annotation-property`	`defaproperty`
`OWLDataProperty`	`data-property`	`defdproperty`

Continuing with the Tawny-OWL spirit of providing a concise syntax for ontology construction, users of Tawny-OWL can import the `tawny.english` shorter aliases for a subset of Tawny-OWL functions. Table 3.3 shows an overview of available Tawny-OWL function aliases. By default these aliases are not imported as the `clojure.core` namespace already utilises these symbols. For example the `clojure.core/some` function is used to find the first logical true value for any collection. Therefore, in order to import these aliases, we must first restrict the imported `clojure.core` mappings. This is done by customising the namespace (for more information see Section B.4).

By introducing variables, Tawny-OWL generally forces the developer to define them before use. However this introduces a problem. For example, when defining two classes as disjoint (see Listing 3.19), we are unable to evaluate the first expression as Clojure does not (yet) know about the symbol `B`. There are three possible solutions to overcome this problem.

```
(defclass A :disjoint B)
(defclass B :disjoint A)
```

Listing 3.19: Desired Tawny-OWL expansion for two disjoint classes.

The first solution is to refine the entity definition. Unlike the `defontology` function the `defclass` function does not overwrite existing class definitions. Instead the `defclass` (and subsequently `owl-class`) collates OWL definitions[7]. This is true for

[7]In order to "overwrite" a class definition, the entity must first be removed using the `remove-`

Table 3.3: Table showing the variety of shorter aliases available for a subset of Tawny-OWL functions. Data taken from the Tawny-OWL project repository.

Tawny-OWL function	Shortcut alias
owl-and	and
owl-or	or
owl-not	not
owl-some	some
owl-class	class
owl-import	import
owl-comment	comment
owl-max	<
max-inc	<=
owl-min	>
min-inc	>=
min-max	><
min-max-inc	>=<

all def entity variants. This means that once the required class for our restriction is defined, we can update the OWL class definition using the `refine` function, which builds on the `owl-class` function. In Listing 3.20, we update the `A` class definition to make `A` disjoint from `B`, after `B` has been declared.

```
(defclass A)
(defclass B :disjoint A)
(refine A :disjoint B)
```

Listing 3.20: An example `refine` solution to the disjoint axiomitisation problem.

The second solution is to introduce Clojure functions which will achieve what we want. In Listing 3.21 we use the explicit function `as-disjoint` which makes the OWL classes defined within the function as disjoint classes. This is a Tawny-OWL function in Clojure that was created especially for this purpose. See Section 3.5 on how to define Clojure functions and Chapter 4 for exemplar pattern implementations.

```
(as-disjoint (defclass Y) (defclass Z))
```

Listing 3.21: An example `as-disjoint` solution to the disjoint axiomitisation problem.

The last solution is to utilise strings, which in Tawny-OWL is less restrictive than symbol usage. In Listing 3.22 we use strings to refer to the disjoint class. This is

`entity` function.

very useful in a programmatic application, however this solution is prone to spelling mistakes and shows a poorer Integrated Development Environment (IDE) integration.

```
(defclass A :disjoint "B")
(defclass B :disjoint "A")
```

Listing 3.22: An example of basic class definitions using strings for classes not predefined.

Generally all Tawny-OWL functions require an OWL ontology argument. This ensures that Tawny-OWL manipulation is not available unless an ontology is provided. By default, this parameter is defined to be the current ontology. For example in Listing 3.23 we use the `defclass` function to add a class with symbol `A` to the `testontology`.

```
(defclass A :ontology testontology)
```

Listing 3.23: An example basic class definition in to ontology.

3.4 The many names of an entity

In Tawny-OWL, there are three ways of referring to an OWL entity: 1. symbol, 2. Tawny-Name or 3. IRI. Normally, these three have a direct relationship; the symbol and Tawny-Name are the same, and both are the same as the fragment of the IRI. However, they can be separated. Unfortunately, in several places of the thesis, they have to be separated, for example to cope with numeric identifiers. Here, we define each:

1. Symbol

A symbol is a Clojure element that evaluates to its bound value. These Clojure symbols are constrained in four ways: they may only contain valid Clojure characters[8]; they must not begin or end with the colon character (`:`)[9]; they must begin with a non-numeric character; and they must not redefine reserved symbols, such as `def`, `true` and `false`. Therefore Tawny-OWL ensures that non legal characters such as parenthesis, white space and forward slashes are replaced by an underscore.

2. Tawny-Name

A Tawny-Name is a string literal value; technically it need not be limited by the constraints on symbols, although in most cases in the thesis, they are.

3. IRI

An IRI is a sequence of characters that identifies and enables interaction with a resource over a network. Unlike a Uniform Resource Identifier (URI), an IRI is a generalisation that is not limited to a subset of ASCII character set; an IRI may contain characters from the Universal Character Set (UCS) such as Chinese kanji. An IRI is defined according to the Request For Comments (RFC) proposed standard [32]. Similar to the Clojure symbol, an IRI is constrained. In Tawny-OWL, an IRI is maintained using the OWL API, meaning validation is ensured internally.

[8] Alphanumeric characters and `*`,`+`,`!`,`-`,`_`, and `?`. See `http://clojure.org/reader`
[9] This is reserved for Clojure keywords.

As previously mentioned (in Section 3.3), the `defclass` creates a Clojure symbol in the current namespace. This symbol is converted to a Tawny-Name, which in turn is converted into an IRI with the Tawny-Name as its fragment. Finally, an OWL API `OWLClass` object is instantiated, and bound to the Clojure symbol which can thereafter be used to refer to the `OWLClass` object. For example the `(defclass A)` expression will generate the symbol `A`, the `OWLClass` OWL API object with (shortened) IRI value `#A`, and the Tawny-Name with value `"A"` (which is added to the OWL object) (see Figure 3.1).

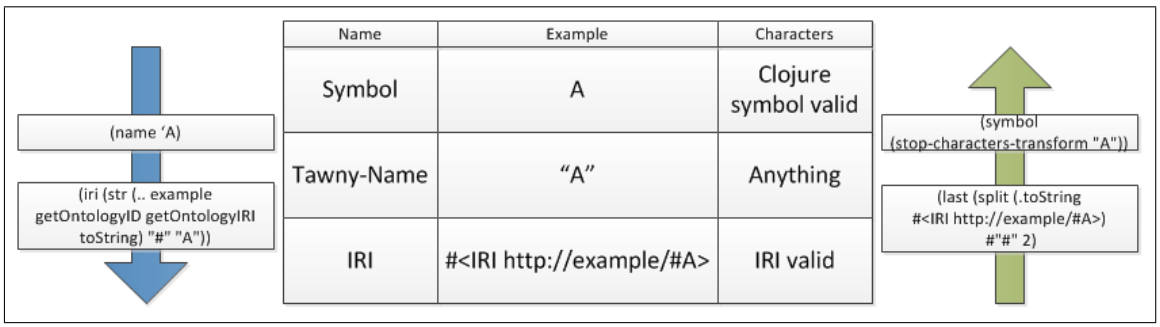


Figure 3.1: The many names of an entity and their mappings.

However the three entity names do not necessarily need to be similar. For example in Listing 3.24[10] we declare an `OWLClass` object with the symbol `clazz`, the Tawny-Name annotation value "Example OWL Class" and the (shortened) IRI value `#A`.

```
(def clazz
 (owl-class "A"
  :annotation
   (annotation name "Example OWL Class")))
```

Listing 3.24: An example of basic class definition with three dissimilar entity names.

In some ontologies, such as SIO and a variety of OBO, numeric identifiers are often used for ontology terms. While these semantic free numeric IDs are useful, as we can change the identifying label without changing the underlying semantics, in Tawny-OWL these IDs are meaningless and could possibly result in "illegal" Clojure symbols. Therefore this facility (i.e. dissimilar entity names) means that we can support the use of numeric identifiers (aka numeric IDs) in Tawny-OWL.

[10]In order for this exemplar to work, we need to exclude `clojure.core/name` and include `tawny.tawny/name` symbols from the current namespace.

When two or three entity names are similar, there is a mapping between these entity names. These mappings can be encoded as programmatic relationships as shown in Figure 3.1.

Generally each symbol maps to one OWL entity as a one-to-one mapping; however it is possible to have more than one symbol map to the same OWL entity.

3.5 Clojure overview

At this point, we know that Tawny-OWL is built on Clojure. While Clojure is one of the top 100 programming languages according to its TIOBE index [149] (we could argue Lisp at position 19, as Clojure is a derivative of Lisp)[11], it is not as well known as languages such as C and Java. Therefore this section is used as a brief overview of Clojure; we will discuss basic Clojure functions and highlight important points that are required to understand the Tawny-OWL exemplars. For a more detailed on Clojure functions please refer to the Clojure web page[12].

Clojure, designed by Rich Hickey and released in 2007, is a modern dialect of Lisp, which is used as a general-purpose language that specialises in functional programming and concurrency. As Clojure is a Lisp dialect, all advantages that apply to Lisp also apply to Clojure; for example functions are a data type just like strings and numbers meaning they can be stored in variables and be passed as arguments. Clojure is a JVM-based language which allows users full access to the mature Java Application Programming Interface (API) and interoperability with other Java APIs (such as the OWL API). Another advantage of Clojure is its interactive REPL which provides dynamic development.

Like many other derivatives of Lisp, Clojure syntax consists of parenthesis delimited lists, called s-expressions, sexprs or sexps, that are evaluated, resulting in values. S-expressions contain elements which can be literals (i.e. strings, numbers, characters, Booleans, nil and keywords), symbols, or immutable persistent data structures (i.e. lists, vectors, sets and maps). Literals evaluate to themselves while symbols resolve to their bound values (see Section 3.4). The first element of an s-expression is an operator (i.e. a function or macro) which performs operation(s) on the given arguments. In the Clojure libraries there are a variety of predefined functions and macros that have distinct functionality. For example, in Listing 3.25, the `*` function is a `clojure.core` function that returns the product of the numbers 2 and 2. Using a REPL, the s-expression is evaluated to return the value 4.

[11] Accessed on September 2014.
[12] `http://clojure.org`

```
user=> (* 2 2)
4
```

Listing 3.25: An example of basic Clojure usage.

Using Tawny-OWL, the majority of our patterns (discussed in Chapter 4) are defined as functions. In Clojure, functions can be defined using the `defn` function while associated arguments are defined as a vector in square brackets (`[ ]`). Listing 3.26 shows an example definition of a custom function using Clojure. Here we create a new function, called `square`, which multiplies the local argument `x` by itself. As shown in the exemplar, lists are expressions enclosed in parenthesis (e.g. `(* x x)`)[13], while vectors are expressions enclosed in square brackets (e.g. `[x]`).

```
user=> (defn square [x] (* x x))
user=> (square 2)
4
```

Listing 3.26: An example of basic function definition.

[13]Though it is more complex than this. There are two list data structures with different performances. It is also possible to say '(1 2 3).

3.6 Summary

As the primary tool for this thesis, here we describe Tawny-OWL, Clojure (which Tawny-OWL is built with) and the idea of using a programmatic environment for constructing ontologies, by focusing on exemplars from The Pizza Ontology. This is beneficial as this knowledge provides the foundation for how a computational representation of the ISCN can be built (RQ1).

Tawny-OWL (developed by Dr Phillip Lord[14]) is a novel environment that was built in parallel with and motivated by the work described in this thesis. As Tawny-OWL has advanced (using an agile life cycle) we have continuously evaluated the tool's fitness for purpose (as a Clojure library to build ontologies), submitted bug reports and provided minor fixes where possible. In addition, this thesis is valid documentation of the application of Tawny-OWL to four ontologies: one test ontology (The Pizza Ontology), two novel bio-ontologies (The Karyotype Ontology and The Mitochondrial Disease Ontology) and one existing bio-ontology.

As previously mentioned, as the main tool used in this thesis, the majority of the examples will be in Tawny-OWL format, unless specified otherwise. In the next section we provide an overview of the presentation of these Tawny-OWL exemplars.

3.6.1 Presentation in this thesis

All exemplars have been simplified and are not direct duplicates of the code. For example, all docstrings[15] as well as pre and post conditions[16] have been elided. In most instances, all exception handling has also been removed. To ensure a concise syntax, all exemplars use `tawny.english` shortcut aliases (e.g. `some` rather than `owl-some`). However all function names have been preserved and each concept maps to only one symbol, for easy code lookup, unless explicitly described. Lastly, all exemplars are syntax highlighted such that we can clearly identify the different core Clojure and Tawny-OWL functions, using `purple`[17] and `brown` highlighting respec-

[14] Newcastle University
[15] A docstring is a source code string literal used to document parts of the code.
[16] These are conditions that must always be true just before, or after, execution.
[17] This excludes Clojure special forms, which are highlighted in `pink`.

tively. All other Clojure functions are highlighted in `blue`. In this thesis, we use Clojure version `1.6.0` and Tawny-OWL version `1.3.1-SNAPSHOT`[18].

In the next chapter, we discuss how we can encode patterns in Tawny-OWL in order to continue the background on how a computational representation of the ISCN was built.

[18]Commit number 691ae78976

4

Pattern-Driven Development

Contents

4.1 Introduction 48

4.2 Ontology Design Patterns (ODPs) implementation in Tawny-OWL 50

4.3 Sources of data 60

4.4 Localised patterns 62

4.5 Summary 65

4.1 Introduction

During the engineering of our domain ontologies, we found many of their components to have similar textual and/or logical definitions. In software engineering, code duplication is considered bad practice, and design patterns are often used to avoid it. Therefore, we used a pattern-driven approach to model and abstract over these repetitive components. As discussed in Chapter 2, there are a variety of tools to express these patterns, such as The Ontology Pre-Processing Language (OPPL) [36] and Mapping Master (M^2) [105]. These have a major limitation; the ontological model and patterns are expressed in two different environments.

As introduced in Chapter 3, we are expressing our patterns using Tawny-OWL[1] which is built on Clojure and the OWL API. Unlike other tools, Tawny-OWL expresses the ontological model and patterns in a single syntax and environment, meaning that the two can be edited and changed together. In Listing 4.1[2], we show an OPPL example[3] and its equivalent definition in Tawny-OWL format.

```
?x:CLASS SELECT ?x SubClassOf NamedPizza
 BEGIN ADD ?x SubClassOf Thing END;
```

Listing 4.1: An OPPL example.

```
(doseq [clazz (isubclasses NamedPizza)]
 (class clazz :super thing))
```

Listing 4.2: The equivalent Tawny-OWL definition.

Previous research on patterns in ontology engineering have resulted in ODPs which are analogous to software design patterns. Many ODPs can be found in online ODP libraries such as `ontologydesignpatterns.org` [119] and the ODPs public catalog[4]. We have implemented some of these ODPs in our ontologies such as the value partition ODP [121].

[1] `https://github.com/phillord/tawny-owl`

[2] In order for this exemplar to work, we need to import the `tawny.reasoner` namespace into the current namespace.

[3] In this particular example we are only making syntactic changes to the ontology (i.e. no change in semantics), though is is not always the case. This example was taken from the OPPL website, see `http://oppl2.sourceforge.net/taggedexamples/`

[4] `http://www.gong.manchester.ac.uk/odp/html/`

In this chapter, we describe how we implement patterns in Tawny-OWL, using known ODPs as exemplars[5]. We then describe localised patterns using our exemplar ontology, The Pizza Ontology[6].

[5]These implementations are not directly taken from the Tawny-OWL and Tawny-Pizza repositories; they are simplified examples which elide some of the complexities. The full definitions are available in source.

[6]In this thesis The Pizza Ontology is used to refer to our mini The Pizza Ontology recasting (see Chapter A)

4.2 ODPs implementation in Tawny-OWL

Implementing ODPs in Tawny-OWL is relatively straightforward as it is built on a programming language. We can use the tools that Clojure provides for general programming to enable abstraction, iteration and comparison, for approximately the same purpose in The Web Ontology Language (OWL). In practice, most patterns are implemented by creating new Clojure functions. Tawny-OWL already includes an implementation of three generic patterns. The first two are so common that they are rarely recognised as patterns; the *closure* axiom and the *covering* axiom.

The underlying formal logic of ontologies means that knowledge is represented using the Open-World Assumption (OWA), which means that just because we do not say something does not mean it is not true [142]. This sometimes has undesirable logical consequences to which the closure axiom is a common solution, simulating a form of Closed-World Assumption (CWA)[7].

An example usage of the closure axiom in The Pizza Ontology can be seen in Listing 4.3[8]. Here, a `MargheritaPizza` is explicitly defined with three superclass property restrictions; two existential and one universal. These restrictions tell us that a `MargheritaPizza` has exactly two toppings: `TomatoTopping` and `MozzarellaTopping`. This is necessary because of the OWA in OWL; if we do not make this statement, then it is assumed that there may be other toppings that have not been mentioned. This means, that we cannot, for example, infer that a `MargheritaPizza` is a `VegetarianPizza` (see Listing 3.14) because it could have a `MeatTopping` in addition to `TomatoTopping` and `MozzarellaTopping`. We use a closure axiom to close the world, and state that a `MargheritaPizza` has these *and only these* toppings.

[7] `http://www.gong.manchester.ac.uk/odp/html/Closure.html`

[8] The exemplars shown in this chapter use to the `tawny.english` shortcut functions and not `clojure.core` functions. The latter namespace includes a `some` function that has different behaviour.

```
(defclass MargheritaPizza
 :super
  NamedPizza
  (some hasTopping TomatoTopping)
  (some hasTopping MozzarellaTopping)
  (only hasTopping (or TomatoTopping MozzarellaTopping)))
```

Listing 4.3: Expansion of the closure axiom example in Listing 4.4.

Explicitly defining closure axioms for numerous `Pizza`s is repetitive, time-consuming and susceptible to error. Therefore we want to encode the pattern and reuse this for each `Pizza`. The desired usage of the closure pattern (called `some-only`) can be seen in Listing 4.4. The `some-only` pattern should expand into a list that contains two existential restrictions and one universal restriction.

```
(defclass MargheritaPizza
 :super
  NamedPizza
  (some-only hasTopping
             TomatoTopping MozzarellaTopping))
```

Listing 4.4: Example usage of the closure axiom.

We can encode the closure axiom as shown in Listing 4.5. The parts of this definition are:

- We define a new function with `defn`, and the name `some-only`
- An argument list `[property & classes]`
- The `classes` argument is *variadic* and can represent any number of values
- A function body
- The `some`, `only` and `or` functions return the relevant restrictions as described in Section 3.3
- The restrictions are returned as a single `list`, containing at least one existential and exactly one universal restriction

```
(defn some-only [property & classes]
 (list (some property classes)
       (only property
             (or classes))))
```

Listing 4.5: An example implementation of the `some-only` pattern.

Another way to close the open world is with the covering axiom; here we explicitly define all the children of a particular class. An example is shown in Listing 4.6. Here, three classes and one disjoint union restriction are defined. More specifically:

- There are three classes `PizzaComponent`, `PizzaTopping` and `PizzaBase`
- Each and every instance of `PizzaBase` or `PizzaTopping` is also an instance of `PizzaComponent` (according to the subclass axioms)
- A `PizzaBase` instance cannot also be and `PizzaTopping` instance (according to the disjoint axiom)
- All instances of `PizzaComponent` must be either a `PizzaBase` or `PizzaTopping` instance (according to the equivalent axiom and the union complex class)

```
(defclass PizzaComponent
  :equivalent
  (or PizzaBase PizzaTopping))

(defclass PizzaBase
  :super PizzaComponent
  :disjoint PizzaTopping)

(defclass PizzaTopping
  :super PizzaComponent
  :disjoint PizzaBase)
```

Listing 4.6: Expansion of the covering axiom example in listing 4.7.

The desired usage of the covering axiom can be seen in Listing 4.7. Using the parent class (`PizzaComponent`) and numerous `defclass` declarations, the `as-covering-subclasses` pattern should expand, and define two subclasses and one disjoint union restriction.

```
(as-covering-subclasses
 PizzaComponent
 (defclass PizzaBase)
 (defclass PizzaTopping))
```

Listing 4.7: Example usage of the covering axiom.

We can encode the covering axiom as shown in Listing 4.8[9]. This pattern will

[9] In Tawny-OWL, covering (and disjoint) axioms are implemented as part of the `as-subclasses` function.

generate at least two subclasses and one disjoint union restriction to the parent class. The parts of this definition are:

- We define a new function with `defn`, and the name `as-covering-subclasses`
- An argument list `[parent & children]`
- The `children` argument is *variadic* and can represent any number of values
- A function body
- The `as-disjoint` functions returns the relevant disjoint restrictions as described in Section 3.3. Similarly the `as-subclasses` function returns the relevant `SubClassOf` axioms
- The `add-equivalent` function, is an explicit Tawny-OWL function used to add one or more equivalent axiom to a given entity
- The `var-get-maybe` function, is a Tawny-OWL support function used to return the value of a given var
- The `map` function is a `clojure.core` function. The `map` function applies a function (the second element) to each item in the provided collections. In this case the `var-get-maybe` function is applied to each item of the `children` collection

```
(defn as-covering-subclasses [parent & children]
  (as-subclasses children)
  (as-disjoint children)
  (add-equivalent
   parent (map var-get-maybe children)))
```

Listing 4.8: An example implementation of the `as-covering-subclasses` pattern.

Some patterns are quite a bit longer, contain other patterns and involve more than just logical patterns. The Pizza Ontology also uses the value partition ODP [121]. This is different to previous exemplars as it involves some lexical manipulation. This pattern is used to model the values of attributes. It splits up ranges, that in reality have continuous values (which can be modelled using a datatype property), into

discrete values because these are easier to work with logically. Discrete values are represented using an object property restriction. In fact, an analogous method is also used outside of the ontology community as a form of "binning", for example the Body Mass Index (BMI) (or Quetelt index) which is used to measure an individual's relative weight classification based on their mass and height. In OWL, we would encode this property such that BMI can only be an instance of either `Underweight`, `NormalRange`, `Overweight` or `Obese`, and is linked via the `hasBMI` object property.

In The Pizza Ontology, the value partition ODP is used to identify the spiciness of pizzas. Spiciness could be measured using the Scoville scale which identifies food in Scoville Heat Units (SHU) [134], a continuous numeric value between 0 and ∼2 million. In The Pizza Ontology however, spiciness is only identified as three levels of spiciness: mild, medium and hot.

An example usage of a value partition pattern in The Pizza Ontology can be seen in Listing 4.9. Here, four classes, one object property and one disjoint union restriction are defined. This example tells us that `Spiciness` can only be `MildSpiciness`, `MediumSpiciness` or `HotSpiciness`. More specifically:

- There are four classes `Spiciness`, `MildSpiciness`, `MediumSpiciness` and `HotSpiciness` and one object property `hasSpiciness`
- Each and every instance of `MildSpiciness`, `MediumSpiciness` or `HotSpiciness` is also an instance of `Spiciness` (according to the subclass axioms)
- A `HotSpiciness` instance cannot also be a `MediumSpiciness` and/or a `MildSpiciness` instance (according to the disjoint axiom)
- All instances of `Spiciness` must be either a `MildSpiciness`, `MediumSpiciness` or `HotSpiciness` instance (according to the equivalent axiom and the union complex class)
- The `hasSpiciness` object property links any given individual to individuals of `Spiciness` (according to the specified range)
- For any given individual, there can only be, at most, one individual related via the `hasSpiciness` object property (according to the functional characteristic)

```
(class Spiciness
 :equivalent
 (or HotSpiciness MediumSpiciness MildSpiciness))

(object-property hasSpiciness
 :range Spiciness
 :characteristic :functional)

(class MildSpiciness
 :super Spiciness
 :disjoint MediumSpiciness HotSpiciness)

(class MediumSpiciness
 :super Spiciness
 :disjoint MildSpiciness HotSpiciness)

(class HotSpiciness
 :super Spiciness
 :disjoint MildSpiciness MediumSpiciness)
```

Listing 4.9: Expansion of the value partition example in listing 4.10.

This value partition expansion is a lot more involved than previous exemplars and can be laborious to encode for two reasons: firstly, handling symbols where they have not been previously declared is somewhat involved in Clojure; and secondly the multiple usage of "Spiciness" string in all subclasses of `Spiciness`.

By using a simple tree-like structure Tawny-OWL should be able to expand the data input and generate the necessary entities and restrictions. The desired usage of the pattern can be seen in Listing 4.10.

```
(value-partition
 Spiciness "Mild" "Medium" "Hot")
```

Listing 4.10: Example usage of the `value-partition` pattern.

We can encode the value partition pattern as shown in Listing 4.11. The parts of this definition are:

- We define a new function with `defn`, and the name `value-partition`
- An argument list `[parent & children]`
- The `children` argument is *variadic* and can represent any number of values
- A function body

- The `str-name` function returns a string representation of the entity's name.
- The `let` function evaluates then binds the results to locally named symbols. In this case we bind the parent name to the local symbol `s`
- The `str` function returns a string concatenation of the arguments. It is this lexical manipulation that is novel in this pattern from others shown previously.
- The `object-property` and `class` functions return the relevant OWL API object, as described in Section 3.8
- The `map` function creates the name and then the class, for each item of the `children` collection
- The `as-covering-subclasses` function, as described earlier in this section, applies the relevant restrictions: one disjoint, one equivalent and at least two subclass restrictions.

```
(defn value-partition [parent & children]
 (let [s (str-name parent)]
  (object-property (str "has" s)
                   :range parent
                   :characteristic :functional)
  (as-covering-subclasses parent
   (map #(class (str % s)) children))))
```

Listing 4.11: An example implementation of the `value-partition` pattern.

Some patterns define annotation axioms rather than logical axioms. As we will see in Chapter 8, for some ontologies, many patterns affect annotations and only annotations. In Chapter 3, we saw that Tawny-OWL has already defined some annotation patterns; the `label` and `comment` functions create annotation axioms using the `rdfs:label` and `rdfs:comment` annotation properties respectively. In Tawny-Pizza, we might wish to add a Dublin Core (DC) ontology creator annotation to our pizza entities. For example, in Listing 4.6 the `MozzarellaTopping` class is defined and annotated with the creator value "Jennifer D. Warrender"[10].

[10] In order for the `dc:creator` annotation property to resolve, we need to import the DC ontology [1].

```
(defclass MozzarellaTopping
          :annotation
          (annotation
           (iri "http://purl.org/dc/elements/1.1/creator")
           (literal "Jennifer D. Warrender" :lang "en")))
```

Listing 4.12: Expansion of the creator annotation axiom example in Listing 4.13

We can simplify this annotation axiom into a more concise expression using a custom creator function. An example usage of this creator function is shown in Listing 4.13.

```
(defclass MozzarellaTopping
          :annotation
          (creator "Jennifer D. Warrender"))
```

Listing 4.13: Example usage of the creator pattern.

We can encode the creator annotation pattern as shown in Listing 4.14. The parts of this definition are:

- We define a new function with defn function, and the name creator
- An argument list [creator]
- A function body
- The dc-creator symbol refers to the dc:creator annotation property using its specified Internationalized Resource Identifier (IRI)[11]
- The annotation and literal functions return the relevant annotation restriction and literal as described in Section 3.3
- The function returns a single annotation restriction

```
(defn creator [creator]
 (annotation dc-creator (literal creator :lang "en")))
```

Listing 4.14: Example usage of the creator pattern.

The manual addition of the creator annotation to numerous entities can be laborious and error-prone. Instead we might consider adding these creator annotations automatically to all entities in the ontology, where those entities start with a given IRI. Here we show an example and explanation of how we could encode this pattern.

[11] http://purl.org/dc/elements/1.1/creator

In this case we name this iterative annotation pattern as `whodunit`. The desired usage of the pattern can be seen in Listing 4.15.

```
(whodunit pizzaontology
          "http://www.ncl.ac.uk/pizza"
          "Jennifer D. Warrender")
```

Listing 4.15: Example usage of the `whodunit` pattern.

We can encode the `whodunit` function as shown in Listing 4.16[12]. The parts of this definition are:

- We define a new function with `defn`, and the name `whodunit`
- An argument list `[o piri & creators]`
- The `creators` argument is *variadic* and can represent any number of values
- A function body
- The `map` function applies the `creator` function to each item in the `creators` argument to create the relevant creator annotation axiom(s). Using the `let` function this result is bound to the local symbol `a`
- The `.startsWith` is an imported Java method from the String class. Simply, the `.startsWith` method tests whether a string starts with a given prefix.
- The `.getSignature`, `.getIRI` and `.toString` methods are imported Java OWL API methods. The `.getSignature` method returns all entities that are in a given `OWLOntology` object, whilst the `.getIRI` method returns the IRI of an OWL API object and the `.toString` method returns a string representation of the IRI.
- The `add-annotation` function is an explicit Tawny-OWL function used to add one or more annotations to a given entity.
- The `doseq` function is a Clojure iterative function used to repeatedly execute the body to the filtered and bound local arguments (in this case `e`). Here, we

[12]This function excludes the annotating of the ontology, because an `OWLOntology` object is not returned in the `.getSignature` OWL API function call. Therefore an extra `add-annotation` call would be required.

are applying the `add-annotation` function to numerous entities of the ontology. The variables are filtered using the `:when` Clojure keyword such that only "when" the predicate is true does the entity bind to the local variable `e`

```
(defn whodunit [o piri & creators]
 (let [a (map creator creators)]
  (doseq
   [e (.getSignature o)
    :when (.startsWith
           (.toString (.getIRI e))
           piri)]
    (add-annotation e a))))
```

Listing 4.16: An example implementation of the `whodunit` function.

Overall, in all of these encoded generic patterns, we find patterns contain OWL constructors and that OWL entities are present only as variables.

4.3 Sources of data

In the patterns described so far all the values are specified in the source. However, it is also possible to take values from other places, for example from external files in different and more appropriate formats.

For example, we might wish to add Italian labels to our pizza ontology. An example of this is shown in Listing 4.17. Here we define a `MargheritaPizza` with the Italian annotation value `"Pizza Margherita"`.

```
(defclass MargheritaPizza
 :annotation (label "Pizza Margherita" "it"))
```

Listing 4.17: Expansion of the labelling of entities in Italian.

However adding labels to the source code is cumbersome, so we instead define these labels in an external properties file. An excerpt from this file is shown below:

```
MargheritaPizza=Pizza Margherita
MozzarellaTopping=Mozzarella Ingredienti
TomatoTopping=Pomodorro Ingredienti
```

By providing a valid file name, Tawny-OWL should be able to read input from the properties file and correctly assign Italian annotations to the relevant OWL entities. An example usage of this facility, which we call the `load-labels` function, is shown in Listing 4.18.

```
(load-labels pizzaontology "pizzalabel_it.properties" "it")
```

Listing 4.18: Example usage of the `load-labels` pattern.

We can encode the load labels pattern as shown in Listing 4.19. This function generates the Italian label axioms for all entities in the ontology that have an Italian translation. The parts of this definition are:

- We define a new function with `defn`, and the name `load-labels`
- An argument list `[o filename language]`
- A function body
- The `.getSignature` function returns a collection of relevant OWL API objects as described earlier in this chapter

- Similar to the `whodunit` pattern, the `load-labels` pattern uses an iterator (`doseq`) to assign the Italian labels. However, in this case the entities are not filtered but still bound to the local symbol `e`.
- The `load-props` function loads the properties file to memory as a set of key and value pairs
- The `resolve-entity`[13] function is a Tawny-OWL function used to return a string representation of the symbol that holds the OWL API object
- The `.getProperty` method is imported from `java.util.Properties` and is used to search for the associated property value for the given key
- In this exemplar the `let` function is used twice; first we bind the loaded property key value pairs to the local symbol `props`; secondly we bind the result of `.getProperty` to the symbol `l`
- The `if-not` and `nil?` functions are `clojure.core` functions used to test the complement conditional and existence of `l` respectively
- The `label` function returns the relevant annotation axiom (as described in Section 3.3), while the `add-annotation` function adds this axiom to the relevant entity

```
(defn load-labels [o filename language]
 (let [props (load-props filename)]
  (doseq [e (.getSignature o)]
   (let [l (.getProperty props (resolve-entity e))]
    (if-not (nil? l)
     (add-annotation e (label l language)))))))
```

Listing 4.19: An example implementation of the `load-labels` pattern.

With the use of the Incanter library[14] we can also read from spreadsheets (of file formats *.xls* and *.xlsx*) such that Tawny-OWL has similar capabilities to tools such as RightField [169] and Populous [62].

[13] In order for this function call to work, the `tawny.lookup` namespace must be imported.
[14] `https://github.com/incanter/incanter`

4.4 Localised patterns

As Tawny-OWL is embedded in a generic programming language, Tawny-OWL allows the expression of localised patterns. In this thesis, we introduce and define *localised patterns* as "custom patterns that are specific to the ontology" (see Chapter 9). In The Pizza Ontology, one such localised pattern is the `generate-named-pizza` pattern, where a particular pizza is defined by an enumeration of its ingredients. An example usage of the localised pattern can be seen in Listing 4.21. In this example, the input expands to a class with one universal restriction each and seven existential restrictions as shown in Listing 4.20.

```
(class CapricciosaPizza
 :super NamedPizza
 (some hasTopping AnchoviesTopping)
 (some hasTopping MozzarellaTopping)
 (some hasTopping TomatoTopping)
 (some hasTopping PeperonataTopping)
 (some hasTopping HamTopping)
 (some hasTopping CaperTopping)
 (some hasTopping OliveTopping)
 (only hasTopping AnchoviesTopping MozzarellaTopping
       TomatoTopping PeperonataTopping HamTopping CaperTopping
       OliveTopping))
```

Listing 4.20: Expansion of the localised pattern example in Listing 4.21.

Explicitly defining closure axioms for numerous `NamedPizza`s is repetitive, time-consuming and susceptible to error. Therefore we want to encode the pattern and reuse this for each `NamedPizza`. The desired usage of the pattern can be seen in Listing 4.21.

```
(generate-named-pizza
 [CapricciosaPizza AnchoviesTopping MozzarellaTopping
  TomatoTopping PeperonataTopping HamTopping CaperTopping
  OliveTopping])
```

Listing 4.21: Example usage of the `generate-named-pizza` function.

We can encode the `generate-named-pizza` function as shown in Listing 4.22. The parts of this definition are:

- We define a new function with `defn`, and the name `generate-named-pizza`

- An argument list `[& pizzalist]`
- The `pizzalist` argument is *variadic* and can represent any number of values
- Each item in the `pizzalist` argument is destructured. Destructuring is the binding of a set of variables to a corresponding set of variables. In this case, each `pizzalist` value is bound to two separate symbols; `named`, which corresponds to the first element of pizzalist, and `toppings`, the rest of the elements. Here, the `toppings` argument is also *variadic*
- A function body
- The `class` function returns the relevant `OWLClass` object as described in Section 3.3
- Once again we use the `doseq` iterator function; for each named pizza, we create the relevant OWL API class object using the local bound arguments `named` and `toppings`
- The `some-only` function returns the relevant restrictions as described in Section 4.2

```
(defn generate-named-pizza [& pizzalist]
 (doseq [[named & toppings] pizzalist]
  (class
   named
   :super NamedPizza
   (some-only hasTopping toppings))))
```

Listing 4.22: An example implementation of the localised pattern, `generate-named-pizza`.

This pattern is simple because it uses the `some-only` function (already defined in Section 4.2), to define the existential and universal restrictions. Unlike other encoded patterns shown in this chapter, you can see that the `generate-named-pizza` pattern has an embedded class definition, which is syntactically similar to our previous uses of `defclass`.

Unlike the generic patterns discussed in Section 4.2, localised patterns have parts of the patterns hard-coded with OWL entities. In the `generate-named-pizza` localised

pattern, the `NamedPizza` class and `hasTopping` object property are specific to the ontology.

The `generate-named-pizza` pattern is useful because it is variadic; it can take any number of pizza. Additionally, it uses variadic destructuring internally, enabling, for example, a `MargheritaPizza` to have two toppings, while a `CapricciosaPizza` has seven. Also it is syntactically concise, which is useful when defining pizzas with a large number of toppings. Lastly the localised pattern also ensures the consistency of definitions for all subclasses of `NamedPizza`, and supporting maintainability should we wish to change these definitions.

4.5 Summary

In this chapter, we discuss the implementation of patterns in Tawny-OWL, which fulfils part of RQ1. Patterns are simple to implement and are generally implemented as functions. Using these functions, we have started to show the many benefits of using a pattern-driven and programmatic approach to ontology engineering (RQ3). For example, it is easy to iterate over all entities in an ontology, and to interact with external files thus allowing us to separate the knowledge from the axiomitisation. In addition, where pattern definitions become complex, they can be abstracted over, hiding the technical details.

Unlike other ways of expressing patterns in OWL (such as OPPL and M^2), in Tawny-OWL the pattern is expressed in the same syntax, which means that patternised and non-patternised class definitions look similar. In addition, the patternised and non-patternised parts of the ontology are expressed in the same environment: they can be in the same file, version and can be tested alongside each other. The use of a single environment is particularly important because it enables the construction of localised patterns, and most of the ontologies discussed in this thesis, are full of localised patterns. Here, we introduce localised patterns (to the ontological community), as patterns that have a specific purpose, rather than generically for reuse in other ontologies.

In the next chapter, we discuss how we can use Tawny-OWL and its ability to encode patterns to build a computational representation of the International System for human Cytogenetic Nomenclature (ISCN).

5

Modelling Karyotypes

Contents

5.1 Introduction 68
5.1.1 Definition of terms 69
5.2 What is an ISCN String 72
5.2.1 Reviewing chromosome components 82
5.2.2 Modelling requirements 85
5.3 Design considerations 88
5.3.1 Portions of reality 88
5.3.2 A partonomic axiomitisation 89
5.3.3 The event-based change axiomitisation 92
5.4 Representing karyotypic knowledge 95
5.4.1 Modelling chromosome components 95
5.4.2 Modelling *normal* karyotypes 96
5.4.3 Abnormality breakpoints 97
5.4.4 Orientation of substitution segments 99
5.4.5 Partial knowledge 103
5.4.6 Modelling uncertainty 103
5.4.7 Multiple copies of rearranged chromosomes 104
5.4.8 Derivative chromosomes 104
5.4.9 Abnormalities involving homologous chromosomes 105
5.4.10 Constitutional anomalies 107
5.4.11 Mosaic karyotypes 107
5.4.12 Identifying the (near-)ploidy levels 108
5.4.13 Defining sex 109
5.5 Assessment 111
5.6 Summary 113

5.1 Introduction

Over the years, the number of known karyotypes has grown to over a hundred thousand. These karyotypes of clinical significance can be found in multiple research databases. There is a standard nomenclature for karyotypes, which is based on semantically meaningful strings. However this specification is outdated and not computationally amenable, which puts the quality and maintenance of this clinically important knowledge into question. Therefore, the development of The Karyotype Ontology is potentially valuable for cytogenetics by transforming collections of karyotypes into a form that is easy to query, check and maintain.

For this work, we are using an ontology to provide a strong computational and formal interpretation of the karyotype. The top-level structure of The Karyotype Ontology can be seen in Figure 5.1.

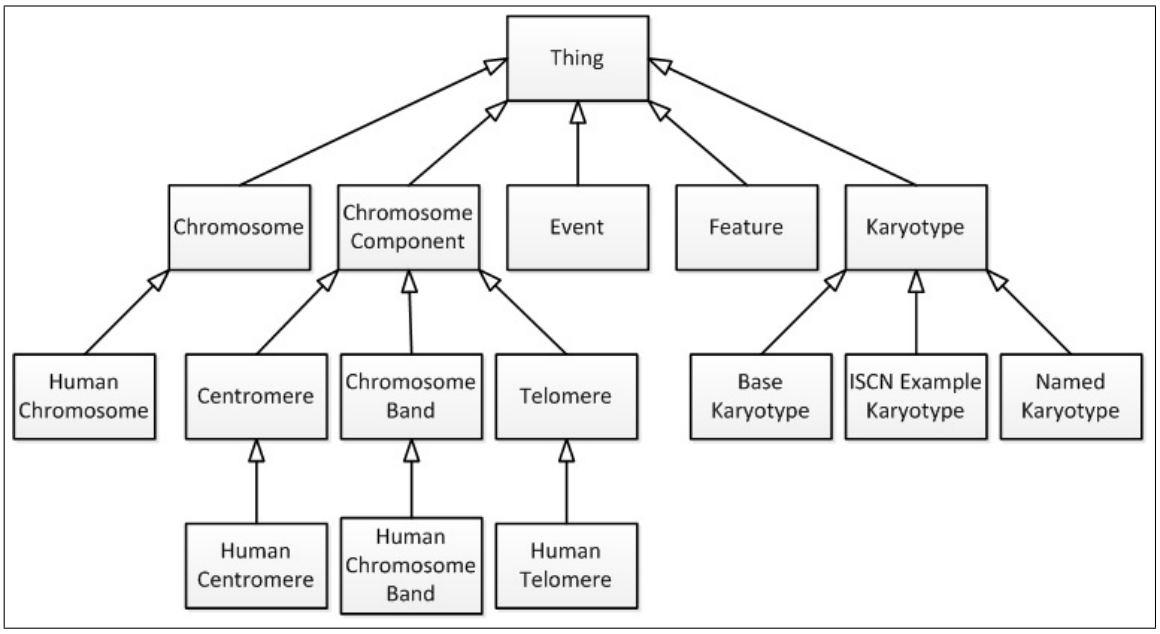


Figure 5.1: The top-level structure of The Karyotype Ontology.

For our methodology, we used features from agile development working from the ISCN specification and other external knowledge. However, even with our approach, there are many similar concepts. Therefore, we define the ontology as a series of parameterisable patterns (see Chapter 4), which expand to the full ontology [163], using Tawny-OWL (see Chapter 3). We evaluate the ontology using examples defined by the ISCN2013 [135].

Within this chapter, we describe how karyotypes are currently represented and how they can be represented ontologically. First, we discuss karyotypic terms and their usage in the cytogenetics community, including a brief review on human chromosome components. Next, we analyse the karyotype components by discussing how they can be represented in string format, as defined by the ISCN specification. From this analysis we present the motivating problem: the current string representation of karyotypes is not computationally amenable, therefore these karyotypes of diagnostic importance can be hard to parse, validate and query. This incompatibility occurs when representing homologous chromosomes; the ISCN underlines these homologous chromosomes which cannot be represented in American Standard Code for Information Interchange (ASCII) . Our analysis of ISCN identifies a variety of key requirements that our ontological representation of karyotypes should model.

Next, we discuss the ontological modelling of karyotypes with numerous examples taken from the ISCN. This discussion includes the three approaches investigated to model karyotypes: realism, partonomic and event-based change. We find the latter approach the most appropriate in creating an ontology for karyotypes, while the chromosomal components are modelled partonomically.

The later part of the chapter provides an overview of the ontology, its functionality and shows how we have fulfilled our requirements introduced in Section 5.2.2.

The corresponding code and supplementary data for the construction of The Karyotype Ontology can be found on the Project Website[1].

5.1.1 Definition of terms

Throughout this chapter, we use the following terms to describe different components of cytogenetics. Within the cytogenetics community, these definitions are freely used and are used interchangeably, even though they are not synonymous. Here we define each term so that they are not synonymous and provide the common community usage of the term:

[1] `https://github.com/jaydchan/tawny-karyotype`

Chromosome complement

The whole set of chromosomes present in the nucleus of a eukaryotic cell. This term crosses many levels of granularity and is applicable at a species, individual, cell line, or cell level. When applied at levels other than the cell it expresses the canonical complement: i.e. some cells in a cell line or individual may have differences from the canonical complement.

Species level: In a normal human, the chromosome complement consists of 46 chromosomes – 22 pairs of autosomal chromosomes (1 to 22) and one pair of sex chromosomes (XX or XY).

Karyogram

A visualisation of the chromosome complement as seen under a microscope, after chromosomes are fixed and stained. Usually a picture or photograph, the chromosomes are arranged in homologous pairs and in descending order of size.

For example, the karyogram for an individual with Tuners Syndrome can be seen in Figure 5.2a.

Ideogram (or Idiogram)

A diagrammatic representation of the Karyogram, in part or in whole. Chromosome(s) are lined up by their centromere and placed with their short arm facing up and long arm facing down. Further annotation can be included to aid classification.

Example ideograms for normal human chromosomes can be seen in Figure 5.2b[2].

This term is commonly substituted with Karyogram.

Karyotype

A description of the chromosome complement at the level of granularity seen

[2]Throughout this thesis, ideograms are used to help explain the biological theory behind karyotypes. The majority of the images were produced by Idiographica and later adapted using Microsoft Visio.

in the Karyogram and/or Ideogram[3]. The karyotype describes the number of chromosomes and the presence of abnormalities (if any).

For example, a "free text" karyotype of Tuners Syndrome is "a female individual with 45 chromosomes and monosomy X".

This term is commonly substituted with chromosome complement and an ISCN String.

ISCN String

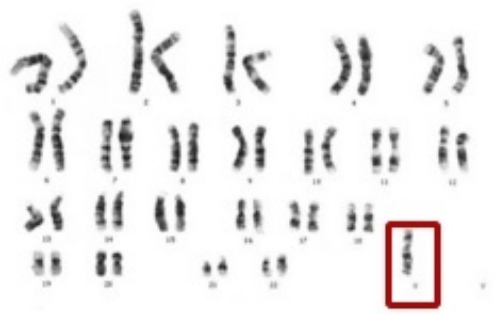
A string representation of the karyotype as defined by the ISCN. It is a subtype of karyotype.

For example, the ISCN String for Tuners Syndrome is *"45,X"*. More detailed examples are explained in Section 5.2.

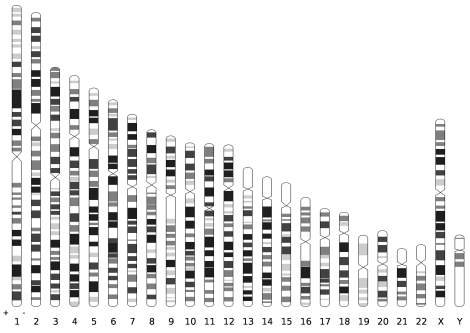
OWL Karyotype

An OWL representation of a karyotype using the concepts defined in this thesis.

For example, the OWL Karyotype for Tuners Syndrome is shown in Listing 5.11.



(a) An example Karyogram depicting a female individual with monosomy X.



(b) Ideogram: a diagrammatic representation of the human chromosomes.

Figure 5.2: Two ways of visualising the chromosome complement: Karyogram and Ideogram.

[3]the full genome is not a karyotype.

5.2 What is an ISCN String

This section provides an extensive and in depth analytic review of how the ISCN defines ISCN Strings; we use this review to identify key requirements that our ontological representation should be able to model as well as areas that could be improved or expanded upon. One thing to note is that the ISCN also describes karyotypic experimental techniques such as Fluorescence In Situ Hybridization (FISH); these techniques will not be modelled in the ontology as it is out-of-scope of this thesis[4].

The ISCN is the specification that defines an ISCN String. Initially designed for writing and printing, it was developed to address the need for an explicit nomenclature "to enable communication between workers in the field" [135]. The ISCN has a long history; early versions date from around 1960, when the emphasis was on human-to-human communication for a small number of karyotypes. This human readability is beneficial to cytogeneticists, however the number of karyotypes is increasing, outstripping our human capabilities to analyse these karyotypes. With no computational representation, we are unable to use computers to aid the validation and querying of karyotypes. Our ontological representation aims to overcome this problem.

An ISCN String represents three key concepts:

1. the number of chromosomes
2. the sex chromosomes present
3. any abnormalities that may be present

In a normal human (with no abnormalities), the chromosome complement consists of 46 chromosomes; 22 pairs of autosomal chromosomes (1 to 22) and one pair of sex chromosomes (XX or XY). These normal karyotypes are represented as *"46,XX"* and *"46,XY"* in ISCN String format for a female and male individual respectively. The first part of the ISCN String identifies the number of chromosomes present in the karyotype, while the second part explicitly identifies the sex chromosomes present.

[4]However there is no *a priori* reason to believe that we could not do this.

These two parts of this ISCN String are important and should not be mistaken as denormalised (contained redundancy that is typically used to optimise readability). For example, given *"XY"* (rather than *"46,XY"*), we are unable to determine if we are representing the *"24,XY"* haploid, *"46,XY"* diploid, *"68,XY"* triploid, or *"90,XY"* tetraploid karyotype. This uncertainty is due to the way constitutional sex chromosomes are represented in ISCN Strings (discussed later in this section). The second part of the ISCN String is also important as it tells us the sex[5] of the karyotype.

Here we identify two key requirements; it is necessary that The Karyotype Ontology be able to model the "normal" karyotypes for each ploidy level (R2) and determine the sex of a given karyotype (R8).

However not all karyotypes are normal; they can include a variety of abnormalities. There are two types of abnormality. *Numerical abnormalities* are abnormalities that affect the number of chromosomes present in the karyotype, either by gaining or losing whole chromosomes. *Structural abnormalities* are abnormalities that involve only parts of the chromosomes.

Only numerical abnormalities change the first part of the ISCN String: an absent chromosome decreases the number while a surplus chromosome increases the number. The second part of the ISCN String can be affected by both types of abnormalities; any sex chromosome that is missing or gained (and is not part of the constitutional karyotype) is removed from, or added to, this part. The third and final part of the ISCN String is a list of abnormalities, in order of chromosomal "number" (X,Y,1,2,...,22), the abnormality type (numerical, structural), then finally alphabetical order of structural abnormalities.

These abnormalities are represented in the ISCN String using symbols and abbreviated terms. For numerical abnormalities, the symbol **-** is used to represent the loss of chromosomes while + represents the gain of chromosomes (except for the sex chromosomes). For example, the karyotype for an individual born with Downs Syndrome (and no other abnormalities) is represented as *"47,XX,+21"*; a female individual that has gained one chromosome 21, results in 47 chromosomes and a

[5]Although in karyotypes, as in life, sex is more complicated then it first appears.

trisomy (three copies of) chromosome 21 (see Figure 5.3a).

Here we identify a further key requirement; it is necessary that OWL Karyotypes be able to model both types of numerical abnormalities; gain and loss (R3).

Using the number of chromosomes present, we can determine the associated (near-)ploidy or *euploid* level; so, for example, for the ISCN String *"47,XX,+21"*, has 47 chromosomes, thus is defined to be a near-diploid cell (see Table 5.1). It is therefore implicit that this cell has two of chromosome 1. This representation allows a more concise form of karyotype: so *"47,XX,+21"* rather than *"47,XX,1,1,2,2,3,3, . . . ,21,21,+21,22,22"*. As well as concision, we have the ability to define our start point. So, Tuners Syndrome is described as a diploid that has lost one chromosome (*"45,X"*), rather than a haploid that has gained 22 (*"23,X,+1,+2,. . . +21,+22"*).

Here we identify another key requirement; it is necessary that The Karyotype Ontology be able to determine the (near-)ploidy levels for a given karyotype (R9).

Table 5.1: Table showing the relationship between the modal number and number of chromosomes. Taken from ISCN2013 [135]. Further ploidy levels and in-depth details have been excluded from the table as they are out-of-scope for this thesis.

Near-ploidy level	Modal number	Number of chromosomes
Near-haploidy (23±)	n	≤34
Near-diploidy (46±)	2n	35-57
Near-triploidy (69±)	3n	58-80
Near-tetraploidy (92±)	4n	81-103

The majority of the structural abnormalities are represented using abbreviated terms. For example, an individual with an inversion abnormality uses the **inv** abbreviation. Immediately after the abbreviation, the chromosome involved in the structural abnormality, followed by any associated breakpoints (see Section 5.2.1), are specified within two sets of parenthesis **( )**. The *"46,XX,inv(6)(p22q23)"* ISCN String represents a female individual with an inversion abnormality in one of the chromosome 6; breakpoints are band `6p22` and `6q23` (see Figure 5.3b).

If more than one chromosome is involved in the abnormality, then the chromosomal information is separated by a semicolon (**;**). For example, an individual with a translocation abnormality (**t**) that involves one chromosome 3 and one chromosome

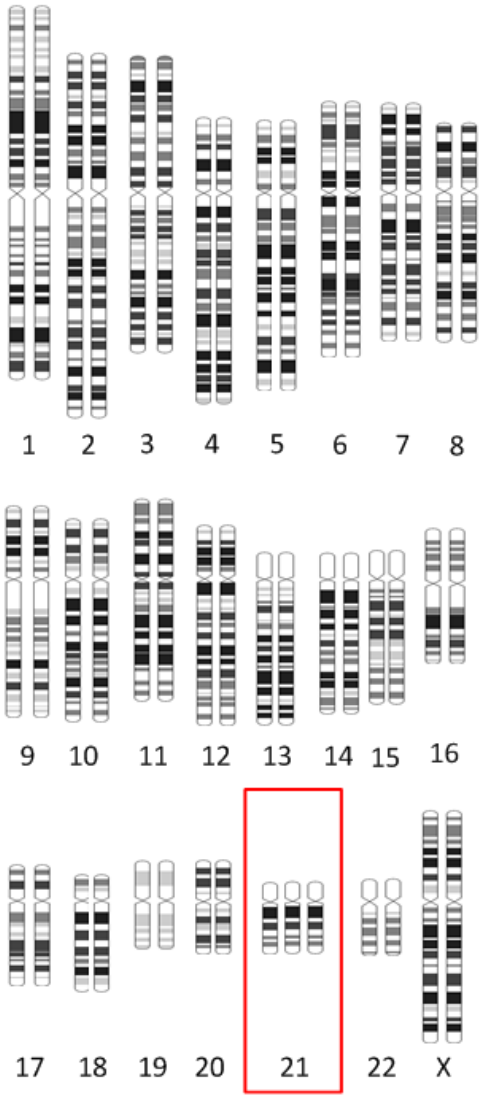
9 with associated breakpoints `3p13` and `9q21` has the ISCN String *"46,XX,t(3;9)(p13;q21)"* (see Figure 5.4).

If an abnormality involves homologous chromosomes, then one of the homologous chromosomes is underlined. For example, the ISCN String *"ins(2;2)(p13;q21q31)"* represents an individual with an insertion abnormality (**ins**) that involves two homologous chromosome 2 with associated breakpoints `2p13` in one chromosome 2 and `2q21` and `2q31` in the homologous chromosome 2 (see Figure 5.11). Although this underlining is sufficient for human-to-human (written) communication, it cannot be represented in ASCII. Further analysis of the ISCN does not show how abnormalities that affect three homologous chromosomes are modelled. This explicit identification of homologous chromosomes and associated breakpoints is clearly a necessary requirement of our ontological model (R5).

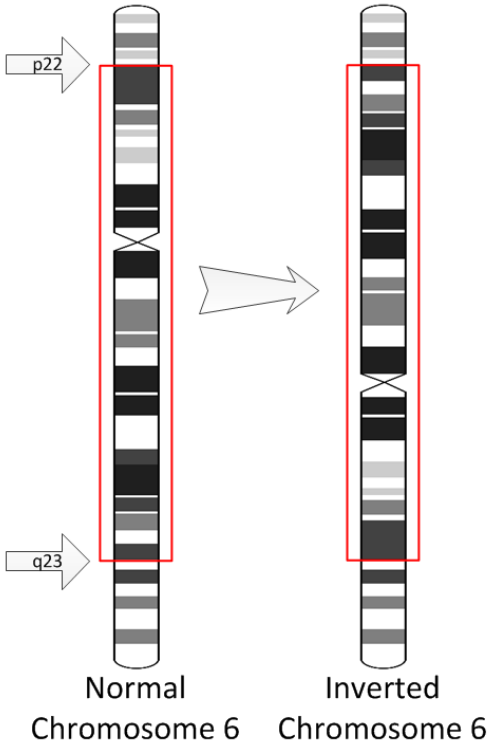
The full list of structural abnormalities, their associated abbreviated terms and definitions are seen in Table 5.2. This table identifies a key requirement; OWL Karyotypes should be able to model the variety of structural abnormalities (R4).

However not all karyotypes are quite so simple; a karyotype can include karyotypic information for numerous different clones. There are two types of clone karyotypes; *mosaic* (see Section 5.4.11) and *chimera* karyotypes. A mosaic karyotype is defined as a cell line originating from the same zygote, while a chimera karyotype is defined as a cell line originating from different zygotes. Each clone karyotype is followed by the number of cells with that karyotype in square brackets (`[ ]`).

In mosaic karyotypes, the karyotype may be preceded by the abbreviation **mos** and clone karyotypes are separated by a forward slash (`/`). A simple example of a mosaic ISCN String is *"mos47,XY,+21[12]/46.XY[18]"* which has two cell lines; 12 cells with a male cell line with trisomy 21 and 18 cells with a normal male cell line. Unlike other ISCN Strings exemplars we have shown so far, the clonal number provided in mosaic ISCN Strings represents results for a specific experiment, rather than a canonicalisation.



(a) Ideogram showing chromosomal gain such that trisomy 21 occurs.



(b) Ideogram showing an inversion abnormality in chromosome 6 with associated breakpoints 6p22 and 6q23.

Figure 5.3: There are two types of abnormalities: numerical and structural. Figure 5.3a shows an example karyotype with a numerical abnormality; specifically chromosomal gain. Figure 5.3b shows an example karyotype with a structural abnormality; specifically an inversion abnormality.

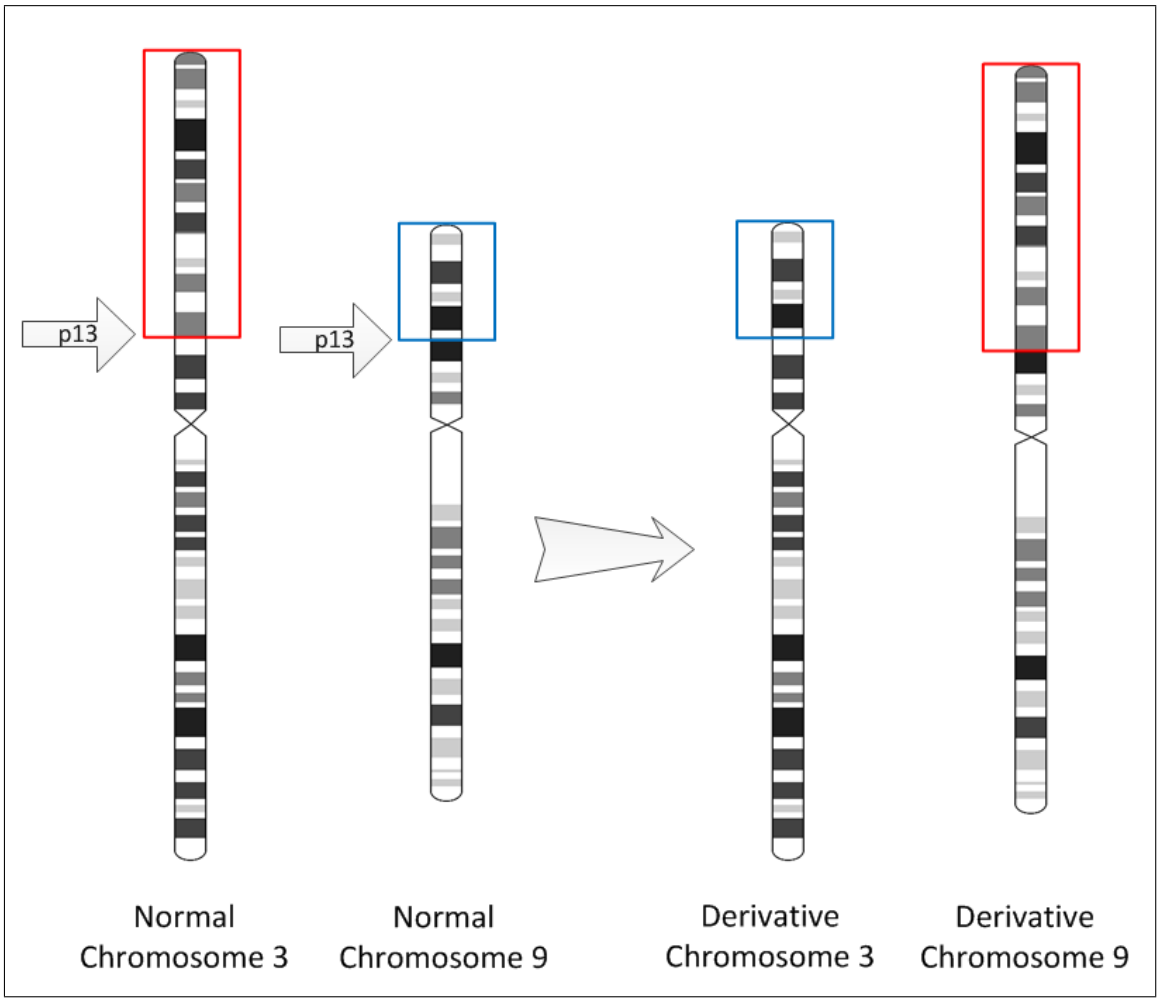


Figure 5.4: Ideogram showing a reciprocal translocation involving chromosomes 3 and 9.

Table 5.2: Table showing the variety of structural chromosomal rearrangements. Where available, definitions are directly taken straight from the ISCN2013 [135].

Type	Abbreviation	Biological Definition
Additional material of unknown origin	add	used to indicate additional material of unknown origin attached to a chromosome's region or band.
Deletion	del	a part of the chromosome is deleted or lost. *del* is used to denote both *terminal* (which involves either telomere) and *interstitial* (occurs in the interior of the chromosome) deletions.
Derivative Chromosome (see Section 5.4.8)	der	a structurally rearranged chromosome, with an intact centromere, generated either by a rearrangement involving two or more chromosomes or by multiple aberrations within a single chromosome.
Dicentric Chromosome	dic	a chromosome that contains two functional centromeres.
Duplication	dup	an extra copy of some chromosome segment exists (a duplicate).
Fission	fis	used to denote *centric fission*; splitting of one functional centromere of a chromosome to produce two centric chromosomes [116].
Fragile Site	fra	parts of the chromosome that show breaks when the cells are exposed to certain drugs or chemicals.

Continued on next page...

Table 5.2 – continued from previous page

Type	Abbreviation	Biological Definition
Homogeneously Staining Region	hsr	autonomously replicating extra chromosomal elements that are frequently associated with gene amplification in a variety of cancers [90].
Insertion	ins	used to indicate the insertion of additional material to a normal chromosome.
Inversion	inv	part of the chromosome has been reversed (rotated 180 degrees) therefore changing the order of the bands. *inv* is used to denote both *pericentric* and *paracentric* inversions; inversions that do and do not involve the centromere respectively.
Isochromosome	i	a chromosome with two identical arms, either two short arms or two long arms.
Marker Chromosome	mar	a structurally abnormal chromosome that cannot be unambiguously identified or characterized by conventional banding cytogenetics.
Neocentromere	neo	a functional centromere that has arisen or been activated within a region not known to have a centromere.
Quadruplication	qdq	a chromosome that contains four copies of the same chromosome segment.
Ring Chromosome	r	fusion of one or more chromosome arms to form a chromosome with no ends (ring)

Continued on next page...

Table 5.2 – continued from previous page

Type	Abbreviation	Biological Definition
Telomeric Association	tas	fusion between two telomeres of two different chromosomes without visible loss of chromosomal material [15].
Translocation	t	parts that are exchanged between non-homologous chromosomes.
Tricentric Chromosome	trc	a chromosome that contains three functional centromeres.
Triplication	trp	a chromosome that contains three copies of some chromosome segment (triplicate).

A chimera ISCN String is similar to a mosaic ISCN String. Instead the karyotype may be preceded by the abbreviation **chi** and the clone karyotypes are separated by a double forward slash (//).

These mosaic and chimera karyotypes are interesting as their existence requires us to review our definition of the chromosome complement; we are modelling more than one canonical karyotype rather than only one canonical, which conflicts with our definition of canonical. Therefore we derive a new requirement; our ontological model should be able to model these composite karyotypes (R7).

An abnormality can be also classified as either a constitutional or acquired abnormality. A constitutional abnormality (see Section 5.4.10), also known as an in-born abnormality, is an abnormality that is present in (almost) all cells of an individual and exists at the earliest stages of embryogenesis, while an acquired abnormality is an abnormality that develops in somatic cells [165].

Generally, constitutional abnormalities are indicated using the suffix **c**. For example the ISCN String *"48,XX,+8,+21c[20]"* represents tumour cells taken from a female individual that has a constitutional trisomy 21 and an acquired trisomy 8.

However constitutional sex chromosome numerical abnormalities are more complex still. Instead of using the + and - symbols to indicate numerical abnormalities, these constitutional sex chromosome abnormalities are included in the initial ISCN String sex description. For example the karyotype for an individual born with Tuners Syndrome (and no other abnormalities) is represented as *"45,X"*: a female individual that has 45 chromosomes and monosomy X (only one X chromosome); while an individual with Klinefelter Syndrome (and no other abnormalities) is represented as *"47,XXY"* (most common variant [158]); a male individual that has 47 chromosomes, of which two are X chromosomes and one a Y chromosome.

Therefore with the **c** suffix, acquired chromosome abnormalities in individuals with a constitutional sex chromosome abnormality can be distinguished. For example the ISCN String *"46,Xc,+21"* represents tumour cells taken from a female individual with Tuners Syndrome; a constitutional monosomy X and an acquired trisomy 21. Using this representation we see that karyotypes with constitutional abnormalities

explicitly define two types of canonicalisation; one of the individual and the other for the cell line they have given rise to.

This implicit and explicit annotating of constitutional abnormalities is important to cytogeneticists due to their relation to oncology; therefore we derive a new OWL Karyotype requirement (R6).

So far most of the karyotypes discussed contain only one or two abnormalities. However, ISCN String can quickly become a complicated composition of information. For example the ISCN String *"46,XX,t(3;9)(p13;q21)[14]/48,XX,+3,+9[11]/46,XX,t(1;6)(p11;p12)[9]/47,XX,t(6;10)(q12;p15),+7[6]/46,XX,inv(6)(p22q23)[3]/46,XX[7]"*. Here we have a mosaic karyotype of clones from a female individual that contains a variety of translocation and inversion events (discussed earlier in this section).

As we can see from this discussion, the description of the human chromosome complement is complex and involved. However in order to fully understand the complexities that underlie structural abnormalities, it is vital that we also briefly review the components of a chromosome and highlight those components that will be modelled in the ontology.

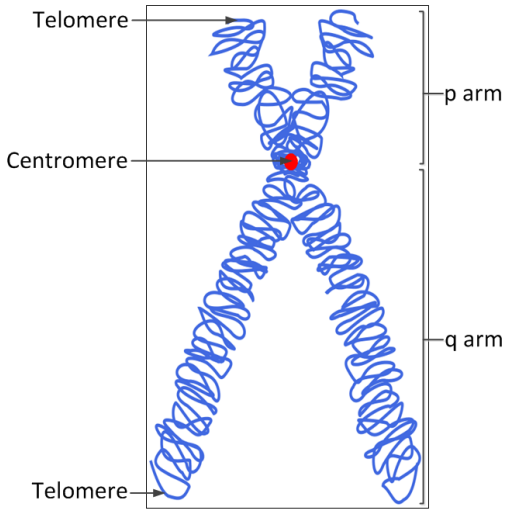
5.2.1 Reviewing chromosome components

A chromosome is an organised structure of DeoxyriboNucleic Acid (DNA), protein and RiboNucleic Acid (RNA) found in cells. The chromosome components can be seen in Figure 5.5a. According to ISCN2013 [135] each component can be represented as follows:

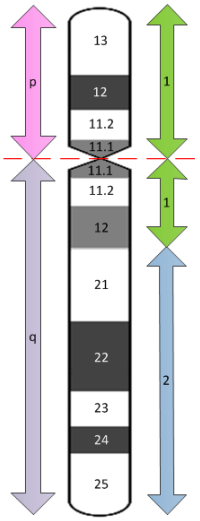
- Arm – the long arm is represented by `q` while the short arm is represented by `p`.
- Centromere – abstractly represented by `cen` or, more specifically, `p10` for the part of centromere facing the short arm and `q10` for the part facing the long arm.

- Telomere – represented by `ter`, `qter` is the long arm telomere and `pter` is the short arm telomere.

After staining, banding patterns can be visualised – the staining for chromosome 17 can be seen in Figure 5.5b. Bands are parts of the chromosome that are distinguishable from their adjacent segments, after being stained. The band name is a combination of: the chromosome number; the arm symbol; the region number; and the band number within that region. Regions and bands proximal to the centromere are labelled as 1, then 2 and so on. An example chromosome band is `17q21`, which informs us that the band is the first band (proximal to the centromere) in region 2 of the long arm of chromosome 17.



(a) Adapted image of a generic chromosome with its components clearly labelled.



(b) Ideogram of chromosome 17 with the bands clearly labelled.

Figure 5.5: Identifying the different chromosome components.

There are a variety of staining techniques available that are used in karyotyping. The first staining technique that produced banding patterns along the length of the human chromosomes is *Q-banding*. This technique uses Quinacrine (mustard or dihydrochloride) to bind to adenine-thymine rich regions of the chromosomes to produce a florescent banding pattern. This technique was followed by *G-banding* which uses Giemsa to bind to the phosphate groups of DNA (see Figure 5.2a) and produces a pattern of light and dark bands. Those techniques which provide banding

patterns showing the inverse (of light and dark) bands seen in G-banding are known as *R-banding*. Unlike Q-banding, G-banding and R-banding, *C-banding*, *T-banding* and *NOR-banding* are used to visualise the specific components of the chromosome resulting in a subset of bands; specifically centromeres, telomeres and Nucleolus Organiser Region (NOR)s respectively.

Banding patterns are complicated and are described at different resolutions. The five standard resolutions are 300-, 400-, 550-, 700- and 850- band levels (i.e. the approximate number of bands seen in a haploid set), such that the 300- band is a low resolution and the 850- band is a high resolution [135]. High resolution banding techniques result in existing bands being subdivided into sub-bands. The naming of these resulting sub-bands uses a hierarchical decimal system similar to the Dewey Decimal Classification (DDC). If we just focus on our continuing example of band `17q21` at resolution 400-, the sub-bands are `17q21.1`, `17q21.2` and `17q21.3` at resolution 500-. At resolution 700-, band `17q21.3` is further subdivided to `17q21.31`, `17q21.32` and `17q21.33`. A visual representation of this example, can be seen in Figure 5.6.

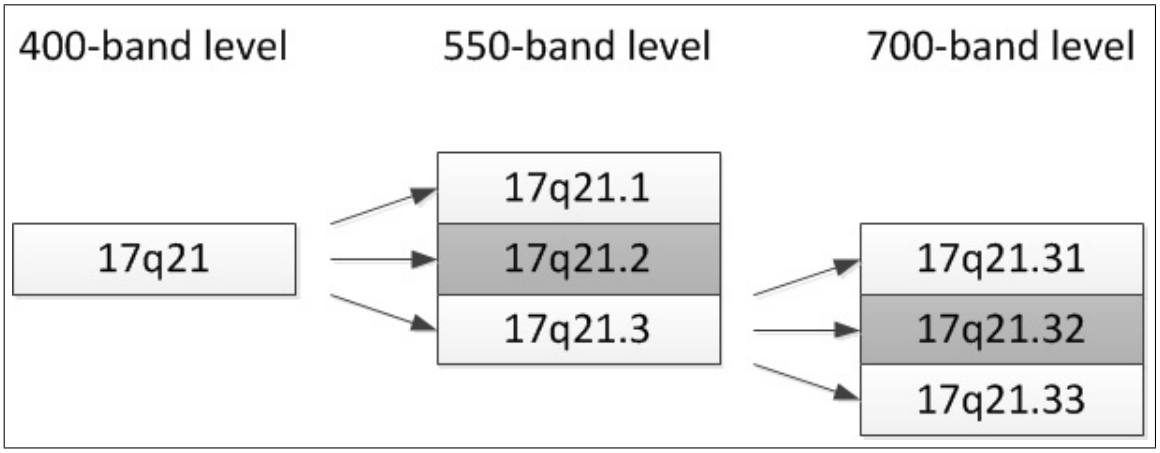


Figure 5.6: Visualising the higher resolution sub-bands of human chromosome band `17q21`.

In order to model karyotypes, we need to include concepts in the ontology to model the human chromosomes and their components (R1). The total number of chromosomal components can be seen in Table 5.3.

Now that we have reviewed the chromosome components we can now conclude our analytical review of ISCN Strings. Throughout this review we have been able to

Table 5.3: Table showing the type and number of chromosome components.

Biological entity	Number of entities
Chromosome	24
Arm	48
Centromere	24
Telomere	48
Bands and Sub-bands	1152
Total Number	1296

extract a set of key requirements for The Karyotype Ontology which we summarise in the next section.

5.2.2 Modelling requirements

Within this section we have seen how karyotypes are represented as ISCN Strings; the string format as defined by the ISCN. From this analysis we see that ISCN Strings have no formal (or computational) grammar and no formal interpretation. Additionally, the specification is not available in a computational format and is, therefore, not searchable. As a result karyotypes of diagnostic importance can be difficult to parse, validate and query, especially for more complicated ISCN Strings. From our analysis, we have identified nine key requirements that our ontological representation should have the ability to model. These modelling requirements, also known as functional requirements, are:

R1 The Karyotype Ontology should model the human chromosomes and chromosome components.

R2 The Karyotype Ontology should model the "normal" human chromosome karyotypes for each ploidy level i.e. haploid, diploid, triploid and tetraploid.

R3 OWL Karyotypes should be able to represent the two types of numerical abnormalities: gain and loss.

R4 OWL Karyotypes should be able to represent the variety of structural abnormalities (involving one or more chromosomes) identified in Table 5.2.

R5 OWL Karyotypes should be able to clearly model and differentiate between homologous chromosomes and their associated breakpoints.

R6 OWL Karyotypes should be able to model constitutional abnormalities.

R7 OWL Karyotypes should be able to model compositional karyotypes, such as mosaic and chimera karyotypes.

R8 The Karyotype Ontology should be able to determine the sex of a given diploid OWL Karyotype.

R9 The Karyotype Ontology should be able to determine the associated (near-)ploidy level.

In addition to the functional requirements, we identify five non-functional requirements:

R10 The Karyotype Ontology should be able to model all exemplars provided by the ISCN (see Section 5.5).

R11 The Karyotype Ontology "*should be made as simple as possible, but no simpler*"[6].

R12 The overall performance of The Karyotype Ontology (and Tawny-Karyotype[7]) should be acceptable for development, testing and deployment. Acceptable is defined loosely here, but for example, (re-)loading The Karyotype Ontology would not hinder development, but a 10 minute load time would. Test performance is less critical, as tests can be selected or run independently (i.e. continuously integrated), but should take minutes rather than hours.

R13 The Karyotype Ontology should be scalable. In Chapter 6 we show how we tested the scalability of The Karyotype Ontology by implementing up to 10^6 karyotypes and analysing the reasoning time.

[6]This paraphrased quote is often attributed to Einstein [109]
[7]Tawny-Karyotype refers to the Tawny-OWL code used to build The Karyotype Ontology.

R14 The Karyotype Ontology (and Tawny-Karyotype) should be extensible (i.e. allow the implementation of further features). In Chapter 6 we discuss the implementation of the `affects` relation and its effect of the ontology.

In the next section we discuss the approach used to model these karyotypes.

5.3 Design considerations

For the purposes of modelling karyotypes, we describe three different approaches to building the ontology: realism (see Section 5.3.1), partonomic axiomitisation (see Section 5.3.2) and event-based axiomitisation (see Section 5.3.3).

5.3.1 Portions of reality

Initial experiments with a realist ontology[8] represented karyotype definitions such that distinctions are made between the biological entity, the experimental artefact and the information content entity. This representation meant that distinctions between a chromosome (as a piece of DNA and protein), the experimental artefact (following staining) and the visualisation of the experimental artefact are all different "portions of reality". Therefore a chromosome in a living cell cannot meaningfully be said to have bands.

Some other distinctions incorporated into the realist ontology include:

- A karyotype may be used to describe the chromosome complement of different biological entities: for example, `Cell`, `Organism`, or `Nucleus`.
- Karyotypes may be descriptive of an actual chromosome complement (in a cell), or different levels of canonicalisation. So, in an individual with a *"46,XX"* karyotype not all cells would have exactly this complement, whilst a subset of species, although women would be expected to have a *"46,XX"* karyotype.
- The distinction between a biological entity and its visualisation. So, for example, an Ideogram is an ordered rearrangement of *images* of stained chromosomes. In some cases, the relationship between the visualisation and the biological entity is not clear; it is still not clear exactly what biological feature results in the visible banding patterns.

These distinctions can be represented ontologically, but results in one major difficulty; many hierarchies are replicated (for the majority of chromosomal compo-

[8]Developed by Dr Phillip Lord.

nents). For example chromosome 1 (biological entity), stained chromosome 1 (experimental artefact), and the visualisation of chromosome 1.

As we aim to develop a lightweight ontology with specific computational goals (R11), these distinctions are not required for our application. Any distinction that was not of practical use would only bloat the ontology without adding any clear value. Thus we follow a pragmatic approach [78].

5.3.2 A partonomic axiomitisation

Originally we defined karyotypes as a strict partonomy (i.e. a non-reflexive partonomy), utilising the `hasPart` and `isPartOf` relations; here we describe this type of ontology as a *partonomic ontology*. A resulting incomplete definition of a karyotype by its canonical parts (chromosomal bands) for a normal female individual can be seen in Listing 5.1.

```
(defclass k46_XX
 :super
  (exactly hasPart 2 HumanChromosome1BandpTer)
  (exactly hasPart 2 HumanChromosome1Bandp36.3)
  (exactly hasPart 2 HumanChromosome1Bandp36.2)
  ... 22 hasPart relations removed ...
  (exactly hasPart 2 HumanChromosome1BandqTer))
```

Listing 5.1: Incomplete karyotype definition for a normal female individual at 300-band level using the partonomic approach.

However we find that the size of the OWL definitions increased as the resolution level increased (see Listing 5.2). In this **incomplete** definition, we have to define 42 extra `hasPart` restrictions, while the **complete** definition would require 561 extra `hasPart` restrictions.

```
(defclass k46_XX
 :super
  (exactly hasPart 2 HumanChromosome1BandpTer)
  (exactly hasPart 2 HumanChromosome1Bandp36.33)
  (exactly hasPart 2 HumanChromosome1Bandp36.32)
  ... 64 hasPart relations removed ...
  (exactly hasPart 2 HumanChromosome1BandqTer))
```

Listing 5.2: Incomplete karyotype definition for a normal female individual at 850-band level using the partonomic approach.

Early versions of The Karyotype Ontology were manually built, which made this form of representation impractical. Programmatic approaches such as OPPL or Tawny-OWL would make it possible. However, due to the large number of axioms, we assumed that this type of modelling would negatively affect the performance of a partonomic ontology (R12). Theoretically, if we model karyotypes using this approach then: 10 karyotypes would have $\sim 8,500$ axioms; 100 karyotypes would have $\sim 85,000$ logical axioms; and so on. In particular, we anticipated that reasoning would have been costly: in Tawny-OWL, The Gene Ontology (GO) (which has $\sim 10^5$ logical axioms) only takes 12 seconds to reason over. Even with linear scaling[9] a partonomic ontology with 10,000 karyotypes (i.e. $\sim 8,500,000$ axioms) would reason in 16 minutes. This time growth conflicts with our idea of creating a useful and pragmatic ontology that the cytogeneticists could use (also contradicting R13). Therefore, we can conclude that we do not want to model individual karyotypes using a partonomy due to the extensive size of the OWL definitions and its potential reasoning time.

With further work, we also find that the partonomic ontology is unable to differentiate between two important edge cases as the definitions are logically equivalent but biologically distinct.

Edge case 1: The karyotype *"46,XX"* represents a female individual with a normal karyotype, while the *"46,XX,inv(1)(q11qTer)"* represents a female individual with the correct chromosome 1 bands, but not (necessarily) in the right order (see Figure 5.7). From this edge case, we learn that order is important when defining karyotypes.

```
(defclass k46_XX_inv!1!!q11qTer!
 :super
  (exactly hasPart 2 HumanChromosome1BandpTer)
  (exactly hasPart 2 HumanChromosome1Bandp36.3)
  (exactly hasPart 2 HumanChromosome1Bandp36.2)
  ... 22 hasPart relations removed ...
  (exactly hasPart 2 HumanChromosome1BandqTer))
```

Listing 5.3: Incomplete karyotype definition for a female individual with an inversion abnormality using the partonomic approach.

[9] which is unlikely

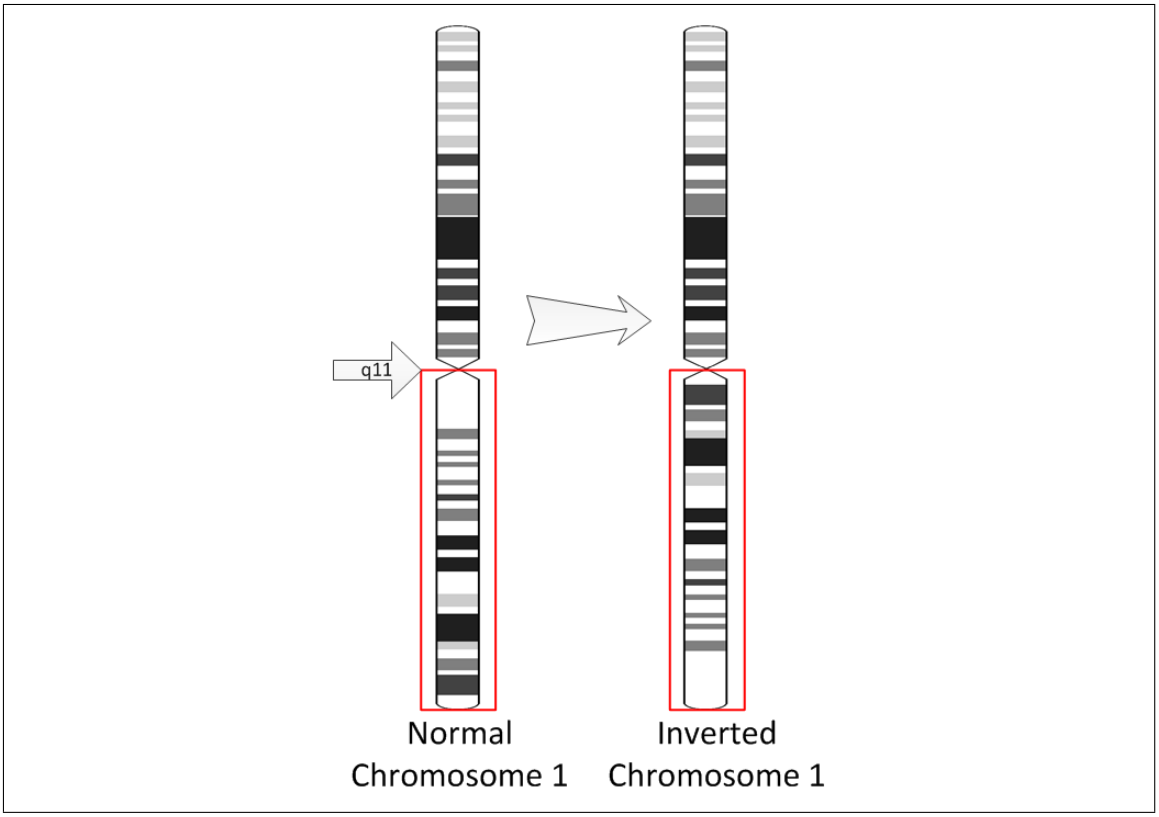


Figure 5.7: Ideogram showing an inversion abnormality involving chromosome 1 and breakpoints `q11` and `qTer`

Edge case 2: The karyotype *"45,X,-Y"* is a cell line from a male individual that has lost its chromosome `Y`, while *"45,X"* is an individual with monosomy `X` (only one chromosome), meaning they never had another sex chromosome in the first place. These karyotypes are partonomically identical, but are considered different. From this edge case, we learn that history is important when defining karyotypes.

```
(defclass k45_X
 :super
  (exactly hasPart 1 HumanChromosomeXBandpTer)
  (exactly hasPart 1 HumanChromosomeXBandp22.3)
  (exactly hasPart 1 HumanChromosomeXBandp22.2)
  ... 11 hasPart relations removed ...
  (exactly hasPart 1 HumanChromosomeXBandqTer))
```

Listing 5.4: Incomplete karyotype definition for a female individual with Tuners Syndrome using the partonomic approach.

Therefore, we modelled karyotypes using what we call event-based change (from a normal karyotype, in a diploid the normal is either *"46,XX"*, *"46,XY"* or *"46,XN"*). This way of modelling is similar to the ISCN Strings.

5.3.3 The event-based change axiomitisation

Similar to ISCN Strings representation, the event-based change approach explicitly asserts changes from the normal karyotype (R2). So, for example, to satisfy R3, we use addition and deletion events which describe chromosomal gain and loss respectively. An example event-based representation of a chromosomal gain abnormality using a cardinality restriction can be seen in Listing 5.5.

```
(exactly hasDirectEvent 1
 (and Addition HumanChromosome21))
```

Listing 5.5: Example chromosomal gain restriction *"+21"* utilising the `hasDirectEvent` object property.

However this numerical abnormality information on its own is not sufficient as we do not know if we are modelling disomy 21 in a haploid biological entity or trisomy 21 in diploid biological entity, and so on. Therefore, in using the event-based approach, all OWL Karyotypes must explicitly state their constitutional karyotype (R6). An example definition of a constitutional karyotype using an existential restriction can be seen in Listing 5.6.

```
(some derivedFrom k46_XX)
```

Listing 5.6: Example constitutional karyotype restriction using the `derivedFrom` object property.

Like the ISCN String representation, we can infer the chromosome aneuploids, using the explicit numerical abnormalities and constitutional karyotype. In the `k47_XX_+21` OWL Karyotype (see Listing 5.7), we know that the OWL Karyotype is a type of near-diploid cell (R2), therefore we have three of chromosome 21. This implicit condensation of knowledge is beneficial as the expanded listing of *"47,XX,+21"* would be similar to the partonomic approach.

```
(defclass k47_XX_+21
 :super
  (some derivedFrom k46_XX)
  (exactly hasDirectEvent 1
   (and Addition HumanChromosome21)))
```

Listing 5.7: Complete karyotype definition for a female individual with an inversion abnormality using the event-based change approach.

In order to model structural abnormalities as required by R4, each structural ab-

normality is modelled as an `Event` or `Feature` concept (20 in total). An example event-based representation of an inversion abnormality using a cardinality restriction can be seen in Listing 5.8. Unlike numerical abnormalities, structural abnormalities require more information; the associated breakpoints (bands in which the breaks have occurred) for that abnormality. In Listing 5.8, the associated breakpoints are `6p22` and `6q23`.

```
(defclass k46_XX_inv!6!!p22q23!
 :super
  (some derivedFrom k46_XX)
  (exactly hasDirectEvent 1
   (and
    Inversion
    (some hasBreakPoint HumanChromosome6Bandp22)
    (some hasBreakPoint HumanChromosome6Bandq23))))
```

Listing 5.8: Complete karyotype definition for a female individual with an inversion abnormality using the event-based change approach.

Further examples of how the ISCN can be modelled using the event-based approach and other modelling decisions applied in this development are shown in Section 5.4.

Most importantly, by using an approach that is similar to the ISCN specification (changes in relation to the near-ploidy level), we can now logically differentiate between the two identified edge cases.

Edge case 1: As discussed in Section 5.3.2, when using a partonomic approach, we cannot differentiate between the *"46,XX"* and *"46,XX,inv(1)(q11qTer)"* karyotypes. With an event-based approach this distinction is straight-forward. In Listing 5.9, we show the latter, which is clearly different by the presence of the inversion restriction.

```
(defclass k46_XX_inv!1!!q11qTer!
 :super
  (some derivedFrom k46_XX)
  (exactly hasDirectEvent 1
   (and
    Inversion
    (some hasBreakPoint HumanChromosome1Bandq11)
    (some hasBreakPoint HumanChromosome1BandqTer))))
```

Listing 5.9: Complete karyotype definition for a female individual with an inversion abnormality using the event-based change approach.

Edge case 2: Similarly, we can distinguish *"45,X,-Y"* and *"45,X"* karyotypes, as we explicitly assert the constitutional karyotype, as shown in Listings 5.10 and 5.11.

```
(defclass k45_X_-Y
 :super
  (some derivedFrom k46_XY)
  (exactly hasDirectEvent 1
   (and Deletion HumanChromosomeY)))
```

Listing 5.10: Complete karyotype definition for a cell line from a male individual with a missing Y chromosome using the event-based change approach.

```
(defclass k45_X
 :super
  (some derivedFrom
   (and
    k46_XN
    (exactly hasDirectEvent 1
     (and Deletion HumanSexChromosome)))))
```

Listing 5.11: Complete karyotype definition for female individual with Tuners Syndrome using the event-based change approach.

From the comparison of these three approaches, we can clearly see that the event-based change approach is the best method for modelling karyotypes. By chance we have arrived at a similar way of representing karyotypes as specified in the ISCN, which is to be expected due to the community's decades of experience in representing karyotypes.

5.4 Representing karyotypic knowledge

In this section, we present an overview of The Karyotype Ontology, its functionality and show how we have fulfilled the requirements identified in Section 5.2.2.

5.4.1 Modelling chromosome components

In Section 5.3.2, we showed that modelling karyotype instances using a partonomic approach can quickly expand the ontology because of the approach's exhaustive nature. This in turn potentially affects the reasoning performance of the ontology. However there are some parts of The Karyotype Ontology that still benefit from the use of the partonomic approach. For example, human chromosome components (identified in Section 5.2) are still modelled using a partonomy approach.

Unlike karyotype instances, there are a finite number of chromosome components (see Table 5.3), 1296 in fact, with approximately double that in terms of the number of logical axioms. Therefore, the potential reasoning performance problem identified earlier seems inconsequential.

We originally thought of modelling human chromosome band information as a strict partonomy, utilising the `hasPart` relations. A resulting definition of human chromosome 1 (including all bands from all five resolutions) can be seen in Listing 5.12.

```
(defclass HumanChromosome1
 :super
  HumanAutosome,
  (exactly hasPart 1 HumanChromosome1BandpTer)
  (exactly hasPart 1 HumanChromosome1Bandp36.3)
  (exactly hasPart 1 HumanChromosome1Bandp36.33)
  ... 87 hasPart relations removed ...
  (exactly hasPart 1 HumanChromosome1BandqTer))
```

Listing 5.12: Chromosome definition for a normal human chromosome 1 using a strict partonomy.

Instead we decided to model the partonomic relation using the inverted `hasPart` and specialised `isBandOf` and `isSubBandOf` relations, cutting down the size of the entity definitions[10].

[10] We are uncertain whether this small change in semantics will impact the reasoning performance. If required, we can invert, or explicitly add both the forward and inverse relations at a later date.

The use of partonomy for human chromosomal components is beneficial as it is used to enforce hierarchy as well as the existence of the chromosome components and the relations involving them in the ontology. For example a `HumanTelomere` cannot be a `isSubBandOf` of `HumanChromosome`. This partonomic modelling of the chromosome components fulfils R1.

5.4.2 Modelling normal *karyotypes*

In The Karyotype Ontology, normal karyotypes for each ploidy level are explicitly defined as `BaseKaryotype` subclasses (R2). Diploid karyotypes are represented as using the `k46_XN`[11] class, i.e. a karyotype that has one chromosome X and one other sex chromosome. The direct subclasses of the generic diploid karyotype are `k46_XX` and `k46_XY` which represent the male and female karyotype respectively.

The complete list of normal karyotypes for haploids, diploids, triploids and tetraploids are shown in Table 5.4.

Table 5.4: Table showing the available base karyotypes for each ploidy level.

Ploidy level	Base karyotypes
Haploid	`k23,N`
	`k23,X`
	`k23,Y`
Diploid	`k46,XN`
	`k46,XX`
	`k46,XY`
Triploid	`k69,XNN`
	`k69,XXX`
	`k69,XXY`
	`k69,XYY`
Tetraploid	`k92,XNNN`
	`k92,XXXX`
	`k92,XXXY`
	`k92,XXYY`
	`k92,XYYY`

These normal karyotypes are essential when using the event-based approach as it implicitly tells us ploidy number and the copy number of each chromosome. It is also

[11] In this section, name refers to the Clojure symbol. The `k` prefix has been introduced because Clojure symbols cannot start with a number.

useful in explicitly defining the constitutional karyotype of a karyotype; examples are discussed further in Section 5.4.10. This explicit modelling of the normal karyotypes fulfils R2.

5.4.3 Abnormality breakpoints

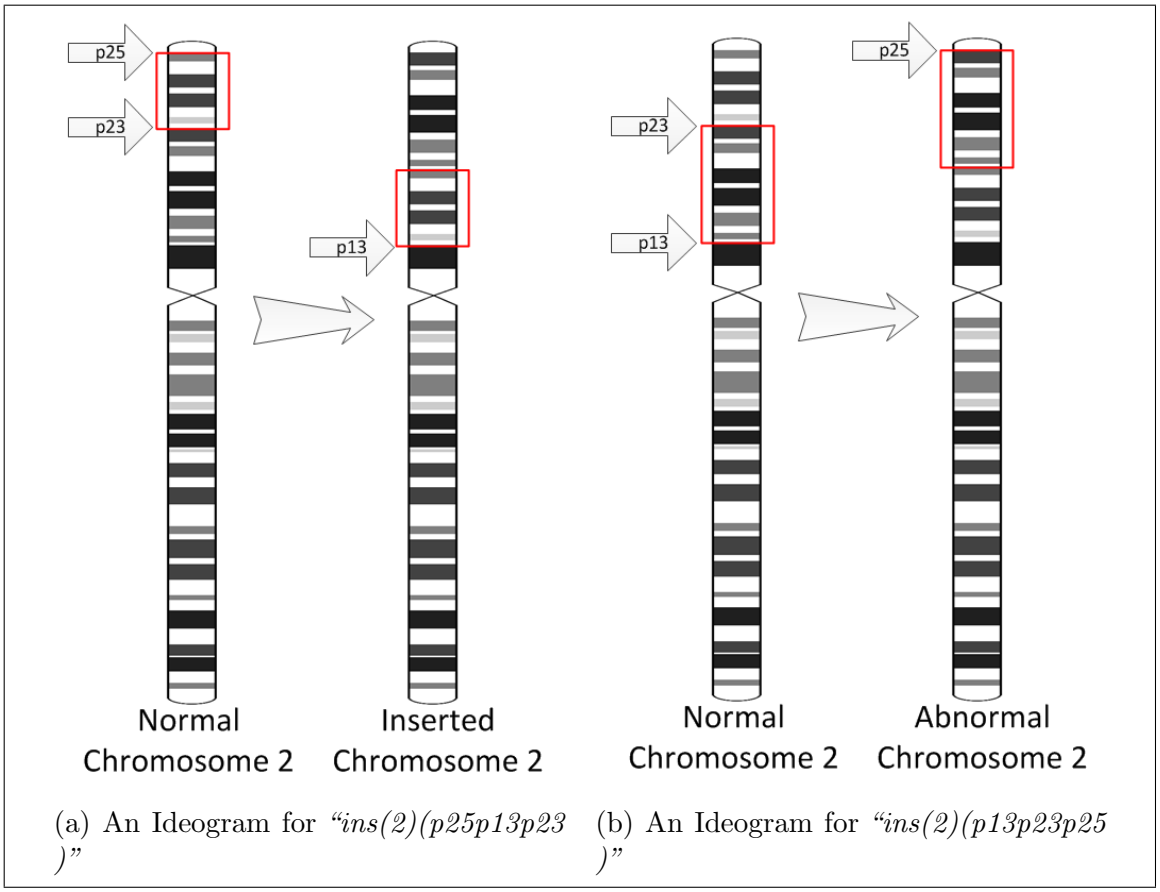
So far we have shown that chromosomal band abnormalities are modelled using the `hasBreakPoint` object property (see Listing 5.8). However this type of modelling is not without its limitations. This is especially true for insertion restrictions. For example, in Listing 5.13, we show the ontological definition of *"ins(2)(p25p13p23)"*[12] (see Figure 5.8a). The breakpoints have correctly been encoded, however by using this generic object property we have lost vital information in the translation. From this definition we can no longer determine the substitution segment and its loci. Due to this loss of information, we find that this ontological definition is also true for *"ins(2)(p13p23p25)"* (see Figure 5.8b). Therefore, in order to ontologically differentiate between these karyotypes we require more information; what we need is some way of specifying which are providing or receiving breakpoints.

```
(exactly hasDirectEvent 1
 (and
  Insertion
  (some hasBreakPoint HumanChromosome2Bandp13)
  (some hasBreakPoint HumanChromosome2Bandp23)
  (some hasBreakPoint HumanChromosome2Bandp25)))
```

Listing 5.13: Example insertion restriction *"ins(2)(p25p13p23)"* only utilising the `hasBreakPoint` object property.

Therefore specialised subproperties of `hasBreakPoint` were created: `hasProvidingBreakPoint` and `hasReceivingBreakPoint`. Now we can logically differentiate between the *"ins(2)(p25p13p23)"* and *"ins(2)(p13p23p25)"* karyotypes as shown in Listings 5.14 and 5.15.

[12]Note that *"ins(2)(p25p13p23)"* can also be written as *"ins(2;2)(p25;p13p23)"*.



(a) An Ideogram for *"ins(2)(p25p13p23)"*

(b) An Ideogram for *"ins(2)(p13p23p25)"*

Figure 5.8: Two valid representations of the ontological modelling shown in Listing 5.13

```
(exactly hasDirectEvent 1
 (and
  Insertion
  (some hasProvidingBreakPoint HumanChromosome2Bandp13)
  (some hasProvidingBreakPoint HumanChromosome2Bandp23)
  (some hasReceivingBreakPoint HumanChromosome2Bandp25)))
```

Listing 5.14: Example insertion restriction *"ins(2)(p25p13p23)"* using specialised sub properties of `hasBreakPoint`.

```
(exactly hasDirectEvent 1
 (and
  Insertion
  (some hasProvidingBreakPoint HumanChromosome2Bandp23)
  (some hasProvidingBreakPoint HumanChromosome2Bandp25)
  (some hasReceivingBreakPoint HumanChromosome2Bandp13)))
```

Listing 5.15: Example insertion restriction *"ins(2)(p13p23p25)"* using specialised subproperties of `hasBreakPoint`.

These specialised subproperties are not just useful for `Insertion` events, they are also useful in defining `Translocation` events. This explicit modelling of abnormality breakpoints partially fulfils R4.

5.4.4 Orientation of substitution segments

So far we have shown abnormalities that have the same orientation as the origin, which is known as a direct substitution. Here, we show that this modelling is not explicit enough when modelling duplication events that have been inverted. For example, in Listing 5.16 we show the ontological definition of *"dup(1)(q25q22)"* (see Figure 5.9) and whilst the breakpoints have been correctly encoded, by using the generic OWL entity `Duplication` we have lost vital information in the translation. From this definition we can no longer determine if this duplicated segment is inverted (i.e. *"dup(1)(q25q22)"*) or direct (i.e. *"dup(1)(q22q25)"*). This lack of detail means that to ontologically differentiate the two, we require more specific OWL classes.

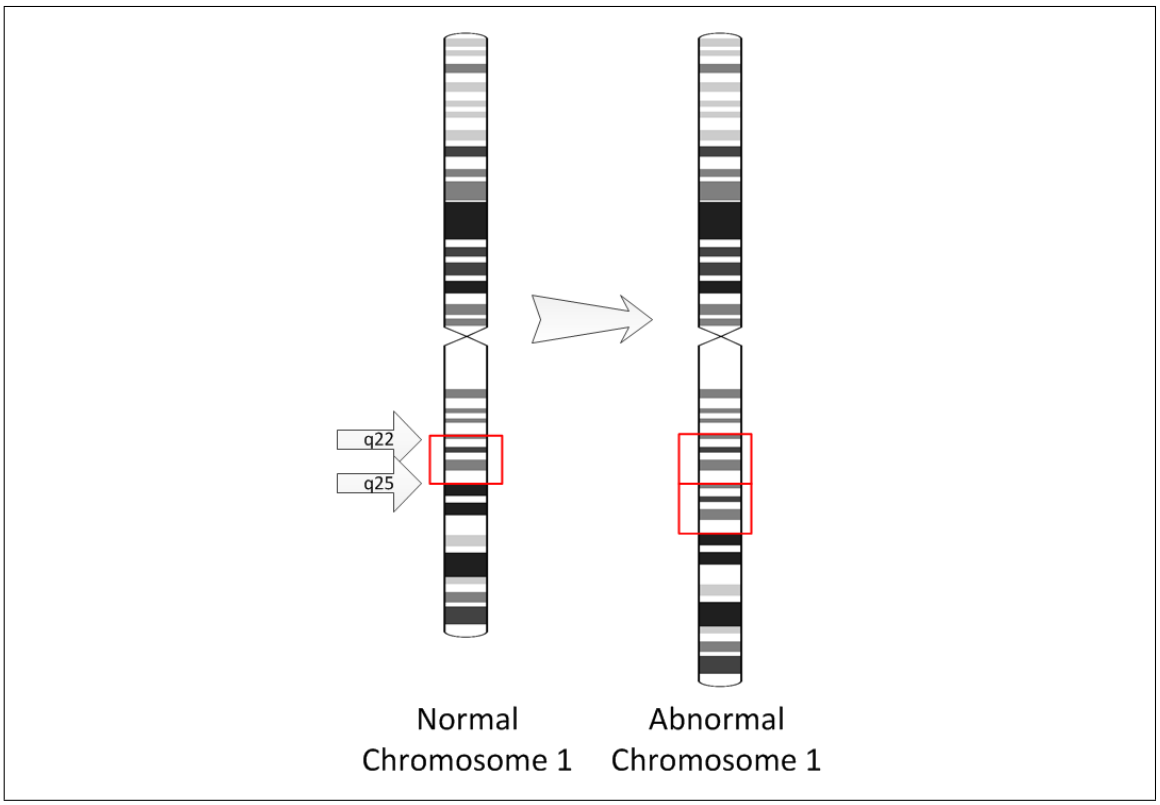


Figure 5.9: An Ideogram of an inverse duplication event.

```
(exactly hasDirectEvent 1
 (and
  Duplication
  (some hasBreakPoint HumanChromosome1Bandq22)
  (some hasBreakPoint HumanChromosome1Bandq25)))
```

Listing 5.16: Example inverse duplication restriction *"dup(1)(q25q22)"* using the `Duplication` class.

Therefore specialised subclasses of `Duplication` were created: `DirectDuplication` and `InverseDuplication`. Now we can differentiate between *"dup(1)(q25q22)"* and *"dup(1)(q22q25)"*, as shown in Listings 5.18 and 5.17 respectively.

```
(exactly hasDirectEvent 1
 (and
  InverseDuplication
  (some hasBreakPoint HumanChromosome1Bandq22)
  (some hasBreakPoint HumanChromosome1Bandq25)))
```

Listing 5.17: Example inverse duplication restriction *"dup(1)(q25q22)"* using specialised subclasses of `Duplication`.

```
(exactly hasDirectEvent 1
 (and
  DirectDuplication
  (some hasBreakPoint HumanChromosome1Bandq22)
  (some hasBreakPoint HumanChromosome1Bandq25)))
```

Listing 5.18: Example direct duplication restriction *"dup(1)(q22q25)"* using specialised subclasses of `Duplication`.

This modelling works for simple abnormalities such as `Duplication` and `Insertion` events, but does not hold for abnormalities such as `Translocation` events, where it is possible that not all of the "substitutions" in the `Translocation` event are direct or inverse, but a combination of both orientations. For example the *"(5;14;9)(q13q23;q24q21;p12p23)"* karyotype has two direct and one inverse substitutions (see Figure 5.10). In this instance it makes more sense to assert an `Inversion` abnormality for the one inverted substitution, as shown in Listing 5.19.

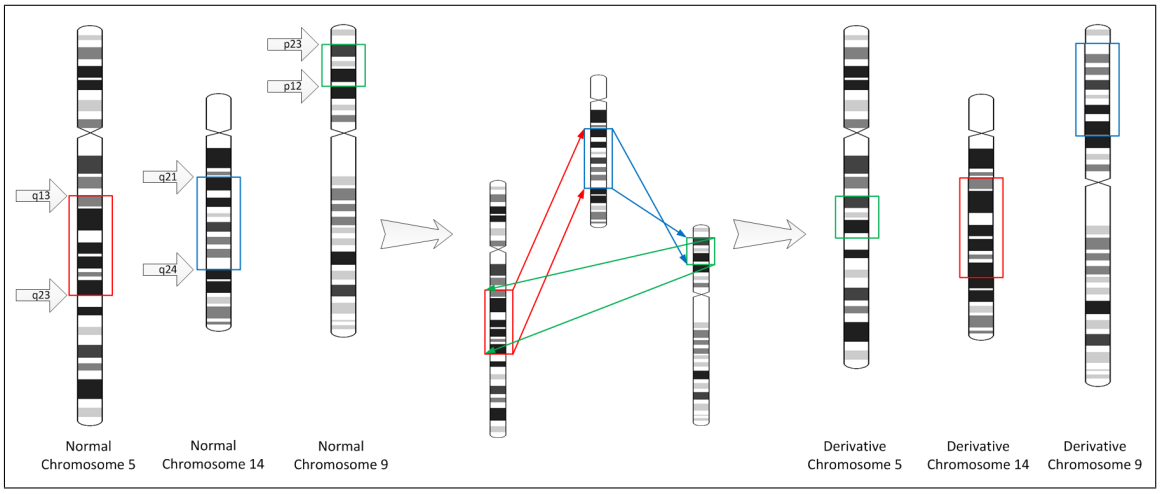


Figure 5.10: An Ideogram of a translocation event that contains direct and inverse substitutions.

```
(exactly hasDirectEvent 1
 (and
  Translocation
  (and
   (some hasProvidingBreakPoint HumanChromosome14Bandq21)
   (some hasProvidingBreakPoint HumanChromosome14Bandq24)
   (some hasReceivingBreakPoint HumanChromosome5Bandq13)
   (some hasReceivingBreakPoint HumanChromosome5Bandq23))
  (and
   (some hasProvidingBreakPoint HumanChromosome14Bandq21)
   (some hasProvidingBreakPoint HumanChromosome14Bandq24)
   (some hasReceivingBreakPoint HumanChromosome9Bandp12)
   (some hasReceivingBreakPoint HumanChromosome9Bandp23))
  (and
   (some hasProvidingBreakPoint HumanChromosome5Bandq13)
   (some hasProvidingBreakPoint HumanChromosome5Bandq23)
   (some hasReceivingBreakPoint HumanChromosome14Bandq21)
   (some hasReceivingBreakPoint HumanChromosome14Bandq24))
  (and
   (some hasDirectEvent
    (and
     Inversion
     (some hasBreakPoint HumanChromosome14Bandq21)
     (some hasBreakPoint HumanChromosome14Bandq24))))))
```

Listing 5.19: Example translocation restriction *"t(5;14;9)(q13q23;q24q21;p12p23)"* with a mixture of direct and inverse "substitutions". In this example the 14q21 to 14q24 segment is inverted.

This modelling works well for `Translocation` events but is unnecessary within the `Duplication` and `Insertion` patterns. In Listing 5.20, we see that the break point information is duplicated. Thus both ways of modelling are available to the user. This explicit modelling of abnormality orientation fulfils part of R4.

```
(exactly hasDirectEvent 1
 (and
  Duplication
  (some hasBreakPoint HumanChromosome1Bandq22)
  (some hasBreakPoint HumanChromosome1Bandq25)
  (some hasDirectEvent
   (and
    Inversion
    (some hasBreakPoint HumanChromosome1Bandq22)
    (some hasBreakPoint HumanChromosome1Bandq25)))))
```

Listing 5.20: Example inverse duplication restriction *"dup(1)(q25q22)"* using the `Inversion` event.

5.4.5 *Partial knowledge*

So far we have only shown abnormalities with complete information. ISCN provides a syntax for situations where we are unable to determine the origin segment or the exact breakpoint loci. Here, a question mark (**?**) is used in the ISCN String. Take as an example the ISCN String *"ins(1,?)(p22,?)"*; in this exemplar, an insertion event has occurred at `1p22`, although we are uncertain as to its origin. One benefit of OWL is its ability to represent partial knowledge straight-forwardly. We represent this partial knowledge by using the more generic parent class, such as `HumanChromosomeBand` (see Listing 5.21).

```
(exactly hasDirectEvent 1
 (and
  Insertion
  (some hasReceivingBreakPoint HumanChromosome1Bandp22)
  (some hasProvidingBreakPoint HumanChromosomeBand)))
```

Listing 5.21: Example inversion restriction with partial information.

5.4.6 *Modelling uncertainty*

The ISCN also provides a syntax for situations where we are uncertain about the abnormality or origin segment. Here, a question mark (**?**) is also used in the ISCN String (i.e. the question mark character precedes the potential abnormality description). An example ISCN String is the *"46,XX,?del(1)(q12)"*; in this exemplar, it is possible that a deletion event has occurred at `1q12`, however we cannot be sure. We model uncertain events using the specialised subproperty of `hasEvent`; `hasUncertainEvent` (see Listing 5.22).

```
(defclass k46_XX_?del!1!!q12!
 :super
 (some derivedFrom k46_XX)
 (exactly hasUncertainEvent 2
  (and
   Deletion
   (some hasBreakPoint HumanChromosome1Bandq12)
   (some hasBreakPoint HumanChromosome1BandqTer))))
```

Listing 5.22: Example karyotype with an uncertain structural abnormality.

5.4.7 Multiple copies of rearranged chromosomes

So far we have only shown distinct abnormalities, i.e. the abnormality definition only occurs once in the karyotype. If the same abnormality definition occurs more than once, i.e. the same abnormality affects the homologous chromosome, then the ISCN String uses a multiplication sign (**x**). An example ISCN String the *"46,XX, del(6)(q13q23)x2"*; in this exemplar, the same deletion effect occurs in both copies of chromosome `6`. Ontologically, we can model this replication by simply increasing the cardinality of the restriction (see Listing 5.23).

```
(defclass k46_XX_del!6!!q13q23!x2
 :super
 (some derivedFrom k46_XX)
 (exactly hasDirectEvent 2
  (and
   Deletion
   (some hasBreakPoint HumanChromosome6Bandq13)
   (some hasBreakPoint HumanChromosome6Bandq23))))
```

Listing 5.23: Example karyotype with multiple copies of rearranged chromosomes.

5.4.8 Derivative chromosomes

During the modelling of our structural abnormalities, we found that unlike the other `Event`s and `Feature`s, a derivative chromosome is generated by multiple abnormalities occurring within a single chromosome[13] i.e. through the composition of many events. An example ISCN String is *"der(5)add(5)(p15.3)add(5)(q23)"*; in this exemplar, a derivative chromosome `5` is generated by the addition of unknown material at breakpoints `5p15.2` and `5q23` (see Listing 5.24). This explicit modelling of derivative chromosomes fulfils part of R4.

[13]Or by the rearrangement of two or more chromosomes.

```
(exactly hasDirectFeature 1
 (and
  DerivativeChromosome
  (some isRearrangedChromosomeOf HumanChromosome5)
  (hasDirectEvent some
   (and
    Addition
    (some hasBreakPoint HumanChromosome5Bandp15.3)))
  (hasDirectEvent some
   (and
    Addition
    (some hasBreakPoint HumanChromosome5Bandq23)))))x
```

Listing 5.24: Example derivative chromosome restriction *"der(5)add(5)(p15.3)add(5)(q23)"*.

5.4.9 Abnormalities involving homologous chromosomes

So far we have shown abnormalities that affect two non-homologous chromosomes. Unfortunately, this modelling is not explicit enough for insertion events that affect homologous chromosomes. For example, in Listing 5.25 we show an ontological definition of *"ins(2;2)(p13;q21q31)"* (see Figure 5.11). From this definition, we can no longer determine if this insertion takes place in the same chromosome (i.e. *"ins (2)(p13;q21q31)"*) or affects two chromosomes (i.e. *"ins(2;2)(p13;q21q31)"*)[14]. This lack of knowledge means that to ontologically differentiate the two, we require more specific OWL classes.

```
(exactly hasDirectEvent 1
 (and
  Insertion
  (some hasProvidingBreakPoint HumanChromosome2Bandq21)
  (some hasProvidingBreakPoint HumanChromosome2Bandq31)
  (some hasReceivingBreakPoint HumanChromosome2Bandp13)))
```

Listing 5.25: Example insertion restriction *"ins(2;2)(p13;q21q31)"* using the `Insertion` class.

Therefore specialised subclasses of `Insertion` were created – `InsertionOneChromosome` and `InsertionTwoChromosome`. Now we can differentiate between *"ins(2;2)(p25;p13p23)"* and *"ins(2)(p13;p23p25)"* as shown in Listings 5.14 and 5.15.

[14]ISCN also does not make this distinction naturally, and has to resort to underlining one chromosome to distinguish the two.

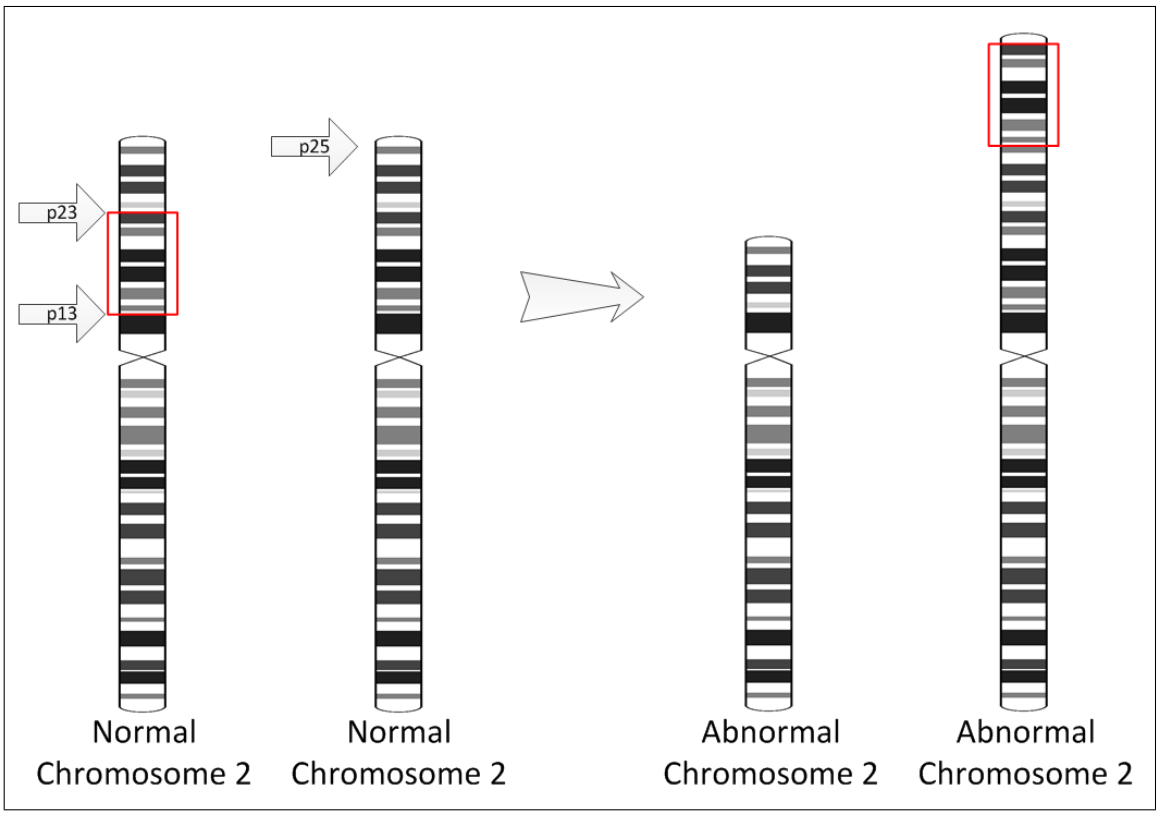


Figure 5.11: An Ideogram showing an insertion event that affects homologous chromosomes.

```
(exactly hasDirectEvent 1
 (and
  InsertionOneChromosome
  (some hasProvidingBreakPoint HumanChromosome2Bandq21)
  (some hasProvidingBreakPoint HumanChromosome2Bandq31)
  (some hasReceivingBreakPoint HumanChromosome2Bandp13)))
```

Listing 5.26: Example insertion restriction ISCN Stringins(2;2)(p13;q21q31) using specialised subclasses of `Insertion`.

```
(exactly hasDirectEvent 1
 (and
  InsertionTwoChromosome
  (some hasProvidingBreakPoint HumanChromosome2Bandq21)
  (some hasProvidingBreakPoint HumanChromosome2Bandq31)
  (some hasReceivingBreakPoint HumanChromosome2Bandp13)))
```

Listing 5.27: Example insertion restriction *"ins(2;2*)(p13;q21q31)"* using specialised subclasses of `Insertion`.

These specialist subclasses are specific to `Insertion` events. In all other ontological abnormality definitions, we can implicitly determine if homologous chromosomes are involved. This modelling of abnormalities that affect homologous chromosomes fulfils R5.

5.4.10 Constitutional anomalies

As described in Section 5.2, abnormalities can be classified as either constitutional or acquired. So far we have only shown the ontological modelling of acquired abnormalities, i.e. abnormalities that develop in somatic cells. If the abnormality is in-born, i.e. is present in most cells of an individual from early embyrogenesis, then the abnormality is classified as a constitutional abnormality. Generally, constitutional karyotypes are indicted with the **c** suffix in ISCN; in The Karyotype Ontology we model constitutional abnormalities explicitly, using the `derivedFrom` restriction. An example ISCN String is the *"k48,XXYc,+X"*; in this exemplar, a male individual has two extra `X` chromosomes, one he was born with, the other has been acquired, (see Listing 5.28). This explicit modelling of constitutional abnormalities fulfils R6.

```
(defclass k48_XXYc_+X
 :super
  (some derivedFrom
   (and
    (some derivedFrom k46_XY)
    (exactly hasEvent 1
     (and Addition HumanChromosomeX))))
  (exactly hasEvent 1
   (and Addition HumanChromosomeX)))
```

Listing 5.28: Example karyotype with constitutional anomalies.

5.4.11 Mosaic karyotypes

So far we have not shown the ontological modelling of compositional karyotypes such as mosaic or chimera karyotypes; i.e. where an individual human appears to have more than one cell line with different karyotypes.

Generally, mosaic karyotypes are preceded with the **mos** abbreviation, with clone karyotypes separated by the forward slash character (/). An example ISCN String

is *"mos47,XY,+21[12]/46,XY[18]"*; in this exemplar, we have a cell line from a male individual that contains 18 normal cells and 12 cells with an acquired chromosome `21` (see Listing 5.29).

In The Karyotype Ontology, chimera karyotypes have a very similar representation; in order to differentiate between the two, the `MosaicKaryotype` and `ChimeraKaryotype` classes are explicitly defined. This explicit modelling of compositional karyotypes fulfils R7.

```
(defclass kmos_47_XY_+21!12!_46_XY!18!
 :super
  MosaicKaryotype
  (and
   (and
    HumanKaryotype
    (some derivedFrom k46_XY)
    (exactly 1 hasEvent
             (and Addition
                  HumanChromosome21)))
   (hasClone value 12))
  (and
   k46_XY
   (hasClone value 18)))
```

Listing 5.29: Example ontological model of a mosaic karyotype.

5.4.12 Identifying the (near-)ploidy levels

As discussed in Section 5.2, we can use the number of chromosomes to determine the (near-)ploidy level of a karyotype. In The Karyotype Ontology, we do not explicitly define the number of chromosomes, but implicitly in the `BaseKaryotype`. Thus, to find all diploid karyotypes we use the equivalency definition, as shown in Listing 5.30[15]. The definitions for all other ploidy levels are similar to the diploid definition shown in Listing 5.30; the concept name and base karyotypes are replaced as appropriate. This explicit modelling of (near-)ploidy levels fulfils R9.

[15] As the `derivedFrom` object property is transitive, we also find diploid karyotypes that contain constitutional abnormalities.

```
(defclass DiploidKaryotype
 :equivalent
  (and
   HumanKaryotype
   (some derivedFrom k46,XN)))
```

Listing 5.30: Tawny-OWL definition for "diploid" karyotypes using the partonomic approach.

5.4.13 *Defining sex*

While building this ontology, we found that sex is not as intuitive as it seems[16]. We need to be able to describe "male" and "female" karyotypes. The obvious definition for sex was that a "male" karyotype should be defined as a karyotype with a `Y` chromosome, while a "female" karyotype as one without, as shown in Listings 5.31 and 5.32 respectively. This type of modelling would be easier with a partonomic ontology rather than event-based change modelling. However further investigation showed that these definitions are, in fact, too simplistic as the karyotype *"45,X,-Y"*[17], has no `Y` chromosome, yet would generally be considered to be a "male" karyotype.

```
(defclass MaleKaryotype
 :equivalent
  (and
   Karyotype
   (some hasPart HumanChromosomeY)))
```

Listing 5.31: Tawny-OWL definition for "male" karyotypes using the partonomic approach.

```
(defclass FemaleKaryotype
 :equivalent
 (and
  Karyotype
  (not MaleKaryotype)))
```

Listing 5.32: Tawny-OWL definition for "female" karyotypes using the partonomic approach.

Therefore, the finalised definition for sex, as shown in Listing 5.33 and Listing 5.34, considers the history of the (constitutional) karyotype by asserting a `derivedFrom`

[16] Ontologically, as in real life.
[17] A male-derived cell line which has lost its `Y` chromosome.

relation. Using these definitions, the *"45,X,-Y"* karyotype can be correctly stated as being a "male" karyotype.

```
(defclass MaleKaryotype
 :equivalent
  (or
   k46_XY
   (some derivedFrom k46_XY)))
```

Listing 5.33: Tawny-OWL definition for male karyotypes.

```
(defclass MaleKaryotype
 :equivalent
  (or
   k46_XX
   (some derivedFrom k46_XX)))
```

Listing 5.34: Tawny-OWL definition for female karyotypes.

However these definitions are unable to ontologically categorise the *"45,X"* karyotype as either female or male though it would generally be considered a "female" karyotype (see Listing 5.11).

There is no correct answer to this problem. We could either redefine our female karyotype to include the *"45,X"* karyotype or add phenotypic sex. This decision needs to be taken by the domain experts themselves.

5.5 Assessment

As well as providing a specification, we are fortunate that ISCN2013 provides many examples; we are using these examples as an initial evaluation for our ontology, to determine whether The Karyotype Ontology is expressive enough to represent these exemplar karyotypes (R10).

We wanted the Clojure symbols of these exemplars to be related to the ISCN String as, in most cases, there is no other more humanly readable name. Thus the following changes were incorporated to ensure that the Clojure symbol was legal with all associated syntaxes (i.e. Clojure, Manchester Syntax, and the Uniform Resource Identifier (URI) specification):

- All karyotype exemplars start with the **k** character – Clojure symbols cannot start with numbers
- Replaced **;** character with **_** – a semicolon is a comment in Clojure
- Replaced **(** and **)** characters with **!** – parenthesis are list delimiters in Clojure
- Replaced **,** character with **_** – a comma is a separator in Manchester Syntax
- Replaced **/** character with **_** – a forward slash is a namespace separator in Clojure

During this implementation process, we have also discovered two difficulties with the existing ISCN2013 specification; in both cases a simple and intuitive correction is possible:

- The lack of bands `17q11` and `Xq12` in figures showing chromosome bands (page 28 and page 31); the figures are the only list of all chromosome bands in ISCN2013 (and ISCN2009 [136]).
- The absence of a band `Yq11.2` in the 300 band resolution (`11.21`, `11.22`, `11.23` do exist on page 31) while this band is used in several exemplars (for example on page 78). This band does exist in ISCN2005 – a past specification.

In total, The Karyotype Ontology has 1616 classes, 29 object properties and 1 datatype property; see Table 5.5 for the complete class statistics.

Table 5.5: Table showing the type and number of implemented classes in The Karyotype Ontology.

Class Type	Number of Classes
Chromosome	27
Chromosome Component	1
Centromere	25
Telomere	25
Bands and Sub-bands	1286
Event	20
Feature	21
Karyotype	1
Base Karyotype	16
ISCN Example Karyotype	170
Named Karyotype	18
Resolution	6
Total Number	1616

Overall, our ontological representation of karyotypes has met almost all of the functional requirements identified in Section 5.2.2, with the exception of R4 in its entirety; specifically structural rearrangements such as Fragile Sites and Homogeneously Staining Regions. As a result, in total, we have represented 170 (of 260)[18] karyotypes in The Karyotype Ontology. Further work is required in order to fulfil R4, which in turn should enable the success of R10.

One limitation still exists; there may be more than one "path" to the same karyotype. One extreme example of this limitation is the *"46,XX,-1,-2,...,-22,-X"* and *"23,X"* karyotypes. However, this limitation also exists within the ISCN, thus we do not make things any worse.

[18]This value does not include the FISH and duplicate ISCN String exemplars.

5.6 Summary

Currently, karyotypes have only been represented as experimental entities, or results of medical procedures. In this chapter, we discuss the steps taken to provide a computational and formal representation and interpretation of karyotypes (RQ1). These steps include the analysis of the ISCN to identify functional (and non-functional) requirements, the design and implementation (i.e. modelling decisions) of the The Karyotype Ontology, and lastly the testing of this model using exemplars from the ISCN. To summarise, we found that the resulting ontology satisfies the functional requirements identified in Section 5.2.2, using the exemplars provided by the ISCN as evidence.

Indeed, in successfully building a computational representation of the ISCN, we have also shown that the pattern-driven and programmatic approach is applicable in the modelling of new biological knowledge. In addition, we have produced an ontology that will potentially be valuable for cytogeneticists by transforming collections of karyotypes to a form that is easy to query, validate and maintain (RQ2). However, as discussed in Chapter 2 the knowledge of karyotypes is bounded as the existing interpretation (i.e. the ISCN specification) is already mature. Thus we must also show that this approach can model other types of biological knowledge, specifically mitochondria, where this complex and incomplete knowledge can be found in many different sources and formats (see Chapter 7).

Lastly, we have shown that by using this approach we enforce consistency meaning we can potentially identify documentation errors (RQ3). By explicitly encoding the biological knowledge underlying the ISCN, we were able to identify two missing bands. Whilst encoding the ISCN exemplars, we found that several exemplars referred to a band that did not exist.

In the next chapter, we aim to prove that The Karyotype Ontology also satisfies the last three non-functional requirements: performance, scalability, and extensibility.

6

Scaling The Karyotype Ontology

Contents

6.1 Introduction 116
6.2 Creating random ontologies 117
6.3 Performance 119
6.4 Scaling The Karyotype Ontology 121
6.5 Incorporating `affects` restrictions 122
6.5.1 The `affects` implementations 122
6.5.2 Results 127
6.6 Summary 129

6.1 Introduction

Over the years, the number of ISCN Strings has grown to at least a hundred thousand across multiple research databases that have clinical importance. Now we have the initial prototype of The Karyotype Ontology (see Chapter 5), we need to test that the ontology could handle these numbers of instances, as well as investigate the general scalability performance of the model (R13).

Our pattern driven approach means that it is possible to change and refine our ontological models rapidly, even where these changes affect many entities within the ontology (R14). In this chapter, we use this capability to investigate the performance of several different axiomitisations of the same knowledge; specifically, a deletion or inversion event `affects` a sequence of bands between two breakpoints. There are various ways to implement this relation, however *a priori* it is difficult to determine which of these will work best, particularly with respect to non-functional characteristics such as reasoning time. Here, we test the scalability of three different axiomatisations for the `affects` relation. The programmatic methodology of The Karyotype Ontology means that we can test scaling within the same environment (Clojure); for example, by generating random karyotypes for an ontology as well as collating and graphing the reasoning results of many random ontologies.

The corresponding code and supplementary data of our investigation into the scaling performance of The Karyotype Ontology and the incorporation of the `affects` relation can be found at the project website[1].

[1] `https://github.com/jaydchan/tawny-karyotype-scaling`

6.2 Creating random ontologies

In order to test the scalability of The Karyotype Ontology, multiple test ontologies were required. Using Tawny-OWL and the Tawny-Karyotype project, we can programmatically create N number of random ontologies.

Tawny-Karyotype provides us with the ability to generate K number of random karyotypes for an ontology; each of these random karyotypes has one random `derivedFrom` restriction (defining the sex) and M number of `hasDirectEvent` restrictions. This is coded in the `random-karyotype-driver` function.

In order to test the effects of the different `affects` relation implementations on The Karyotype Ontology, we aim to compare the reasoning times for: an ontology with no `affects` relation, and three further ontologies that contain one `affects` implementation each. This was accomplished by refining the random ontologies and applying each `affects` implementation (A). These implementations are available in the functions `affects1-driver`, `affects2-driver` and `affects3-driver` respectively.

For this work we use the following variable values:

N = number of ontologies = 100

K = number of karyotype classes within an ontology = { 10^1 10^2 10^3 10^4 10^5}

M = number of event restrictions for each karyotype class = { 0 . . . 10 }

A = the `affects` implementation = { 0 . . . 3 } where 0 means an ontology that has no `affects` relationship incorporated (i.e. null-affects) and 1, 2, 3 with the closure-affects, sequence-affects, data-affects incorporated respectively.

This means that in total 22,000 ontologies were generated.

The main limitation of our randomness is that we do not know that the distributions in our random karyotypes are similar to a real-world collection of karyotypes. Part of the problem here is that we do not have a large body of karyotypes in a structured format[2], so this is difficult or impossible to do.

[2]This is why we are writing The Karyotype Ontology.

Therefore, we have picked a simplistic scheme for generating random ontologies; first, we only use 4^3 from the 13 events, and secondly only use the 300-band level chromosome bands from chromosome `1`. The Karyotype Ontology is highly patternised, and all events are defined in terms of cardinality and existential restrictions, therefore this simple scheme should be sufficient.

[3]Chromosomal addition, chromosomal deletion, chromosomal band addition and chromosomal band deletion

6.3 Performance

Due to the large number of ontologies that are to be generated, then reasoned, we automated the launch of the tests. For this, we used shell scripts which also allowed each reasoning test to be performed in a clean, newly invoked Java Virtual Machine (JVM). Only the reasoning time was measured i.e. JVM start, and ontology load time was excluded. The newly invoked JVM avoids the risk of optimisations increasing later performance, although potentially introduces a small start-up overhead – given the overall length of the reasoning task this is probably not significant.

We tested two types of computer specifications:

C1 – Intel Core i7 CPU 920, 8 multi-core processor with 2.67GHz speed and 6 GiB RAM.

C2 – Intel Xeon E5335, 8 multi-core processor with 2.00GHz speed and 12 GiB RAM.

While **C1** can generate up to 10^5 karyotypes in one ontology, it can only reason up to 10^4 number of karyotypes. For 10^5 number of karyotypes, the reasoner is unable to complete the reasoning process due to insufficient memory. This conflicts with our initial requirement that there now exists around 10^5 ISCN Strings. However **C2**, can generate and reason up to 10^5 karyotypes[4]. These results suggest that The Karyotype Ontology can scale to individual lab size (10^4) easily while world scale (10^5) will require more resources.

On **C1**, serial generation and reasoning of 176,000 ontologies takes about two days to complete, while on **C2**, it took about a week to complete only 220 ontologies. It is because of this substantial increase of time taken, that for the rest of this chapter we will discuss the results produced by **C1**.

In order to ensure that the reasoning tasks are not affected by possible time-sensitive background computer processes, we randomise the reasoning tasks. The results of these tasks is shown in Figure 6.1. We can see no obvious correlation, so we conclude that our results are not affected by time-sensitive background computer processes.

[4]While it can also generate up to 10^6 karyotypes, it can only reason up to 10^5

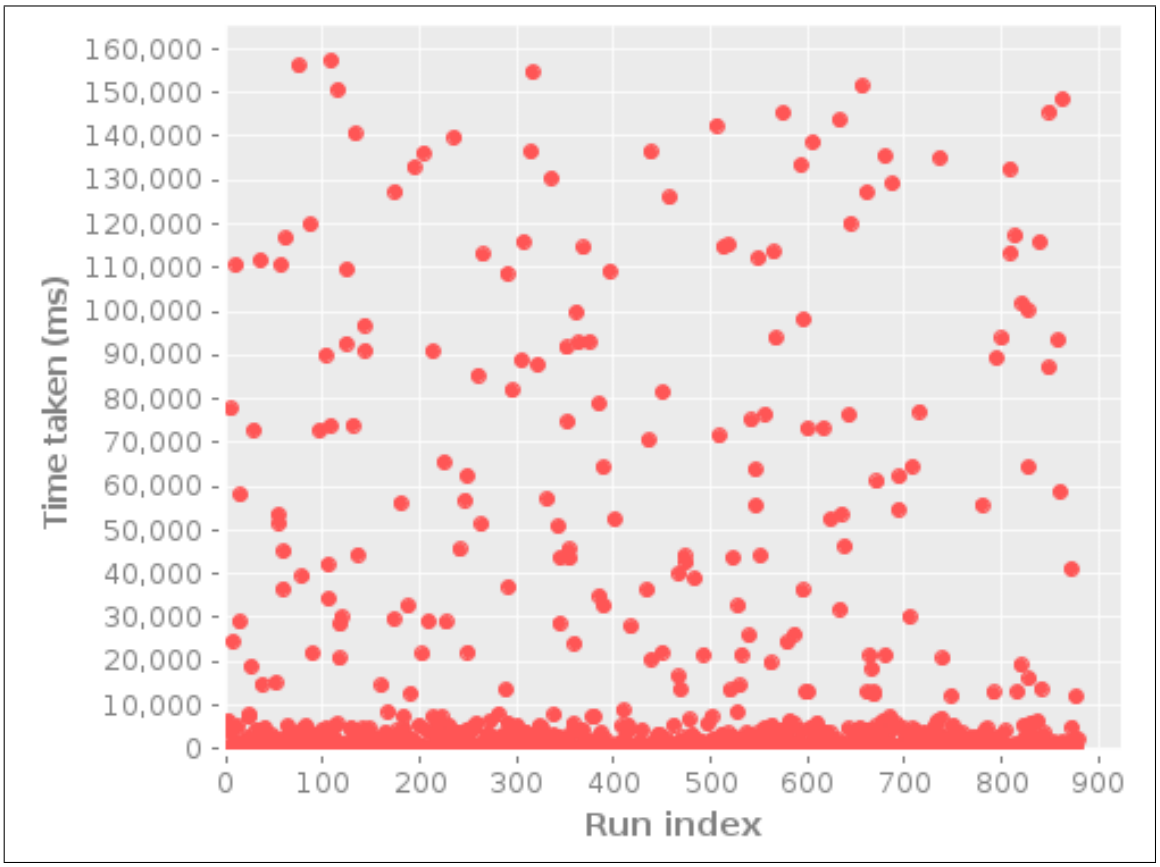


Figure 6.1: Scatter plot of the time taken versus the run index for randomised reasoning tasks.

6.4 Scaling The Karyotype Ontology

In this section, we investigate whether The Karyotype Ontology scales well by calculating the mean reasoning performance (R13), by reasoning over a set of random ontologies using HermiT [137][5]. The mean time taken to reason an ontology grouped by M number of event restrictions and K number of karyotypes, can be seen in Figure 6.2.

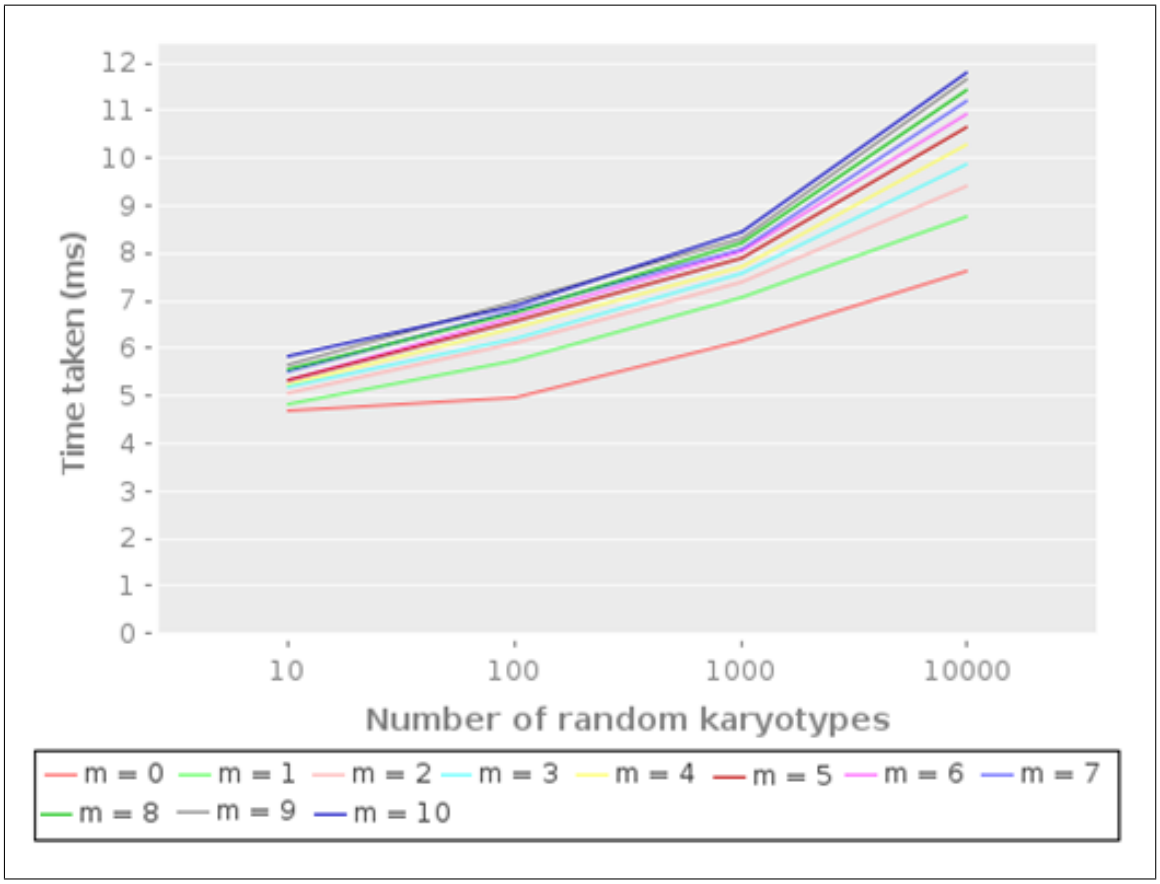


Figure 6.2: Line chart showing the mean reasoning times versus the number of random karyotype classes for a variety of restrictions. Each line represents the mean reasoning time taken for various M number of event restrictions.

From graph 6.2, we can see that, unsurprisingly, the reasoning time increases as the number of restrictions (M) increases, and the number of random karyotypes (K) increases.

[5]At this time of writing (2014) Tawny-OWL only supports two maven compliant reasoners: ELK see `http://code.google.com/p/elk-reasoner/` and HermiT see `http://github.com/phillord/hermit-maven`

6.5 Incorporating `affects` restrictions

In this section, we show an extension of The Karyotype Ontology (R14). We do this by incorporating `affects` restrictions, which is currently not modelled in The Karyotype Ontology. For example, a deletion or inversion event `affects` a sequence of bands between two breakpoints. In the thesis, we investigate three ways of implementing this relation and how these implementations effect reasoning performance.

The four representations are:

A0 null-affects no `affects` relation is incorporated.

A1 closure-affects all bands are named, and a closure axiom added.

A2 sequence-affects a variant of the sequence ODP.

A3 data-affects use of ordinal numbers for the chromosome bands as a datatype.

The diagrammatic representation of these `affects` implementations can be seen in Figure 6.3[6].

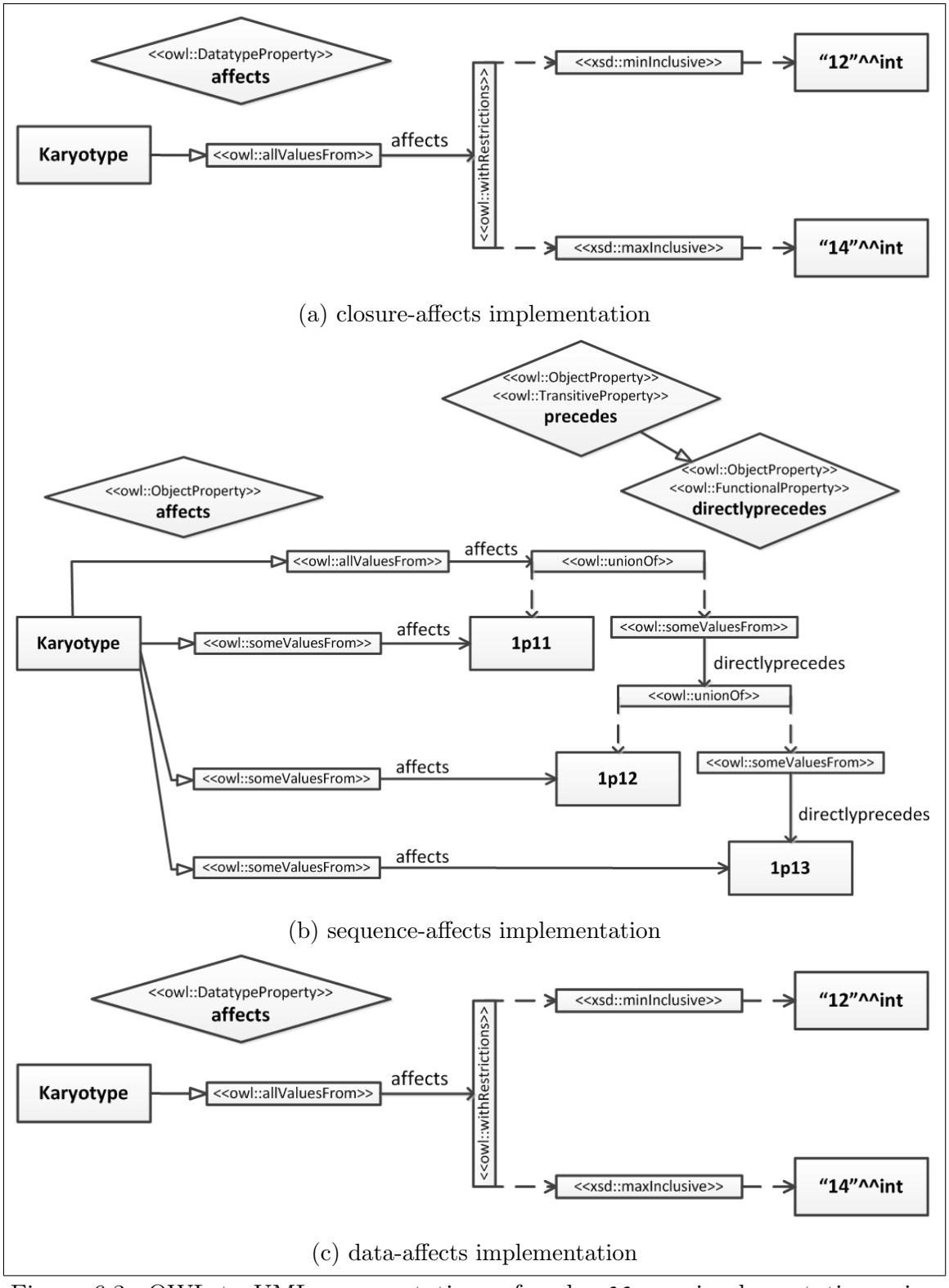
A priori, it is difficult to determine which of these implementations will work best, particularly with respect to non-functional characteristics such as reasoning time. With the use of Tawny-OWL, Tawny-Karyotype and HermiT we are able to test this by generating multiple test versions of The Karyotype Ontology (see Section 6.2).

6.5.1 The `affects` implementations

In order to demonstrate the three `affects` implementations, we will use the same example as shown in Listing 6.1. Here, we define a karyotype with an inversion restriction that involve breakpoints `1p11`[7] and `1p13`. Therefore the band expansion of the breakpoints `1p11` and `1p13` is an ordered sequence of three bands; `1p11`, `1p12` and `1p13`. As the implementations are encoded in different namespaces, name clashes are averted and the same function name is used, in this case `affects-band`.

[6] These diagrams are similar to the formal diagrams seen in the ODPs public catalog.

[7] In this chapter, chromosome band concept names have been simplified to ensure a concise syntax; for example, `1p11` actually refers to `HumanChromosome1Bandp11` in the ontology.



(a) closure-affects implementation

(b) sequence-affects implementation

(c) data-affects implementation

Figure 6.3: OWL to UML representations of each `affects` implementation using the exemplar in Listing 6.1.

```
(defclass k46_XX_inv!1!p11p13!
 :super
  (some derivedFrom k46_XX)
  (inversion 1 1p11 1p13)
  (affects-band [1p11 1p12 1p13]))
```

Listing 6.1: The usage used to define affects restrictions.

Affects type 1: (closure-affects) – One way of implementing the `affects` restrictions for karyotype classes that have event restrictions, is through the use of the generic closure axiom. Thus the usage pattern shown in Listing 6.1 will expand to generate four restrictions; three existential and one universal restriction (see Listing 6.2)[8].

```
(defclass k46_XX_inv!1!p11p13!
 :super
  (some derivedFrom k46_XX)
  (inversion 1 1p11 1p13)
  (some affects 1p11)
  (some affects 1p12)
  (some affects 1p13)
  (only affects (or 1p11 1p12 1p13)))
```

Listing 6.2: The expansion of an example karyotype with an inversion event and affects object property restrictions as a closure axiom.

The encoding of this pattern is simplistic and only requires the use of the predefined Tawny-OWL `some-only` generic pattern (see Section 4.2). The full pattern encoding is shown in Listing 6.3.

```
(defn affects-band [bands]
  (some-only affects bands))
```

Listing 6.3: The pattern used to define `affects` restrictions as a closure axiom.

In the existence of more than one insertion restriction, bands are collated. The resulting collection is then applied to the `affects-band` pattern.

Affects type 2: (sequence-affects) – The second way of implementing the `affects` relation in karyotypes, uses a variant of the sequence ODP [31]. Thus the usage pattern shown in Listing 6.1 will expand to generate four existential restrictions; one of which explicitly models the order of the bands (see Listing 6.4).

[8]In real usage, we would expect the list of bands to be generated automatically using Tawny-OWL

```
(defclass k46_XX_inv!1!p11p13!
 :super
  (some derivedFrom k46_XX)
  (inversion 1 1p11 1p13)
  (some affects 1p11)
  (some affects 1p12)
  (some affects 1p13)
  (some affects
   (and 1p11
    (some directlyPrecedes
          (and 1p12
           (some directlyPrecedes 1p13))))))
```

Listing 6.4: The expansion of an example karyotype with an inversion event and affects object property restriction implemented as a sequence ODP.

The full pattern encoding for this implementation is shown in Listing 6.5. The new Clojure part of this definition is the `clojure.core` `vector?`[9] function; this function checks if the given parameter is an instance of type vector (or not).

```
(defn affects-band [bands]
 (list
  (some affects bands)
  (if (vector? bands)
    (some affects
          (apply sequence-odp bands)))))
```

Listing 6.5: The pattern used to define **affects** restrictions as a variant of the sequence ODP.

This `affects-band` function makes use of a second pattern, namely the `sequence-odp` function (see Listing 6.6); a recursive function that uses the `directlyPrecedes` object property to implement the ODP. The new Clojure parts of this definition are the `clojure.core` `first`[10] and `rest`[11] functions; these return the head element and tail elements of a collection respectively.

[9] `http://clojuredoc.org/clojure.core/vector_q`
[10] `https://clojuredocs.org/clojure.core/first`
[11] `http://clojuredoc.org/clojure.core/rest`

```
(defn sequence-odp [args]
 (if (= 0 (count args))
  (first args)
  (and
   (first args)
   (some directlyPrecedes
         (sequence-odp (rest args))))))
```

Listing 6.6: Example implementation of the `sequence-odp` function.

In the existence of more than one insertion restriction, unlike the first implementation, it is possible to determine the order of bands by reasoning in OWL. Potentially, this would mean that the membership of directional classes such as `DirectInsertion` or `InverseInsertion` could be inferred rather than asserted (see Section5.4.4).

Affects type 3: (data-affects) – The third and last way of implementing `affects` restrictions is as a datatype property and assign ordinal numbers to chromosome bands. Thus the usage pattern shown in Listing 6.1 will expand to generate one restriction (see Listing 6.7[12]).

```
(defclass k46_XX_inv!1!p11p13!
 :super
  (some derivedFrom k46_XX)
  (inversion 1 1p11 1p13)
  (some affects (span >=< 12 14)))
```

Listing 6.7: The expansion of an example karyotype with an inversion event and affects datatype property restriction. Here, 12 = 1p11, 13 = 1p12 and 14 = 1p13.

The full pattern encoding for this implementation is shown in Listing 6.8. The new Clojure parts of this definition are:

- The Tawny-OWL `min-max-inc` function which applies an `OWLDatatypeMinMaxInclusiveRestriction` OWL API restriction.
- The `get-start-ordinal` and `get-finish-ordinal` functions finds the start and finish ordinal numbers respectively, for the first and last bands for a given collection of bands.

[12]The `span` Tawny-OWL function returns a numeric datatype restriction, identified by its first argument. In this example, `(span >=< 12 14)` returns a min max inclusive restriction, while the `><` argument would return a min max exclusive restriction.

```
(defn affects-band [bands]
  (let [start (get-start-ordinal bands)
        finish (get-finish-ordinal bands)]
    (some affects (min-max-inc start finish))))
```

Listing 6.8: The pattern used to define data-affects restrictions.

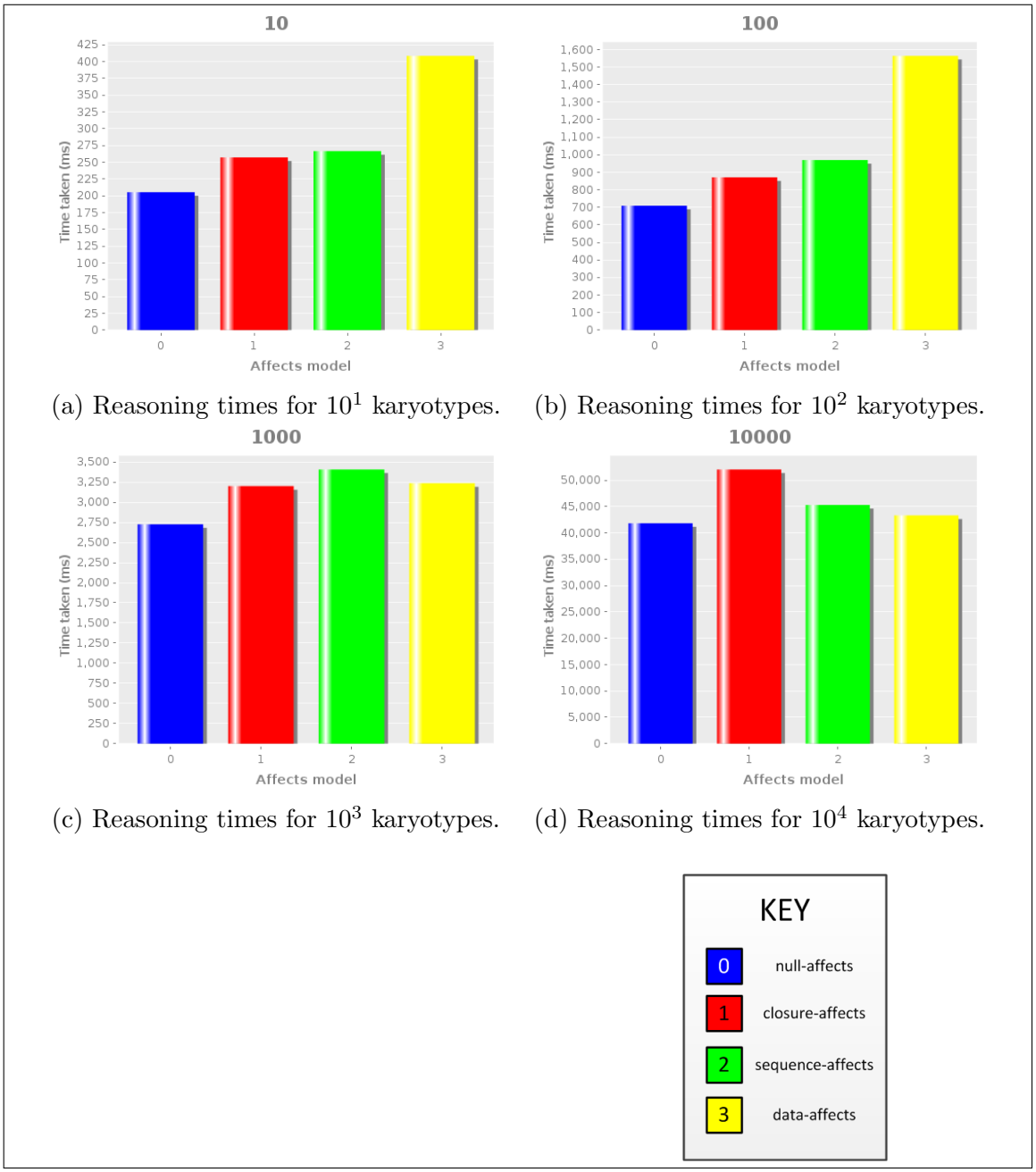
Similar to the sequence-affects implementation, the ordering of the bands is explicit within OWL and could be inferenced over.

6.5.2 Results

Using bash scripts and Tawny-OWL, we created 1200 refined ontologies – **N=100** for each affects implementation (A) and K number of karyotypes with **M=5** event restrictions. The mean time taken to reason an ontology grouped by K number of karyotypes then affects A implementation can be seen in Figure 6.4.

As with scaling in general (Figure 6.2), we can see (Figure 6.4) that the reasoning time increases as the number of random karyotypes (K) increases for each affects implementation. If we compare the three affects implementations we find that while the reasoning times for all implementations increases with the number of karyotypes, the data-affects which initially is the worst performer, scales to become the best, while the closure implementation is initially the best and becomes the worst. Generally the reasoning times for the sequence-affects implementation seems consistent, as it is mostly the second best representation of the three implementations (with the exception of 10^3 karyotypes).

From these results we are unable to deduce the best way to implement the `affects` relation due to the variability. However we can make the statements that for ontologies with $\leq 10^4$ karyotypes, the closure-affects implementation is the best for the `affects` relationship, while ontologies with $\geq 10^5$ karyotypes, the data-affects implementation is the best. In fact, the results suggest that all three implementations scale reasonably well with respect to the null representation (null-affects).



(a) Reasoning times for 10^1 karyotypes.

(b) Reasoning times for 10^2 karyotypes.

(c) Reasoning times for 10^3 karyotypes.

(d) Reasoning times for 10^4 karyotypes.

Figure 6.4: Bar charts showing the mean reasoning times for each affects implementation from 10^1 to 10^4 number of karyotypes. As shown in the key, the blue bar represents reasoning times for the original representation, while the red, green, yellow bars represent the reasoning times for closure-affects, sequence-affects, and data-affects implementation respectively.

6.6 Summary

In this chapter we have shown that with the use of multiple generated test ontologies (see Section 6.2), we are able to successfully test the performance (R12), scalability (R13) and extensibility (R14) of The Karyotype Ontology thus successfully proven that The Karyotype Ontology has met the requirements (identified in Section 5.2.2).

This was accomplished due to the main benefit of a pattern-driven and programmatic approach to ontology engineering (RQ3); once encoded we can automatically and consistently (re-)generate ontologies easily and quickly. This means that any changes in the code also effects the resulting ontology. As shown in Sections 6.2 we use this ability to generate many random ontologies (i.e. test ontologies with random karyotypes) of various sizes.

As shown in Section 6.4 we can use our random ontologies to determine how well the computational representation scales. From our results we see that The Karyotype Ontology generation and reasoning of 176,000 ontologies (that contain more than 200,000 axioms), is plausible on two desktop machines. While the slightly older computer (**C1**) was unable to reason ontologies 10^5 karyotypes, due to the limited memory available, the other computer (**C2**) was successful in this plight. This suggests that we should be able to reason without any more resources.

As shown in Section 6.5 when we can apply new patterns to our random ontologies we can quantitatively determine the effect different modelling representations have on an ontology by how well they scale. We showed the reasoning results for three different representations of an event which `affects` numerous chromosome bands. Generally the results tell us that there are two viable ways of implementing the `affects` restriction; the enumeration of bands (closure-affects) and the datatype restriction of ordinal bands (data-affects), depending on the number of karyotypes we wish to model. However, the non-functional scalability requirement is probably not the best basis to make for this design decision. Instead, with Tawny-OWL, we can keep all three implementations and allow downstream users of The Karyotype Ontology to decide which implementation is best for them.

To summarise, we have shown a novel means to explore modelling choices and how

well they scale.

In the next chapter, we will investigate whether this programmatic and patternised approach to building ontologies is also possible for another interesting domain of biology; mitochondrial disease.

7

The Mitochondrial Domain

Contents

7.1 Introduction 132
7.2 Stage 1 – Term Capture 135
7.2.1 Term of the week 135
7.2.2 Lab meetings 136
7.2.3 Published papers 136
7.2.4 Assessing the term capture techniques 139
7.3 Stage 2 – Competency Questions 140
7.4 Stage 3 – Refinement 141
7.4.1 Canonicalising terms 141
7.4.2 Identifying disease relevant terms 142
7.5 Stage 4 – Construction 143
7.5.1 Constructing The Mitochondrial Disease Ontology classes . . 144
7.6 Stage 5 – Evaluation 148
7.7 Summary 149

7.1 Introduction

The knowledge about mitochondria is complex and rich; we wished to build an ontology representing some of this knowledge so that we could provide a formal and computational interpretation of mitochondria and mitochondrial disease. This also provides an opportunity to extend the programmatic and patternised ontology development methodology developed and described in Chapter 5 to a new domain. In contrast to The Karyotype Ontology, however, there is no existing specification that we wished to formalise; rather the knowledge is represented in databases, academic papers and individual academics. For this work, therefore, we start by describing our experiments with knowledge capture, where we attempt to illicit the domain; we then move onto formalising the knowledge, and the incorporation of existing resources, into The Mitochondrial Disease Ontology.

An overview of the methodology being used to build The Mitochondrial Disease Ontology can be seen in Figure 7.1. As with most ontology engineering approaches, we can split the methodology into different stages. The five stages are:

S1 Term Capture – acquire knowledge about the mitochondrial domain

S2 Competency Questions – identify interesting questions about the mitochondrial domain

S3 Refinement – filter terms and competency questions with respect to the ontology's purpose and scope i.e. in this case inherited mitochondrial disease.

S4 Construction – build the ontology

S5 Evaluation – evaluate the ontology

While these different stages could be pursued in a traditional waterfall-style, we instead take a more agile approach. This is for two main reasons: first, the time required for the individual stages is hard to judge, which makes planning the process to fit within the time available difficult; second, maintaining the interest of the domain experts is not served by a lengthy process with no apparent end-outcomes.

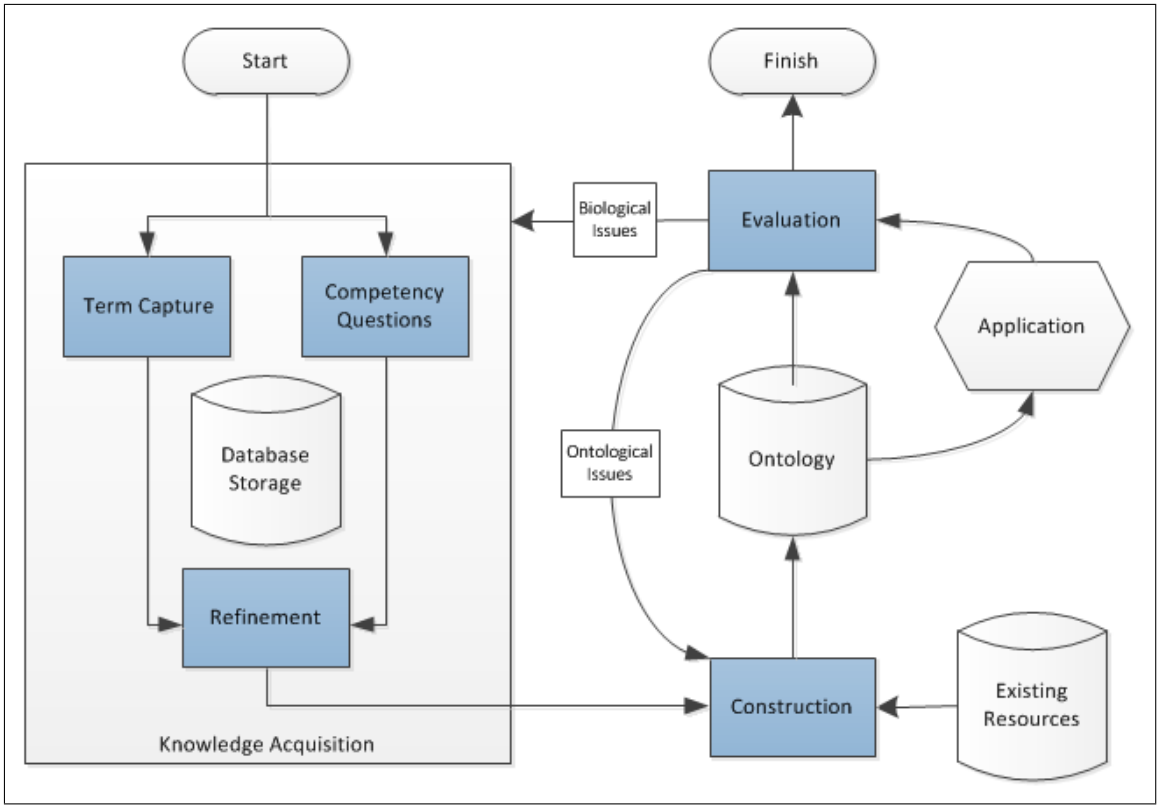


Figure 7.1: An overall pictorial representation of the methodology discussed in this chapter. The fives stages have been highlighted in blue. The first three stages are combinatorially known as the knowledge acquisition stage of the methodology that results in a database of refined terms and competency questions. Feedback from the evaluation stage is used for the next iteration of the ontology – as with software development, the ontology model will need multiple iterations.

Within this chapter, we describe a more in-depth illustration of the methodology used for the construction of The Mitochondrial Disease Ontology; we explore numerous challenges and apply a number of heuristics to overcome these challenges. Within each stage of the methodology we provide the results obtained and lessons learned. The corresponding code and supplementary data for the construction of The Mitochondrial Disease Ontology can be found at the Project Website[1].

[1] `https://github.com/jaydchan/tawny-mitochondria`

7.2 Stage 1 – Term Capture

The term capture stage is used to acquire knowledge about the domain of interest, in this case mitochondria and to identify relevant terms. The approach we use for extracting terms was naïve; no linguistic processors were used in the extraction of terms, instead we used our experience (and our lack of mitochondrial knowledge) to identify these terms. For the purpose of acquiring knowledge, we describe three different techniques: "Term of the week", lab meetings and published papers. For some of these techniques, we required access to domain experts; in this case, our local domain experts are members of The Mitochondrial Research Group (MRG) (known as The Wellcome Trust Centre for Mitochondrial Research[2] since 2012) based at Newcastle University. In this section we explore the (potential) effects of the three techniques.

7.2.1 Term of the week

Here, we describe the *Term of the week* process, pioneered by Dr Frank Gibson and Dr Phillip Lord as part of the Code Analysis, Repository & Modelling for E-Neuroscience (CARMEN) project[3], as it was considered for use in building of our ontology. In this technique, group mailing lists were exploited; every week a term (chosen by the ontologists) was sent to the mailing lists to illicit definitions. In general, this approach had worked well in soliciting engagement from some domain experts. However it also produced significant negative feedback from others, due to the volume of email traffic; the process was halted after several months. The main outcome was to demonstrate to domain experts that term definitions were not consistent (i.e. *not* "everybody knows what that means") and that, in fact, considerable diversity of understanding existed.

For this project, we decided not to use this technique as we were unable to identify a suitable mailing list; existing MRG mailing lists were used largely for other purposes. Using these lists, or even bulk mailing, was considered likely to have alienated many

[2] `http://www.newcastle-mitochondria.com/`
[3] `http://www.carmen.org.uk/`

domain experts as found earlier or, worse, been considered spam. We considered that this risk was too great, as we have only a single set of experts to work with. What we require is a technique that will not disrupt the daily lives of our domain experts.

7.2.2 Lab meetings

At the time of this work[4], we attended weekly lab meetings held by MRG. In these meetings, the experts would take turns to present their current research, in a semi-formal presentation. The purpose of these meetings was to update and receive feedback from their colleagues. In attending these meetings, we "captured" words or phrases that we thought were important about the mitochondrial domain. Words that we did not understand were also captured as, they were quite likely to be of high information content.

While this technique was informative as to what current research is being carried out, we lacked the expertise and so found the knowledge to be too specialised and difficult to understand. Also as the mode of communication was mainly oral capturing the spelling of terms was problematic; however with supplementary knowledge, corrections could be made at a later date. Lastly, we found that as well as capturing mitochondrial domain information, environment specific information such as (preferred) equipment names, samples, cell lines and so on, were also captured. Therefore we conclude that as well as not disrupting the daily lives of our domain experts (see Section 7.2.1), we require a form of written communication (similar to Term of the week) and a broader range of domain experts (to lessen the amount of environmental specific information).

7.2.3 Published papers

A tried and tested approach for term gathering is simply to read published papers and manually extract related terms. This allows access to mitochondrial data from at least the past ten years (although introduces a time lag from publication delay).

[4] 2010-2011

One significant problem here is that there are too many papers to read and process with only one person employed full-time. Our solution was as follows: papers were selected randomly from PubMed [125] that were published between the years 2000 and 2011; for each batch of papers, we noted the number of new terms, which allowed us to judge when the process was saturating.

A total of 3666 terms from 30 papers were extracted using the published paper data source (see Table C.1 for paper details)[5]. However when we collate these terms we find that while many are unique for each paper some exist across numerous papers (i.e. are duplicates). For example the term **melas** was manually identified as a term in five different papers [21, 25, 124, 148, 171]. Once we removed these duplicates, 3311 terms remained.

The number of unique terms captured (without duplicates) using this method for every batch of five papers is shown in Figure 7.2. Generally, the number of new terms is seen to decrease over time. However the figure also shows that a large number of new terms per batch (approximately 250 terms per 5 papers) are still being extracted. It is clear that the process is far from saturated, with the production of biological knowledge outstripping the ability to describe it; this is, however, a common problem [12].

Sample Cases: The following examples show the terms extracted from one paper from the mitochondrial corpus [148]. The terms are **highlighted**.

Sample 1:

> “For **LHON**, a strict maternal pattern of inheritance was evident and point mutations involving the **ND family** of genes that encode subunits of **complex I** were identified.”

[5]Terms extracted from each paper are available at `https://github.com/jaydchan/tawny-mitochondria/tree/master/resources/input/Terms`.

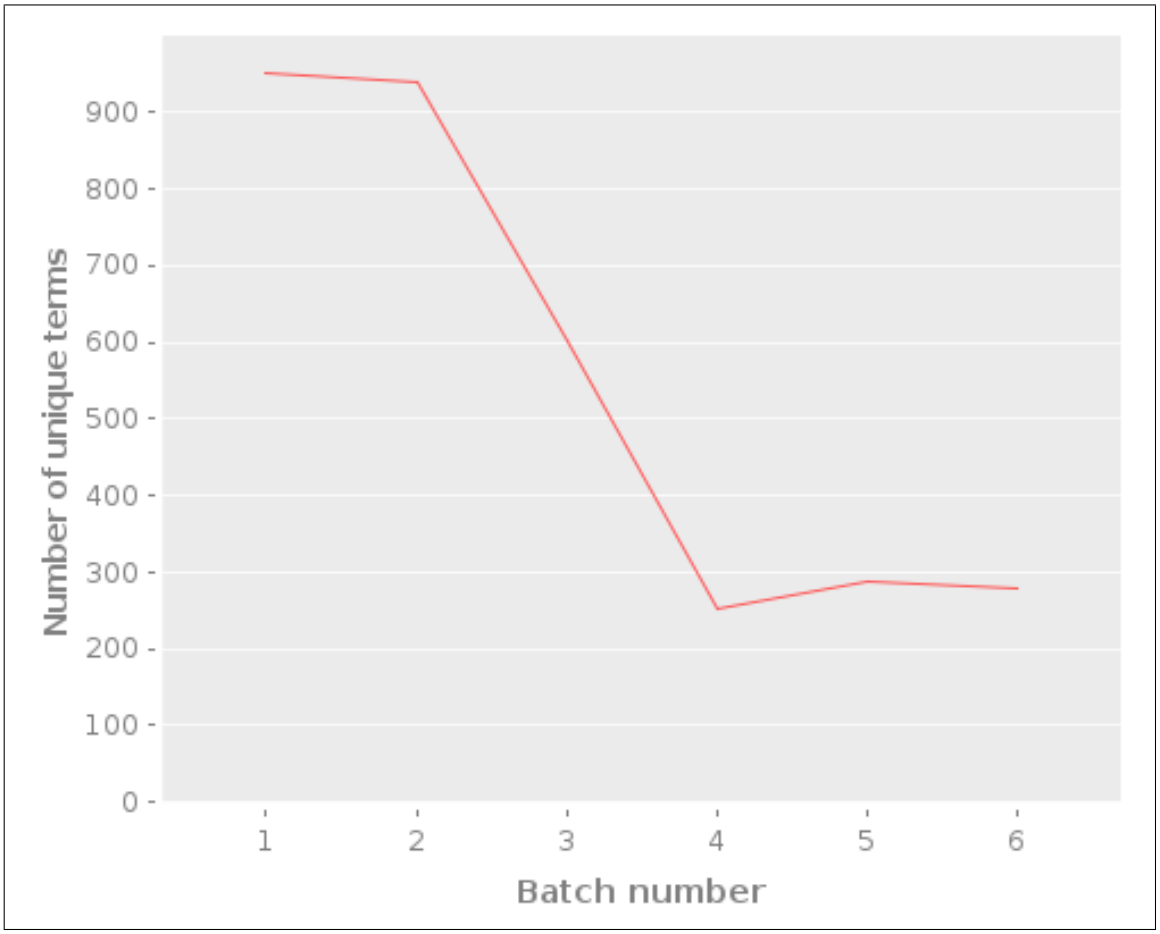


Figure 7.2: Line plot showing the number of new terms extracted from each batch (five) of papers.

Sample 2:

> “There has been considerable interest in the possibility that **mitochondrial DNA (mtDNA)** variants might predispose to common diseases; for example, **diabetes**, **Alzheimer disease (AD)** and **Parkinson disease (PD)**.”

Sample 3:

> “Mitochondria are found in all nucleated cells and are the principal generators of cellular Adenosine TriPhosphate (ATP) by **oxidative phosphorylation (OXPHOS)**, incorporating the **electron-transferring respiratory chain** (**complexes I-IV**) and the **ATP synthase (complex V)**.”

7.2.4 Assessing the term capture techniques

The techniques described here result in somewhat different forms of knowledge: when used Term of the week results in fewer terms, but with (multiple) *ad hoc* definitions, extracted from email threads; lab meetings result in less formal transcripts often without references, but which represent local knowledge; terms extracted from published papers result in many referenced terms, but provide no definitions and may contain gaps in community knowledge (due to the publishing time-lag).

For the first iteration of our ontology, we found the published paper technique the most useful and appropriate knowledge source. Through this technique we have identified numerous relevant and referenced mitochondrial terms that: were indirectly provided by the global mitochondrial community and did not disturb the normal lives of our experts; are less likely to be environmentally specific; and are (most likely) spelt correctly. However this technique is labour-intensive and time-expensive, thus as future work, we may investigate use of Term of the week and lab meetings for later iterations of the ontology, when our aims and objectives are clearer to our local domain experts, MRG.

7.3 Stage 2 – Competency Questions

The breadth of knowledge about mitochondrial biology is significant; attempting to represent this with the resources available is impractical. We therefore compiled an informal list of competency questions, which define those questions which our ontology should reasonably be able to answer (see Section 7.6). In total we identified 133 competency questions[6].

Sample Cases: Example competency questions that relate to the sample terms extracted from the term capture task (see Section 7.2.3):

- What are all the (point) mutations that are associated with a specific syndrome?
- What are all the (point) mutations that are associated with a specific gene/protein?
- What are all the genes/proteins that are associated with a specific syndrome?

[6] A list of identified competency questions is available at `https://github.com/jaydchan/tawny-mitochondria/blob/master/resources/input/cq.txt`.

7.4 Stage 3 – Refinement

The refinement task is used to filter the terms and competency questions, ensuring that these meet the overall direction of the research (in this case mitochondrial disease). The refinement task involves reviewing both terms and competency questions against the following criteria:

- **Relevant:** related to mitochondrial pathology
- **Representable:** plausible to ontologically represent

The last of these was interpreted quite broadly – we made no judgement at this stage whether a term or competency question would be easily or well represented.

The refinement process was largely done without any domain expert engagement, again due to the labour-intensive nature of the work. This brings with it the risk that the resultant ontology will represent a domain that is too specific or broad; we believe that subsequent iteration through earlier steps should help to alleviate this possibility.

Both the terms and competency questions were reviewed; any which failed the stated criteria were labelled as out-of-scope, and "quarantined". In the case of terms, this will avoid subsequent reintroduction of the material, while quarantined competency questions ("incompetency questions") will be later used to evaluate the ontology. At this stage, we also canonicalised terms, in an attempt to remove acronyms, synonyms and other forms of duplication, where this was not obvious at the initial capture stage[7].

In this section we discuss the results of the canonicalisation and filtering processes.

7.4.1 Canonicalising terms

As mentioned in the term capture stage (see Section 7.2), we utilised a naïve approach to extracting terms – terms that we thought were important in the mitochondrial domain or obscure to us were extracted. However this approach resulted

[7]In order to aid the canonicalisation process, terms were converted and stored in lower case.

in various forms of duplication, such as synonyms and acronyms; for example **ant** is the acronym for **adenine nucleotide translocator**. Therefore we removed these duplication, using simple string searches.

These string searches predicted that we had collected 5644 duplicates and 128 acronyms. Through manual evaluation we found that there was 222 duplicates and 22 acronyms. Having now identified these duplications we can decrease the number of terms we need to refine and therefore model. The total number of canonicalised capture terms we have at the end of this task is 3061.

7.4.2 Identifying disease relevant terms

Now that we have been able to reduce the number of terms, we can now move onto filtering the canonicalised terms. Table 7.1 shows the statistics behind the refinement stage. As can be seen in the table, we are attempting to keep a relatively narrow scope, avoiding in their entirety some areas if they are likely to produce a large increase in the workload. Maintaining a tightly-defined scope has previously been identified as a key factor in the success of GO [7].

Table 7.1: Table showing the number of terms and competency question found to be in-scope or quarantined.

	Number of Terms	Number of Competency Questions
In-scope	2174	125
Quarantined	887	8
Total	3061	133

7.5 Stage 4 – Construction

In this section, we aim to formalise the knowledge extracted from our earlier elicitation stages. Initial attempts to do this were done using Protégé, RightField and Populous (which uses OPPL) but found the separation between the three environments made the process difficult to manage. Instead, we now use a semi-automated construction using Tawny-OWL and localised patterns. There are numerous reasons for this decision:

Firstly, from the term gathering stage it is clear that many of the terms (such as mitochondrial genes and proteins) are already available in a structured form either in another ontology or, more commonly, in a database such as Online Mendelian Inheritance in Man (OMIM) and MitoMiner [139]. We wish to have the option of reusing this work. The incorporation of external knowledge is a common best practise in ontology engineering; for example in The Semanticscience Integrated Ontology (SIO) (see Chapter 8), 118 chemical elements are defined and related to the ChEBI database. In this case, there are many more terms we could incorporate; for example human anatomy as well as mitochondrial anatomy, diseases, genes, proteins and mutations. Through the use of Tawny-OWL, we can build patterns, read the terms from file and translate them into the ontology.

Secondly, contradictory knowledge; this causes obvious issues with integration into an ontology. For example, we might have two statements, "mutation 1 cause disease" and "mutation 1 does not cause disease". Ontologically, this is contradictory and may result in unsatisfiable classes in OWL [131]. One solution to this is to use reification. So "mutation 1 causes disease according to person X" and "mutation 1 does not cause disease according to person Y". These statements are no longer contradictory. However, reification would prevent us from using the ontology to discover contradictions which is, itself, a useful feature of an ontological model. With Tawny-OWL, we can separate out the full axiomatisation from the immediate representation; we can change the axiomatiation later consistently. We can even have both representations under different circumstances, an ability exploited in Chapter 6. As shown in [78], retaining the flexibility in our representation of the biological

knowledge is often useful for many reasons.

Finally, importing other ontologies or databases will potentially allow us to generate a large number of terms rapidly, which raises the spectre of scalability issues when reasoning, as seen with karyotypes (see Chapter 6). More specifically, in [65], the author has shown poor performance occurs when describing all mitochondrial genes and then asserting them as disjoint which, though true, was not necessary for our application. We may wish to change our representation to use simpler definitions or a less expressive, but computationally cheaper, OWL profile (such as EL). This is hard to plan for upfront, as the complexity of reasoning for a given ontology is hard to predict *a priori*.

To summarise, Tawny-OWL enables us to import knowledge from a variety of sources which carry much of the background knowledge. It should also allow us to retain the flexibility we need with respect to representation and expressibility that we use. In this section, we briefly discuss the construction of The Mitochondrial Disease Ontology classes, as well as provide the basic statistics for the first iteration of The Mitochondrial Disease Ontology.

7.5.1 Constructing The Mitochondrial Disease Ontology classes

Once, we have our identified our in-scope and quarantined terms and competency questions, we can move onto building the ontology. However, before we incorporate the refined terms identified in the previous stage (see Section 7.4), we want to first incorporate any existing (structured) data found in various sources, such as online databases and articles. To accomplish this task, we first identify generic bins or categories that are related to mitochondrial disease. These categories are: mitochondrial anatomy, diseases, genes and proteins, as well as human anatomy. For each, we discerned all associated child terms (see Table 7.2) and subsequently encoded and incorporated these terms into The Mitochondrial Disease Ontology, using localised patterns (see Chapter 4).

In addition, mutations (DNA and protein) are useful, as proved by their presence in the in-scope terms and competency questions. However, unlike mitochondrial genes

Table 7.2: Table showing the type, number of and data source for each generic The Mitochondrial Disease Ontology class.

Class type	Count	Data source
Disease	41	The United Mitochondrial Disease Foundation (UMDF) website
Gene	761	The NCBI Gene portal
Human Anatomy	61	The Terminologia Anatomica (TA) is the international standard on human anatomic terminology.
Mitochondrial Anatomy	15	The Mitochondrial Research Group (MRG) website
Protein	479	The UniProt Knowledge Base (UniProtKB)

and proteins, there is no comprehensive list of mitochondrial mutations available to incorporate into the ontology. Instead we can use the mutations nomenclature[8] and pattern matching to naïvely identify these mutations from our refined terms. The regular expressions used to identify DNA mutations and protein mutations are shown in Listing 7.1. After manual inspection of the filter results, we find that 6 (of 6) were correct DNA mutations, while 0 (of 36) were correct protein mutations. This shows that whilst the protein mutation regular expression is too generic, as we also found genes and proteins, the DNA mutation regular expression is too specific, as it does not include other valid DNA mutations found in another format, e.g. `g8993t`.

```
DNA mutation: "[acgt]>[acgt]"
Protein mutation:
 "[gpavlimcfywhkrqnedst]\d+[gpavlimcfywhkrqnedst]"
```

Listing 7.1: Applied regular expressions for identifying mitochondrial mutations.

Now, we can start to incorporate the refined terms into the ontology. However, since we already have ∼1300 classes in the ontology that are related to mitochondria, it is possible that some of the refined terms already exist in the ontology, therefore, we require further canonicalisation of terms. Thus, as we incorporate the terms we check to see if the term is equivalent to any existing classes. If it is, then the class is edited to include paper provenance information. An example of an existing term that has been updated with provenance information is shown in Listing 7.2. Those terms that are not found pre-existing are also incorporated into the ontology. This

[8] `http://www.hgmd.cf.ac.uk/docs/mut_nom.html`

incorporation task is aided with the use of localised patterns. After incorporation we find that 45 (of 2174) refined terms were pre-existing in the ontology.

```
(class melas
 :label "melas"
 :super
   Disease
   (see-also "OMIMID:540000")
   (see-also "Mitochondrial Encephalomyopathy Lactic Acidosis
    and Strokelike Episodes")
   Term
   (source paper7))
```

Listing 7.2: Example of an existing term in The Mitochondrial Disease Ontology. The `source` function ensures that each term has a `hasSource` relation.

In total, The Mitochondrial Disease Ontology has 3532 classes, 3 annotation properties and 2 object properties; see Table 7.3 for the complete class statistics and Figure 7.3 for a visual representation of The Mitochondrial Disease Ontology top-level structure. Now complete, we can focus on the different evaluation techniques that could be applied to The Mitochondrial Disease Ontology.

Table 7.3: Table showing the type and number of implemented classes in The Mitochondrial Disease Ontology.

Class Type	Total Number	Number of Refined
Disease	42	10
Gene	762	19
Human Anatomy	62	1
Mitochondria	1	0
Mitochondrial Anatomy	16	7
Mutation	1	0
DNA Mutation	7	6
Protein Mutation	1	0
Paper	31	0
Protein	480	3
Term	2129	2128
Total Number	3532	2174

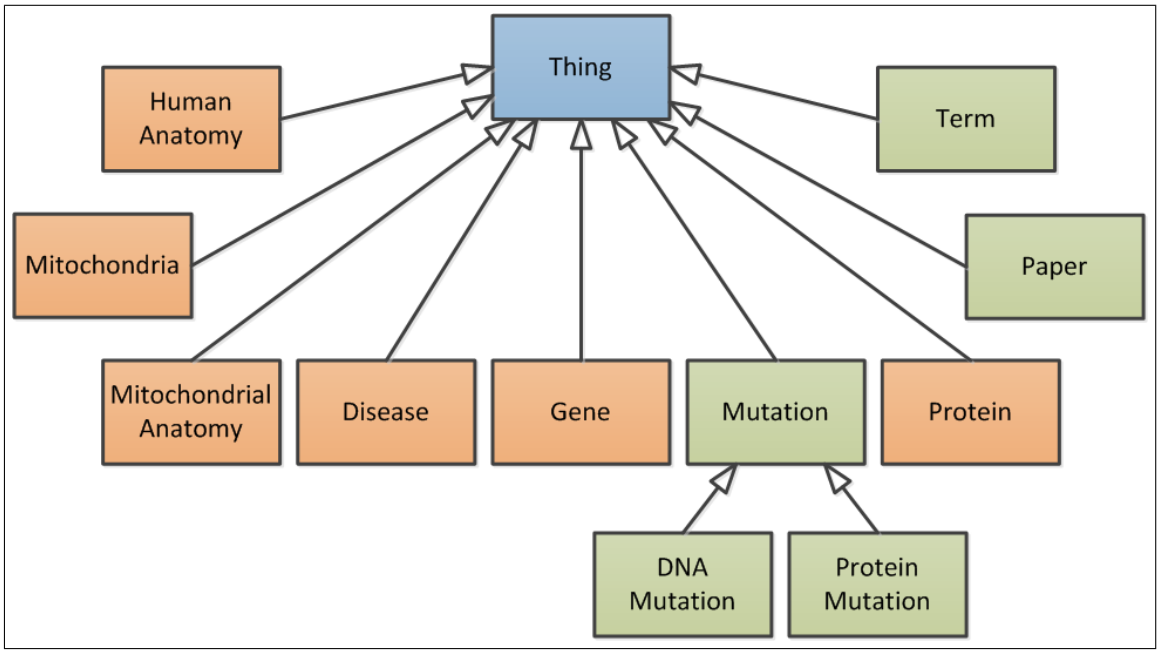


Figure 7.3: The top-level structure of The Mitochondrial Disease Ontology. Classes that were imported from external sources are coloured in orange, while classes that were extracted from the paper are coloured in green.

7.6 Stage 5 – Evaluation

There is no standard way of evaluating ontologies, however four potential methods to evaluate the ontology are discussed in [42]:

Specification evaluation – utilises the identified competency and incompetency questions. The ontology must have the ability to answer all of the in-scope competency questions. This can be achieved through the use of competency and incompetency questions: the former should be answerable, while the later should not be.

Terminology evaluation – originally proposed in [44], this determines ontology accuracy by checking definitions for consistency, completeness and conciseness.

Taxonomy evaluation – evaluating the structure of the ontology by testing the axioms for inconsistencies using reasoners such as Pellet[9], HermiT[10] or Fact++[11].

Application-dependent evaluation – to evaluate the ontology's fitness for purpose.

Of these, application-dependent evaluation is perhaps of most interest to us, as this allows evaluation of the ontology in the context of an otherwise useful task; for example, classifying phosphatase proteins [166], or comparing annotation similarity with sequence similarity [69]. The usefulness of the task increases the likelihood that domain experts will be willing to contribute time to it; this is particularly the case if, as with the classification of phosphatases, it raises the possibility of new biological understanding. Further work is required in order to implement this.

[9] `http://clarkparsia.com/pellet`
[10] `http://hermit-reasoner.com/`
[11] `http://owl.man.ac.uk/factplusplus/`

7.7 Summary

In this chapter, we discuss the progress and steps taken to provide a reference ontology, that describes the terms associated with mitochondrial disease.

Unlike The Karyotype Ontology, there is no standard nomenclature for mitochondrial knowledge, which meant that some form of knowledge capture was required. In order to accelerate this knowledge capture, we tried to implement some techniques (e.g. implement "Term of the week" and attend lab meetings) but found limited success. Instead, we utilised the tried and tested (i.e. trusted) method of reading and capturing our knowledge from published papers. However, even the methodological reading of papers has one major limitation; the seemingly infinite amount of papers that contain vital mitochondrial knowledge. Indeed, our statistics demonstrate that our term gathering is far from saturated, suggesting that the end product will have a far from complete coverage.

In further iterations of the ontology, we might need to consider incorporating gamification in order to make the term capture stage entertaining and encourage participation as this has been shown to be beneficial. In particular, the iCAPTURer methodology [46] demonstrated the construction of a simple ontology at the cost of "3 t-shirts, 4 coffee mugs and one chocolate mousse". Or a more recent study has consciously attempted to adapt techniques from agile software development and apply them to ontologies resulting in The Agile Ontology Development (AOD) [23, 81, 82].

While The Mitochondrial Disease Ontology is far from complete, we have still shown that a pattern-driven and programmatic approach is also useful in representing this domain knowledge. In addition, the resulting ontology, The Mitochondrial Disease Ontology, is our first step in building a complete ontology which will potentially have the ability to classify and clarify mitochondrial disease by their symptomatic and/or genomic definition. Thus this chapter concludes our research into the application of our approach into two novel areas of biology in order to produce two novel bio-ontologies (RQ2).

Whilst lots of the mitochondrial knowledge can only found in papers, there are many databases and online resources that also contain vital information. With the use of

our approach and localised patterns we were able to incorporate the mitochondrial knowledge from a variety of sources and different formats (RQ3).

In the next chapter, we aim to prove that pattern-driven and programmatic approach can also be applied to an existing bio-ontology in order to highlight further benefits of this approach.

8

Patternised Development of an Existing Ontology

Contents

8.1 Introduction 152
8.2 Non-patternised rendering of Tawny-SIO 153
8.3 Patternised refactoring of Tawny-SIO 157
8.4 Tawny-SIO errors 163
8.5 Patterns for downstream usage 166
8.6 Summary 170

8.1 Introduction

While we have shown in detail that localised patterns are beneficial within the development of highly patternised novel ontologies, we need to prove that this methodology is useful to other ontologies. To test this, we have taken an existing bio-ontology and re-written it using Tawny-OWL, refactoring it into a patternised form.

The chosen ontology is SIO [33], a simple upper OWL ontology, useful for the integration of types and relations that provide rich descriptions of objects, processes and their attributes. The ontology (version `1.0.10`) defines 1414 classes, 203 object properties, 1 datatype property and 8 annotation properties. We have chosen SIO as it is explicit in promoting ODPs to describe and associate numerous entities such as databases and measurements. The ontology is available on the SIO wiki (and BioPortal) as an OWL2 file[1].

An overview of the workflow used for this investigation is shown in Figure 8.1. The three steps are:

1. **Read and Render** the original ontology (`sio.owl`) into Tawny-OWL syntax. This step results in the non-patternised version of Tawny-SIO.

2. **Refactor** the transformed ontology that uses common patterns identified from visual inspection. This step results in the patternised version of Tawny-SIO.

3. **Implement** localised patterns for downstream usage, identified from the SIO wiki. This step results in the patternised version of the downstream (exemplars) ontology.

In this thesis, *SIO* is used to refer to the existing ontology, whilst *Tawny-SIO* refers to the Tawny-OWL refactoring of SIO. The corresponding code and supplementary data for our application of a programmatic and pattern-driven approach to Tawny-SIO can be found at the project website[2].

[1] `http://semanticscience.org/ontology/sio.owl`
[2] `https://github.com/jaydchan/tawny-sio`

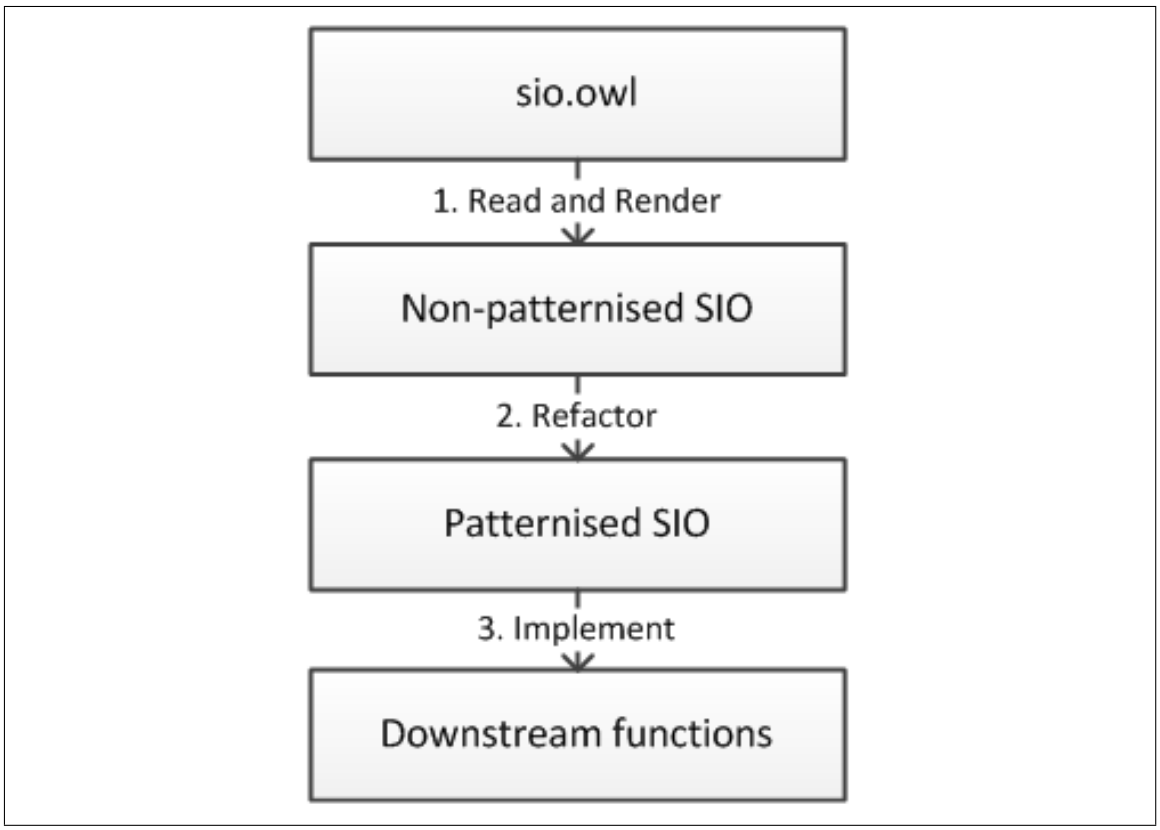


Figure 8.1: Overview of the SIO workflow.

8.2 Non-patternised rendering of Tawny-SIO

SIO was not developed using Tawny-OWL and is only available as an OWL2 file. Therefore, in order to enable the refactoring of SIO, we must first transform this into Tawny-OWL expressions, such that, when evaluated, it results in an ontology similar (i.e. semantically alike) or identical (i.e. syntactically and semantically alike) to the OWL version of SIO.

Secondly, we wish to make SIO appear as an ontology created natively with Tawny-OWL. This means that we want to be able to refer to SIO classes with symbols, rather than strings, or full IRIs. However SIO uses numeric identifiers as the *fragment*[3] of its URL, and while there are good reasons for this, it means that the fragment is unsuitable as a human memorable identifier at code level.

Therefore, we choose a part of SIO entity to transform into the usable name; the

[3]The technical definition of "fragment" is relatively involved, but in many cases is the same as the anchor of a Uniform Resource Locator (URL)

obvious choice is the `rdfs:label`[4]. However these label values may not be valid Clojure names. Therefore, we apply a simple syntactic transformation, with a few specific replacements for SIO entity names that would otherwise transform into either: reserved words (e.g. `true` and `false`), Tawny-OWL functions (e.g. `annotation` and `label`)[5], invalid Clojure names (e.g. `e.coli`)[6] or cause an OWL API parser error (e.g. `implies (->)`[7]). The short list of specific replacements is shown in Table 8.1. In this case, the original SIO identifiers are lost from Tawny-SIO, as they were not explicitly needed for this work; it would be possible to recover these if necessary.

Table 8.1: Table showing replacements for the short list of SIO entities.

SIO name	Tawny-SIO name
`annotation`	`_annotation`
`e.coli`	`e_coli`
`false`	`_false`
`implies (->)`	`implies`
`label`	`_label`
`true`	`_true`

When we applied this name transformation to version `1.0.10` of SIO, we found that two entities (`SIO_000944` and `SIO_001246`) have the same `rdfs:label` (namely `interval`). This is defined as pitfall **32** according to The OntOlogy Pitfall Service! (OOPS!) catalogue[8], and we interpret this as an error. Historically[9], we see that this error was introduced in version `1.0.6` of SIO, when `SIO_001246` was added to the ontology even though `SIO_000944` already existed. Therefore in order to differentiate between these two classes in Tawny-OWL, the original OWL file was edited such that the `SIO_001246` class (a subclass of `set`) now has the `set_interval rdfs:label`, while `SIO_000944` continues to have the `interval rdfs:label`. This is the first of many discoveries summarised in Table 8.3. This table is expanded on later in this

[4] `http://www.w3.org/TR/rdf-schema/#ch_label`

[5] We could avoid the namespace clash by not importing `tawny.owl/annotation` and `tawny.owl/label`, then specifying the namespace of both when in use.

[6] In Clojure, the full stop character (`.`) is used to designate a fully-qualified class name (e.g. `java.util.Properties`) or a name in the namespace.

[7] The OWL API parser for OWL/XML syntax cannot handle a greater than character (`>`) in the IRI.

[8] See `http://oeg-lia3.dia.fi.upm.es/oops/catalogue.jsp`

[9] Using the OWL submissions provided on the BioPortal website, see `http://bioportal.bioontology.org/ontologies/SIO`.

Section 8.4.

The various utilities in Tawny-OWL for rendering and reading ontologies have been written to deal with the transformation and Clojure names. Once transformed and rendered, these Tawny-OWL expressions are saved to a local file (see Step 1 of the workflow)[10].

Here we show the transformation of the `study_subject` class after Step 1 of the workflow has been applied. A direct excerpt from SIO (see Listing 8.1), shows the OWL definition of `study_subject` (aka `SIO_000399`).

```
<owl:Class rdf:about="&resource;SIO_000399">
 <rdfs:label xml:lang="en">study subject</rdfs:label>
 <rdfs:subClassOf
  rdf:resource="&resource;SIO_000498"/>
 <dc:description xml:lang="en">a study subject is an individual
  that is the subject of the study.</dc:description>
</owl:Class>
```

Listing 8.1: Example class defined in SIO.

Once applied, this definition is transformed; Listing 8.2 shows the non-patternised definition of `study_subject` in Tawny-OWL syntax. You will notice that the `study_subject` entity now refers to the readable `person` entity, instead of referring to `SIO_000498`.

```
(defclass study_subject
 :super person
 :annotation
 (annotation (iri "http://purl.org/dc/terms/description")
  (literal "a study subject is an individual that is the
subject of the study." :lang "en"))
 (label (literal "study subject" :lang "en")))
```

Listing 8.2: Example class defined in the non-patternised rendering of SIO.

The full effects of Step 1 on the ontology can be seen in Table 8.2. This application creates two files that (with the header file) spans 581KB. This is substantial decrease in size compared to the original OWL file of 896KB. If we consider the number of

[10] As discussed in Section 3.3, by introducing Clojure symbols, the ordering of entity declarations becomes important; an entity must be declared before use. While the transforming and rendering of terms is simplistic, the ordering of these rendered expressions with a definition of terms first is not possible in Tawny-SIO as some entities refer to themselves (e.g. `_label`). Thus we require a *predump* to declare all entities before any of their definitions. In a fully refactored version of Tawny-SIO, it would be possible to avoid most of these declarations.

lines, we see that the number of lines has been halved; the original had ∼20,000 lines while the non-patternised version has ∼10,000. This is most likely due to difference in syntax; in OWL/XML each construct contains a start and end tag, while Tawny-OWL does not.

If we take a closer look at the number of entities and axioms, we see that while we have the same number of entities, there are more axioms in the non-patternised ontology than the original ontology. In Tawny-OWL all disjoint axioms are rendered as pairwise disjoint axioms. This means that it takes three (`DisjointWith`) axioms, rather than one (`AllDisjointClasses`) axiom to declare that three classes are disjoint (see Listings 8.3 and 8.4). The current facilities in Tawny-OWL render one entity at a time, as opposed to one axiom at a time, making it difficult to remove this difference. This means that while the non-patternised ontology is semantically identical, it is not syntactically similar to the original ontology[11].

```
(as-disjoint A B C)
```

Listing 8.3: This expression expands to an `AllDisjoint-Classes` axiom.

```
(as-disjoint A B)
(as-disjoint A C)
(as-disjoint B C)
```

Listing 8.4: These expressions expand to three `DisjointWith` axioms.

Now that we have our non-patternised rendering of SIO in Tawny-OWL syntax, we can move onto applying patterns to SIO. In the next section, we show how the non-patternised `study_subject` class in Listing 8.2 is refactored in the patternised `study_subject` class.

[11] A brief investigation of how this effects the functional characteristics of the ontology, such as reasoning time, shows that quality is preferred over quantity, e.g. one `owl:AllDisjointClasses` axiom with three classes is quicker to reason than three `owl:DisjointWith` pairwise axioms. However it is unclear if this is because of reasoner functionality or communication between the ontology and reasoner.

8.3 Patternised refactoring of Tawny-SIO

In software engineering, patterns are used to avoid replication in code by abstraction. This principle holds for ontology engineering. The need for a pattern can be identified in two ways; visual inspection and/or by using tools [162], such as Regularities Inspector for Ontologies (RIO) [88]. For this work, all the patterns encoded were found by visual inspection. In this section, we briefly discuss some of the identified localised patterns of Tawny-SIO and their effect on the patternised refactoring of Tawny-SIO.

In Tawny-SIO, the majority of Tawny-SIO classes are defined with a name, label and description. An example Tawny-SIO class is shown in Listing 8.5. The SIO wiki documentation[12] supports that this is an intentional pattern:

> “ Resources **should** be described with brief English labels (`rdfs:label`) and human readable definitions (`dc:description`)...
>
> *Basic Design Principle No. 4, SIO wiki documentation* ”

```
(defclass study_subject
 :super person
 :annotation
 (annotation (iri "http://purl.org/dc/terms/description")
  (literal "a study subject is an individual that is the
   subject of the study." :lang "en"))
 (label (literal "study subject" :lang "en")))
```

Listing 8.5: Example class defined in SIO.

Explicit defining 1414 SIO classes is repetitive, time-consuming and susceptible to error. Therefore we want to encode a generic pattern, known as `defsclass`, and reuse this for each Tawny-SIO class. The desired usage of the `defsclass` pattern is shown in Listing 8.6.

```
(defsclass
 study_subject
 "a study subject is an individual that is the subject of
  the study."
 :super person)
```

Listing 8.6: Example usage of the `defsclass` pattern.

[12] `https://code.google.com/p/semanticscience/wiki/ODP`

We can encode the `defsclass` pattern as a two part pattern as shown in Listings 8.7 and 8.8. This type of functionality is a common theme throughout Tawny-OWL; there are functions that create the OWL API object (i.e. `class`) and a `def` equivalent that bind the resulting OWL API object to a local valid Clojure symbol (i.e. `defclass`). In this case we have the `sio-class` function that creates the relevant OWL API object while `defsclass` interns this value. The general functionality of this localised pattern is used to ensure that each SIO class:

- has a label annotation (automatically generated using symbol name)
- has a description annotation

The `defsclass` macro is created straight-forwardly from the equivalent function using the `defentity` macro from Tawny-OWL as shown in Listing 8.7.

```
(defentity defsclass sio-class)
```

Listing 8.7: An example implementation of the `defsclass` function.

We can encode the `sio-class` patterns as shown in Listing 8.8. The new Clojure parts of this definition are:

- The `make-label` function transforms the Clojure safe name into the desired label by replacing underscore characters with a whitespace character, then removing prefix whitespace characters.
- The `apply`[13] function is a `clojure.core` function. It applies a given function (the second element) to the args collection and any intervening arguments.

```
(defn sio-class [name description & frames]
 (apply class name
              :label (make-label name)
              :annotation (desc description)
              frames))
```

Listing 8.8: A common pattern for Tawny-SIO classes. **name**, **description** and **frames** are variables.

[13] `http://clojuredocs.org/clojure.core/apply`

The sio-class function makes use of a second pattern, namely the description annotation pattern, which creates a standardised annotation using the DC ontology description entity, as shown in Listing 8.9[14]. The desc function is another example of an annotation pattern (see Section 4.2).

```
(defn desc [description]
 (annotation dc-description
  (literal description :lang "en")))
```

Listing 8.9: The description pattern for Tawny-SIO classes.

In total, there are 1274 classes that use the defsclass pattern, leaving 23 exceptions which are defined directly using defclass. There are two reasons for these exceptions. 21 of these exceptions are because of the different legal characters between Clojure and IRI. The other two are the SIO classes, `product` and `target`, which do not contain a description annotation. This appears to be erroneous with SIO.

Although the defsclass pattern is a small and simple pattern, it has a substantial effect on Tawny-SIO. The original class definitions save to a file 496KB in size; this has decreased by over a half to 238KB after patternisation.

A similar pattern exists for the declaration of SIO object properties, named defsoproperty which utilises the sio-oproperty function. In total, there are 162 SIO object properties declared using the defsoproperty pattern. This means that there are 41 object properties that are declared using the regular defoproperty Tawny-OWL function; 37 of these do not have a description value, 3 are caused by differing legal character sets and 1 (`SIO_000288` with `rdfs:label` "is covalently connected to (transitive)") has both issues.

Similar to the defsclass macro, the application of the defoproperty pattern to the ontology has had a reductive effect on the size of the file. If we compare the full rendered Tawny-OWL forms versus the patternised forms we see a decrease from 82KB to 51KB when serialised.

Further exceptions to the defsclass pattern are the concepts that model the chemical elements, known as `atoms`. Unlike other Tawny-SIO classes, `atoms` do not have

[14]The `dc-description` predefined symbol refers to the `dc:description` object property via its IRI (`http://purl.org/dc/elements/1.1/description`). The `desc` is similar to the `label` function in Tawny-OWL, which uses an OWL built in property.

a description annotation; instead they have a `sio:see-also` annotation. This seems intuitive, otherwise the ontology would need to go into further detail (e.g. atomic number and element category). An example `atom` class is shown in Listing 8.10.

```
(defclass boron_atom
 :super atom
 :annotation
 (label (literal "boron atom" :lang "en"))
 (annotation seeAlso
  (literal "CHEBI:27560" :type :RDF_PLAIN_LITERAL)))
```

Listing 8.10: Example `atom` class.

There are 118 atoms defined in SIO. Explicitly defining all `atoms` is repetitive and can be avoided by utilising a localised pattern. The desired usage of the `defsatom` pattern can be seen in Listing 8.11.

```
(defsatom boron_atom "CHEBI:27560")
```

Listing 8.11: Example usage of the `defsatom` pattern.

Similar to other SIO entities, we define a pattern as a function, `sio-atom` as shown in Listing 8.12[15]. As this pattern is specialised for `atoms`, the superclass is "hard-coded" into the pattern. This localised pattern is used to ensure that each Tawny-SIO atom class:

- has a `atom` superclass restriction
- has a label annotation (automatically generated using the `make-label` function)
- has a see-also annotation (if provided)

```
(defn sio-atom [name chebi]
 (sio-atom-annotation-maybe
  (class name
         :super sio-atom-class
         :label (make-label name))
  chebi))
```

Listing 8.12: An example implementation of the `sio-atom` pattern.

[15]The `sio-atom-class` predefined symbol refers to the `sio:atom` class via its IRI (`http://ncl.ac.uk/sio/mysio#atom`)

Seven of the SIO atom subclasses lack a see-also annotation, for reasons that we describe later. In this case, we have built these exceptions into the pattern through `sio-atom-annotation-maybe` which simply applies a conditional (see Listing 8.13).

```
(defn sio-atom-annotation-maybe [cls chebi]
 (if-not (nil? chebi)
  (add-annotation cls (see-also chebi)))
 cls)
```

Listing 8.13: An example implementation of the `sio-atom-annotation-maybe` pattern.

This is used for example, in the definition of Copernicium (see Listing 8.14).

```
(defsatom copernicium_atom nil)
```

Listing 8.14: Example atom with no associated ChEBI identifier.

This application of the `defsatom` pattern to the ontology has a massive effect on the size of the file. The expanded class definitions spanned 22KB in size; this has decreased by over a half to 5KB.

Table 8.2: Table showing a comparison of the three versions of SIO.

Metric	`sio.owl`	Non-Patternised	Patternised
Size of file (bytes)	895,726	580,855	309,504
Number of lines	20654	10544	4854
Load time (msecs)	109.52	103.26	105.00
Number of entities	1626	1626	1626
Number of axioms	7463	7558	7460
Number of disjoints	75	170	72
Number of other axioms	5388	5388	5388

In total, we have identified and encoded 15 localised patterns (see Table 9.3). The full effects of Step 2 on the ontology can be seen in Table 8.5. This application utilises five files that in total would have a file size of 310KB. This is substantial decrease in size compared to the non-patternised ontology of 581KB. In terms of number of lines, the size has been halved; the non-patternised and patternised have ∼10,000 and ∼5,000 lines respectively. This is most likely due to the removal of the `:annotation` frame (when there were no further annotations other than the label and description annotations) and the expanded description definition for most of the SIO classes and object properties.

If we take a closer look at the number of entities and axioms, we see that while we have the same number of entities, there are less axioms in the patternised ontology than the original and non-patternised ontologies. This decrease occurs as the patternised ontology utilises `AllDisjointClasses` axioms where applicable. Thus the three `female`, `male`, `hermaphrodite` pair wise disjoints collate to one `AllDisjointClasses` axiom. Further, in the original ontology we find two `AllDisjointClasses` axioms declared for the subclasses of coordinate; in Listing 8.15 we see that the bottom axiom is an unnecessary `AllDisjointClasses` axiom as the top axiom semantically also declares this disjointness.

```
(as-disjoint _3D_cartesian_coordinate
             x_cartesian_coordinate
             y_cartesian_coordinate
             z_cartesian_coordinate)
(as-disjoint x_cartesian_coordinate
             y_cartesian_coordinate
             z_cartesian_coordinate)
```

Listing 8.15: Two `DisjointAllClasses` axioms found in SIO.

In this section we have shown that through the use of simple patterns we can ensure consistency. The encoding of these patterns forces us to deal with exceptions explicitly. We show that the use of patterns has significantly compressed the size of the Tawny-OWL file compared to the non-patternised Tawny-OWL syntax file. In the next section we will summarise further identified inconsistencies.

8.4 Tawny-SIO errors

During the application of this workflow we found inconsistencies in SIO that we have interpreted as errors. For example, in Section 8.2 we found that two entities had the same `rdfs:label` annotation. This error was identified through the use of a label to Clojure name transformation pattern. In Section 8.3 we found that numerous entities were missing the necessary `dc:description` annotation. These missing descriptions were also found with the use of patterns. In this section, we briefly discuss some of the other errors of Tawny-SIO.

In this thesis we have shown that with the use of patterns we can ensure the consistent modelling of entities. When there is a lack of consistency in the original ontology, we start to question the integrity of the ontology and/or external resources.

In SIO version `0.9.22`[16] we found that elements 112 to 118 lack a `sio:see-also` annotation [163]. This meant that, according to SIO, these elements do not exist in the ChEBI database [52]. This is generally true, however there was one exception; element 112 (aka Copernicium aka Ununubium) can be found with ChEBI identifier `CHEBI:33517` [34]. This discovery was identified to the ontology authors who have since corrected and published this information since SIO version `0.9.22`[17].

However as introduced earlier this is not the only error; while refactoring, we found incorrect ChEBI annotations and missing or incorrect PATO identifiers. The complete list of errors is shown in Table 8.3. These were identified through the use of patterns, grouping[18] and/or visualisation. Using these methods we found 43, 4, and 3 errors respectively. A list of correct identifiers for entities with missing or incorrect annotations is shown in Table 8.4.

In this section we have shown that during the patternisation process we have found numerous inconsistencies; the majority of which were identified through use of patterns. This occurs as the encoding of patterns, forces us to have to deal with exceptions explicitly. In the next section, we will show how patterns can also aid

[16] Published in July 2013.
[17] Published in October 2013.
[18] During Step 2, similar entities were grouped by type (e.g. class or object property) and logical definition (e.g. parent).

Table 8.3: Table showing the list of potential errors found in SIO. The identification method is categorised as either **P**atterns, **G**rouping, or **V**isualisation.

Error type	Method	Effected entities
Duplicate label annotations	P	`interval`, `set_interval`
Incorrect ChEBI identifier	V	`chemical_entity`, `molecule`
Incorrect PATO identifier	V	`hermaphrodite`
Missing PATO identifier	G	`female`, `bent`, `curved`, `abnormal`
Missing ChEBI identifier	P	`copernicium_atom`
Missing description annotation	P	`product`, `target`, `is_transitively_related_to`, `has_identifier`, `is_identifier_for`, `is_satisfied_by`, `has_realizable_property`, `is_realizable_property_of`, `is_broader_than`, `is_broad_match_to`, `is_exact_match_to`, `is_close_match_to`, `has_expression`, `is_manifestation_of`, `is_subject_of`, `is_evidence_for`, `is_disputing_evidence_for`, `is_supporting_evidence_for`, `is_refuting_evidence_for`, `is_denoted_by`, `is_modelled_by`, `affects`, `is_affected_by`, `is_realized_in`, `is_regulated_by`, `is_preceded_by`, `results_in`, `is_result_of`, `is_trigger_for`, `is_transcribed_from`, `is_translated_from`, `is_translated_into`, `is_time_boundary_of`, `has_time_boundary`, `has_start_time`, `is_start_time_of`, `has_end_time`, `is_end_time_of`, `is_covalently_connected_to__transitive_`, `is_weakly_interacting_with`

Table 8.4: Table showing the list of correct identifiers for each mislabelled entity identified in Table 8.3.

Entity	Identifier
abnormal	PATO:0000460
bent	PATO:0000617
chemical_entity	CHEBI:24431
copernicium_atom	CHEBI:33517
curved	PATO:0001591
female	PATO:0000383
hermaphrodite	PATO:0001340
molecule	CHEBI:25367

downstream users of the ontology.

8.5 Patterns for downstream usage

The SIO wiki documents many exemplar ODPs that can be used in conjunction with SIO. Although they are not explicitly described as such, in our terminology, this means that authors of SIO have already identified the potential downstream localised patterns. These patterns are described by exemplars written in "The Pretty Turtle Syntax (TPS)"[19]; so there is no direct computational representation of these as a pattern, since the syntax does not support variables. As with the abnormality patterns in Tawny-Karyotype, with Tawny-OWL we can encode these SIO ODPs for downstream usage. In this section we investigate the value of this explicit representation.

One example ODP documented by the authors of SIO is the `biochemical-reaction` pattern. An example expansion of this pattern refactored into Tawny-OWL syntax is shown Listing 8.16[20]. If we build this pattern based on the original documentation, the pattern does not work, due to inconsistencies occurring between the ontology and the documentation. We found that the `target_role` entity should be `reactant_role` [19]. Their documentation (i.e. wiki) has since been updated to show this.

```
(defclass hexokinase_reaction
 :equivalent
 (and
  biochemical_reaction
  (some realizes
   (and catalytic_role (some is_role_of hexokinase)))
  (some realizes
   (and product_role (some is_role_of ADP)))
  (some realizes
   (and product_role (some is_role_of glucose-6-phosphate)))
  (some realizes
   (and target_role (some is_role_of ATP)))
  (some realizes
   (and target_role (some is_role_of glucose)))))
```

Listing 8.16: Part of an instance of the `biochemical-reaction` pattern, based on the original SIO documentation.

[19] `https://code.google.com/p/semanticscience/wiki/PrettyTurtleSyntax`.

[20] This expansion was based on the TPS example seen at `https://code.google.com/p/semanticscience/wiki/ODPBiochemistry` on December 2013 [163].

Another example ODP documented by the authors of SIO is the biochemical-pathway pattern. An example encoding of this pattern is two-fold, as shown in Listings 8.17 and 8.18. The biochemical-pathway function creates the composite class definition while the biochemical-pathway0 recursive function creates the complex nested axiom required by the biochemical-pathway pattern. Here, the pathway class, the has_proper_part and precedes object properties are "hard-coded" into the pattern. Similar to the earlier implementations, the biochemical-pathway function is paired with a macro, named defbpathway, which binds the resulting OWL API object to a local variable.

```
(defn biochemical-pathway [name & reactions]
 (class name
 :equivalent
 (and pathway
      (some has_proper_part
            (biochemical-pathway0 reactions))
      (some has_proper_part reactions))))
```

Listing 8.17: Example encoding of the biochemical-pathway function.

```
(defn biochemical-pathway0 [reactions]
 (if (= 1 (count reactions))
   (first reactions)
   (and
    (first reactions)
    (some precedes
          (biochemical-pathway0 (rest reactions))))))
```

Listing 8.18: Example encoding of the biochemical-pathway0 recursive function.

In Listing 8.19, we show an example usage of the defbpathway pattern. Here we describe the glycolysis pathway in OWL. This exemplar expands to one entity and two axioms; one declaration and one equivalent logical axiom, as shown in Listing 8.20.

```
(defbpathway glycolysis_pathway
 hexokinase_reaction
 phosphoglucose_isomerase_reaction
 phosphofructokinase_reaction
 fructose-bisphosphate_aldolase_reaction
 triosephosphate_isomerase_reaction
 glyceraldehyde_phosphate_dehydrogenase_reaction
 phosphoglycerate_kinase_reaction
 phosphoglycerate_mutase_reaction
 enolase_reaction
 pyruvate_kinase_reaction)
```

Listing 8.19: Example usage of the `defbpathway` function.

Here, we show that with the use of patterns we can explicitly define entities in a concise and simple way. Instead of requiring 26 lines of explicit code, we only require 11. This concise syntactic expressions mean that we save space. Also we do not have to consider the complicated axiomitisations as the `biochemical-pathway` pattern automatically constructs these for the user.

```
(defclass
 glycolsis_pathway
 :equivalent
 (and
  pathway
  (some has_proper_part hexokinase_reaction)
  (some has_proper_part phosphoglucose_isomerase_reaction)
  (some has_proper_part phosphofructokinase_reaction)
  (some has_proper_part fructose-bisphosphate_aldolase_reaction)
  (some has_proper_part triosephosphate_isomerase_reaction)
  (some has_proper_part glyceraldehyde_phosphate_dehydrogenase_reaction)
  (some has_proper_part phosphoglycerate_kinase_reaction)
  (some has_proper_part phosphoglycerate_mutase_reaction)
  (some has_proper_part enolase_reaction)
  (some has_proper_part pyruvate_kinase_reaction)
  (some has_proper_part
   (and hexokinase_reaction
    (some precedes
     (and phosphoglucose_isomerase_reaction
      (some precedes
       (and phosphofructokinase_reaction
        (some precedes
         (and fructose-bisphosphate_aldolase_reaction
          (some precedes
           (and triosephosphate_isomerase_reaction
            (some precedes
             (and glyceraldehyde_phosphate_dehydrogenase_reaction
              (some precedes
               (and phosphoglycerate_kinase_reaction
                (some precedes
                 (and phosphoglycerate_mutase_reaction
                  (some precedes
                   (and enolase_reaction
                    (some precedes
                     pyruvate_kinase_reaction))))))))))))))))))))
```

Listing 8.20: Expansion of the glycolysis biological pathway.

From the SIO wiki pages, we have identified and encoded 15 localised patterns (see

Table 9.3) and 14 simple example usages of the patterns (Step 3 of the workflow). These exemplars encode 33 entities and 76 axioms.

The implementation of downstream patterns and exemplars results in a file of size 16KB; the full expansion of these exemplars creates a file of size 21KB. In terms of lines of code, the non-patternised and patternised files are 220 and 508 lines of Tawny-OWL syntax code respectively. In this instance we have decreased the size of the file but have increased the number of lines. This is probably due to the amount of whitespace occurring in the patternised file.

Table 8.5: Table showing a comparison of the non-patternised and patternised versions of the downstream (exemplars) ontology.

Metric	Non-Patternised	Patternised
Size of file (bytes)	20,729	16,105
Number of line	220	508

In summary, we show that patterns are of benefit to quality control of the documentation and consistency between the ontology and documentation, as we showed when encoding the `biochemical-reaction` pattern. They are also useful for the concise definition of complex entities such as the `glycolysis` biological pathway.

8.6 Summary

In this chapter, we show further benefits of applying a programmatic and pattern-driven approach of ontology engineering by refactoring an existing bio-ontology; SIO (RQ3).

SIO was created independent of Tawny-OWL and without an explicit encoding of patterns. Despite this, through careful refactoring, we have shown that it is possible to refactor SIO into Tawny-SIO and that doing so enables us to decrease the overall size while aiding consistency. This conforms with prior work looking at patternisation of the anatomy ontology which also found the presence of significant numbers of patterns [89].

We found that localised patterns were as useful within Tawny-SIO as they were in The Karyotype Ontology and The Mitochondrial Disease Ontology. A number of patterns were identified and their use helps increase the consistency and concision of the ontology. Furthermore, localised patterns, whilst not themselves used in SIO, are potentially useful for downstream users. However there are potential limitations; with our current encoding of the `biochemical-pathway` pattern, inferences of circular pathways (e.g. the TriCarboxylic Acid cycle (TCA) pathway) might require further thought.

Lastly, we show that through the application of patterns to an existing ontology, we can potentially identify any existing ontology errors. In SIO, the majority of the identified errors were found as we had to explicitly handle exceptions to our preconceived patterns. Some of these errors have been sent to the authors of SIO, and have since been updated.

In the next chapter, we discuss the classification and provide basic statistics of our localised patterns in order to improve the community's understanding of localised patterns.

9

Pattern classification

Contents

9.1 Introduction 172
9.2 Classification by role 174
9.2.1 Internal localised patterns 174
9.2.2 External localised patterns 175
9.3 Results 176
9.4 Summary 182

9.1 Introduction

There are a number of existing ODP classifications. Generally, these have classified generic ODPs. However, during the ontology development described in this thesis, whilst we have used some of these, the majority of the patterns that we have used are specific to a single ontology, i.e. they are *localised patterns* (see Section 4.4). To our knowledge there has been no previous classification of this form of ODP, therefore in this chapter, we analyse our use of localised patterns and introduce a preliminary classification of this form of pattern.

Two well-known examples of ODP classifications are found in [35] and [119]. In [35], ODPs are classified according to their usage; more specifically as Extension, Good Practice or Domain Modelling ODPs. Extension ODPs are defined as ODPs that enable the ontologies to overcome their limits; an example of this is the nary relationship ODP [103] that enables the community to link an individual to more than one individual or value. Good Practice ODPs, as the name suggests, are ODPs used for good practice, such as the value partition ODP [121]. Lastly, Domain modelling ODPs are used to model a concrete part of the biological domain, e.g. sequence ODP [31]. For further discussion on each generic ODP, see Section 4.2.

In [119], ODPs can be classified into nine categories (see Figure 9.1); definitions for each are provided in Table D.1. Of these, we find that the majority of our localised patterns are classified as Content Patterns (CPs) (aka content design patterns) i.e. design patterns that are built to solve recurring domain modelling problems [28].

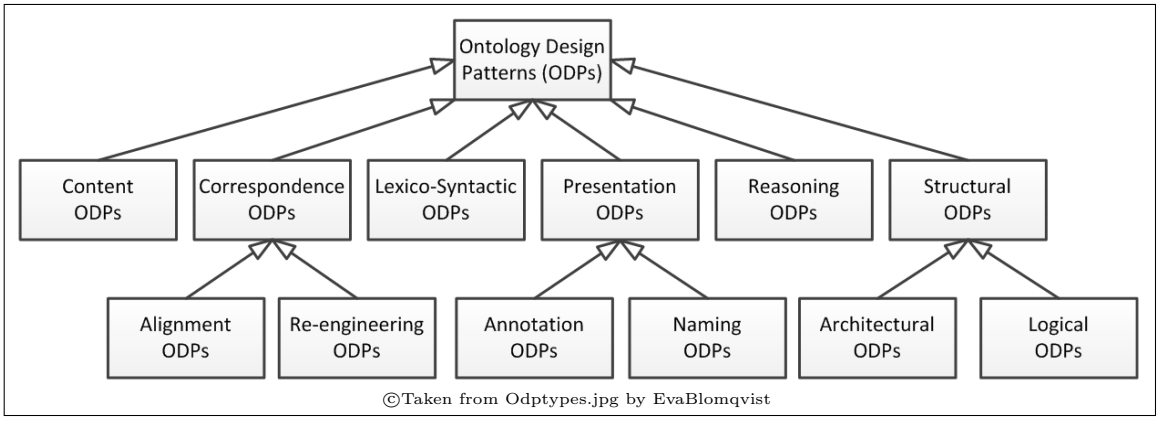


Figure 9.1: Simplified classification of ODPs.

However, as described in Section 9.3, we find that these two classifications are not descriptive, as the bulk of the localised patterns (from our three ontologies: The Karyotype Ontology, The Mitochondrial Disease Ontology and Tawny-SIO) are grouped into one category in each i.e. CPs or domain modelling ODPs for [119] and [35] respectively. Therefore, in this chapter we introduce a new simple classification of localised patterns.

9.2 Classification by role

Our first criteria for classification of localised patterns is by "role": they can be further classified as an *internal localised pattern* and/or an *external localised pattern* (see Figure 9.2). We define *internal localised patterns* as "localised patterns that aide ontology construction" and *external localised patterns* as "localised patterns that aide ontology downstream usage". A visualisation of this classification is shown in Figure 9.2.

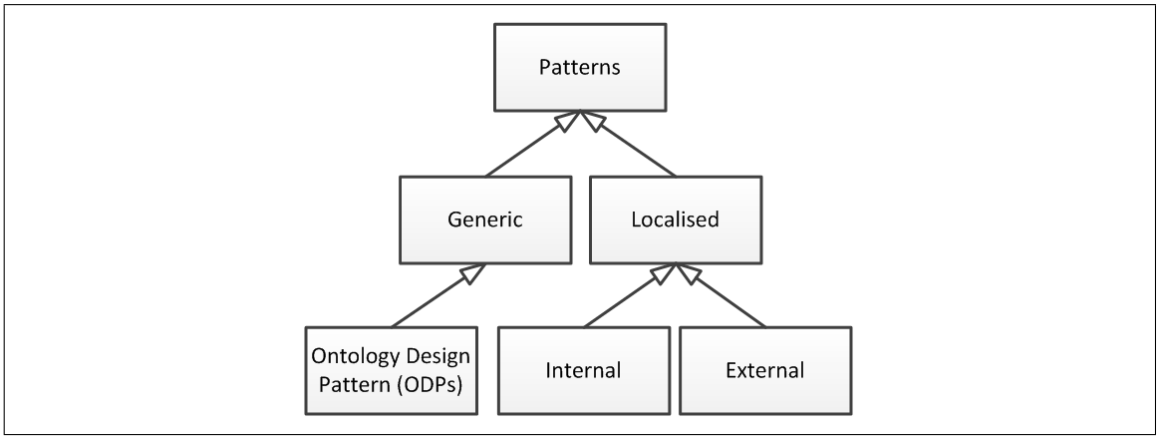


Figure 9.2: Classification of patterns.

This classification of localised patterns is not disjoint; sometimes a localised pattern can be both internal and external. In the recasting of The Pizza Ontology, the `generate-named-pizza` pattern is used to populate The Pizza Ontology and defines a finite set of `NamedPizza`s. In reality however, the `generate-named-pizza` pattern could be used by a variety of pizza chains to populate The Pizza Ontology to store information about their `NamedPizza`s. Thus, the `generate-named-pizza` pattern is classified as both an internal and external localised pattern.

In this section, we briefly discuss some of the internal and external localised patterns that exist in The Karyotype Ontology.

9.2.1 Internal localised patterns

Like The Pizza Ontology, we make use of localised patterns in The Karyotype Ontology. In fact, most of the entities in The Karyotype Ontology are a part of at least one pattern, making The Karyotype Ontology our most highly patternised ontology.

Of 1616 entities, 1248 entities are created using a single pattern, the humanbands pattern. As the name suggests, the humanbands pattern models the chromosome bands; using a simple tree-like structure, the humanbands pattern expands the input data and generates the necessary classes and restrictions. The humanbands pattern is classified as an internal localised pattern as its role is to define a finite number of human chromosome bands. As this pattern is unlikely to change, this pattern is parameterised directly in the source code.

Another internal localised pattern is the resolution pattern, which refines chromosome bands to include cytogenetic resolution information. Similar to the named-generate-pizza pattern, the resolution pattern uses the generic closure pattern. This pattern is also classified as an internal localised pattern as resolution refines a finite set of human chromosome bands. Unlike the humanbands pattern the data source of this pattern is a local resource text file.

9.2.2 External localised patterns

A number of patterns are defined within The Karyotype Ontology which were not used during the construction of The Karyotype Ontology; rather they are intended for use by end-users of the ontology (and also by the test cases and exemplars for The Karyotype Ontology). Within The Karyotype Ontology, we define an abnormality as a concept and support each with a usage pattern and a function(s) that generates an OWL restriction according to the pattern. One simple example of an abnormality pattern is the inversion localised pattern, which abstracts over the axioms that define an inversion event in a karyotype.

In The Karyotype Ontology, we define a karyotypic inversion restriction with an exact cardinality restriction. The purpose of the pattern is to abstract from the exact axiomatisation and require parameterisation with experimental knowledge alone: i.e. the number of chromosomes and the two breakpoint bands that are affected.

9.3 Results

In this section we apply three classifications to the localised patterns used in the engineering of our ontologies (with the exception of The Pizza Ontology). In summary, the three classifications[1] are:

VP In [119], ODPs are sorted into nine categories, see Figure 9.1 and Table D.1 for descriptions of each.

ME In [35], ODPs are classified according to the way they are used, see Table D.2 for descriptions of each.

JW Patterns are classified according to their role in the ontology, i.e. internal localised patterns (see Section 9.2.1) and external localised patterns (see Section 9.2.2).

The results of this classification can be seen in Tables 9.1, 9.2 and 9.3.

Firstly, with **VP**, we find that most of the ontology localised patterns are grouped in the CP category, as they are specific to the domain. Though it is of interest to note that in Tawny-SIO, half of the localised patterns are (also) Annotation Patterns. With **ME**, we find that the identified localised patterns are sorted as either Good Practice or Domain Modelling ODPs. For both of these classifications, most of the patterns have been classified into a single category as these classifications have focused on generic patterns. While our work instead focuses on the patterns that are specific to the ontology, we suggest that the heavy use of patterns in our ontologies is as a result of technology; with Tawny-OWL it is now easy to add a new pattern on a per ontology basis.

Lastly, our simple classification (based on role) shows that the majority of the patterns in Tawny-Karyotype and Tawny-Mitochondria[2] are external localised patterns and internal localised patterns respectively, whilst Tawny-SIO has a balance of both internal localised patterns and external localised patterns.

[1] We refer to each classification by the initials of the first author.

[2] Tawny-Mitochondria refers to the Tawny-OWL code used to build The Mitochondrial Disease Ontology.

Table 9.1: Table highlighting pattern classification statistics of localised patterns used in the Tawny-Karyotype and Tawny-Karyotype-Scaling projects. **VP** and **ME** are defined as the category codes identified in Table D.1 and Table D.2 respectively, while **JW** is classified as I(nternal) and/or E(xternal). For each localised pattern we show: the number of times we use the localised pattern in Tawny-Karyotype (i.e. *usage*); the number of *calls* (internal localised pattern only); the number of logical *axioms* generated; and the logical axiom *coverage %* shown here as C% (internal localised pattern only). For each external localised pattern that has a variadic argument, we show the minimum number of axioms that could be generated; for example `affects-band` [1] generates at least 2 axioms for each call.

Name	**VP**	**ME**	**JW**	Usage	Calls	Axioms	C%
`resolution`	Con	D	I	1	1248	7395	71.4
`human-sub-band`	Con	D	I	2	845	1690	16.3
`humanbands0`	Con	D	I	2	285	195	1.9
`humanbands`	Con	D	I	48	48	304	2.9
`set-ordinal`	Con	D	I/E	1	25	25	0.2
`addition-band`	Con	D	E	1	–	0	–
`addition-chromosome`	Con	D	E	1	–	0	–
`affects-band` [1]	Con	D	E	1	–	2+	–
`affects-band` [2]	Con	D	E	1	–	1+	–
`affects-band` [3]	Con	D	E	2	–	1+	–
`deletion-band`	Con	D	E	2	–	0	–
`deletion-chromosome`	Con	D	E	1	–	0	–
`derivative`	Con	D	E	56	–	1	–
`dicentric`	Con	D	E	6	–	1	–
`duplication-pattern`	Con	D	E	3	–	0	–
`exactly-direct-event`	Con	D	E	1	–	0	–
`exactly-direct-feature`	Con	D	E	1	–	0	–
`exactly-event`	Con	D	E	1	–	0	–
`exactly-feature`	Con	D	E	1	–	0	–
`fission`	Con	D	E	2	–	1/2	–
`fragilesite`	Con	D	E	4	–	1	–

Continued on next page...

Table 9.1 – continued from previous page

Name	**VP**	**ME**	**JW**	Usage	Calls	Axioms	C%
hsr	Con	D	E	7	–	1	–
insertion-pattern	Con	D	E	6	–	1	–
inversion-pattern	Con	D	E	1	–	0	–
isochromosome	Con	D	E	6	–	1	–
isoderivative	Con	D	E	3	–	1	–
isodicentric	Con	D	E	1	–	1	–
karyotype-class	Con	D	E	1	–	2+	–
marker	Con	D	E	5	–	1	–
pseudo-dicentric	Con	D	E	2	–	1	–
pseudo-isodicentric	Con	D	E	0	–	1	–
quadruplication	Con	D	E	2	–	1	–
ring	Con	D	E	9	–	1	–
robertsonian	Con	D	E	4	–	1	–
some-direct-event	Con	D	E	1	–	0	–
some-event	Con	D	E	1	–	0	–
some-feature	Con	D	E	1	–	0	–
some-direct-feature	Con	D	E	1	–	0	–
translocation	Con	D	E	78	–	1	–
tricentric	Con	D	E	4	–	1	–
triplication-pattern	Con	D	E	3	–	1	

Table 9.2: Table highlighting pattern classification statistics of localised patterns used in the Tawny-Mitochondria project. For further details see Table 9.1.

Name	**VP**	**ME**	**JW**	Usage	Calls	Axioms	C%
existing-class	Con	D	I	2	2174	4348	57.7
source	Con	D	I	1	2174	0	0.0
term-class	Log	G	I	1	2121	0	0.0
protein-class	Con	D	I	1	479	479	6.4
gene-class	Con	D	I	1	461	461	6.1
hanatomy-class	Con	D	I	1	163	163	2.2
author-fact	Con	D	I	1	87	0	0.0
disease-class	Con	D	I	1	41	41	0.5
paper-class	Con	D	I	1	30	30	0.4
pmid-fact	Con	D	I	1	30	0	0.0
title-fact	Con	D	I	1	30	0	0.0
manatomy-class	Con	D	I	1	15	15	0.2
dna-mutation-class	Con	D	I	1	6	6	0.1
protein-mutation-class	Con	D	I	1	2	2	0.0

Table 9.3: Table highlighting pattern classification statistics of localised patterns used in the Tawny-SIO project. For further details see Table 9.1.

Name	**VP**	**ME**	**JW**	Usage	Calls	Axioms	C%
desc	Ann	G	I	27	1461	0	0.0
sio-class	Con	D	I	1274	1274	1772	76.1
sio-oproperty	Con	D	I	162	162	436	18.7
sio-atom	Con	D	I	118	118	118	5.1
sio-atom-annotation-maybe	Ann	G	I	1	118	0	0.0
see-also-rdf	Ann	G	I	6	117	0	0.0
example	Ann	G	I	1	68	0	0.0
synonym-en	Ann	G	I	46	46	0	0.0
synonym-rdf	Ann	G	I	43	43	0	0.0
equivalent-rdf	Ann	G	I	36	36	0	0.0
subset-rdf	Ann	G	I	18	18	0	0.0
equivalent-uri	Ann	G	I	13	13	0	0.0
see-also-uri	Ann	G	I	6	6	0	0.0
similar-uri	Ann	G	I	2	2	0	0.0
sadi	Ann	G	I	1	2	0	0.0
biochemical-pathway	Con	D	E	1	–	1	–
biochemical-pathway0	Con	D	E	3	–	0	–
biochemical-reaction	Con	D	E	1	–	1	–
enzyme-dissociate	Con	D	E	2	–	1	–
enzyme-part-of	Con	D	E	2	–	1	–
enzyme-role	Con	D	E	2	–	1	–
molecule-atom	Con	D	E	4	–	1	–
molecule-pattern	Con	D	E	1	–	3+	–
parameter-pattern	Con	D	E	1	–	1	–
protein-containment	Con	D	E	1	–	1	–
protein-pattern	Con	D	E	1	–	2+	–

Continued on next page...

Table 9.3 – continued from previous page

Name	**VP**	**ME**	**JW**	Usage	Calls	Axioms	C%
protein-residue	Con	D	E	4	–	2	–
reaction-pattern	Con	D	E	1	–	1+	–
software	Con	D	E	2	–	1	–
some-role	Con	D	E	2	–	1+	–

9.4 Summary

In this thesis we have identified numerous patterns; this is especially true for The Karyotype Ontology. We accept that The Karyotype Ontology is, no doubt, an extreme example, but we also found the same thing in The Pizza Ontology, The Mitochondrial Disease Ontology and Tawny-SIO. Here, we show that the use of these patterns was beneficial in both the construction of the ontology and the downstream usage of the ontologies (RQ3).

In addition, we use explicit metrics to show how highly patternised our ontologies actually are. Specifically 73.3%, 97.7% and 99.9% of the logical axioms are generated by localised patterns in The Mitochondrial Disease Ontology, The Karyotype Ontology and Tawny-SIO respectively (see Tables 9.2, 9.1 and 9.3). Furthermore, each ontology has at least one pattern that generates at least half the required logical axioms.

Many of the localised patterns are very small, but even small patterns help with regularity, and, more importantly, help with developer comprehensibility. Our most extreme example comes from outside of this work; the Open Biomedical Ontologies (OBO) ID pattern is designed to hide IDs from the developer. We have also used patterns for maintainability and for "late binding" so we can change our mind about axiomitisations.

Lastly, we show that two existing ODP classifications are insufficient in the classification of localised patterns. With our novel classification (that classifies localised patterns based on their role) and basic statistics of the localised patterns used in this work, we aim to improve the ontological community's understanding of localised patterns. Thus this chapter concludes our research into how a pattern-driven and programmatic approach can be used to build a computational representation of the ISCN (RQ1).

In the next chapter, we will summarise the findings of this thesis and discuss how the work discussed in this thesis can improve the ontology engineering process.

10

Discussion

Contents

10.1 Introduction 184
10.2 Utilising a pattern-driven and programmatic approach . . . 185
10.3 The Karyotype Ontology 188
10.4 The Mitochondrial Disease Ontology 191
10.5 Re-purposing software engineering 193
10.6 Improving the ontology engineering process 197

10.1 Introduction

Since the success of GO, the awareness and usage of ontologies in biology has grown. The engineering of these ontologies is well-studied and there are a variety of methodologies and techniques, some of which have been re-purposed from software engineering methodologies and techniques. However, due to the complex nature of bio-ontologies, they are not resistant to errors and mistakes. This is especially true for more expressive and/or larger ontologies.

In order to overcome these issue, we explored the usage of a pattern and programmatic approach to ontology engineering. In this chapter, we reflect on the usage of this approach in the development of two novel models of biology and the recasting of one existing bio-ontology. Further, we discuss how the approach can also be used for the downstream usage of ontologies (see Section 10.2).

As a result of this work, two computational artefacts were produced for two domains of biology; The Karyotype Ontology and The Mitochondrial Disease Ontology. The Karyotype Ontology provides a formal representation of karyotypes in a form that is easy to query, validate and maintain, while The Mitochondrial Disease Ontology is a reference ontology that describes the terms associated with mitochondrial disease. In this chapter, we discuss the limitations and identify possible improvements and extensions that could be applied to each of these ontologies (see Sections 10.3 and 10.4 respectively).

In addition, while modelling these two domains, we were able to re-purpose some software engineering techniques, such as versioning and Continuous Integration (CI), to aid ontology engineering. While we have not provided the specifics of this data in this thesis, we briefly reflect on the success (or failure) with a collection of re-purposed techniques (see Section 10.5).

In summary, this work has taken a step towards building ontologies using a software engineering perspective, thus removing ourselves from having to utilise bespoke tooling. This chapter ends with a review of this work, specifically highlighting improvements that could help enhance the (computing science) ontology community (see Section 10.6).

10.2 Utilising a pattern-driven and programmatic approach

During the construction of The Karyotype Ontology, we found that many of the entities were very similar in logical structure; this forced us to use a pattern-driven approach. We tried to express these patterns in various ways; the best way being OPPL, which allows us to change or update an existing ontology using a declarative query language. However, there are difficulties with using OPPL in a pattern-driven approach. For instance, we introduce a completely new syntax over and above that of OWL. This necessitates a new environment such as the OPPL plugin for Protégé. However, this is a one-shot tool; OPPL is applied to the ontology which is then changed. But what if we find our OPPL transformation was wrong, or we wish to reapply with new knowledge? And, where do we store and version our OPPL scripts?

We also considered generating the Manchester Syntax using a script or using the OWL API, but believed neither of these approaches to be workable in practice. The overall structure of the ontology would be hidden in the former case, and the semantics of Java, with its use of classes and method invocation bears little relationship to the semantics of OWL. We used Tawny-OWL to implement these patterns; the advantage of a single syntax and environment for both patternised and non-patternised parts was overwhelmingly clear. That we could also use the same environment and syntax for scalability, performance testing and parsing was unexpected but also extremely useful.

Tawny-OWL currently provides support for three well-known existing patterns: value partition, closure and covering axioms. These patterns have been used in Tawny-Pizza (see Chapter 4) and are available for use in other ontologies. As Tawny-OWL is fully programmatic, we can easily encode patterns that are localised to the scope of a single ontology. These localised patterns are useful as they enable syntactic concision, ensure consistency and support maintainability should the need for change arise. This form of localised pattern stands in contrast to most work on ODPs which have focused on generic patterns.

The Karyotype Ontology is predominantly made up of a variety of localised patterns (see Table 9.1). Around half of these entities, specifically the human chromosome bands, are generated via the `humanbands` pattern. In practice, the use of a pattern is the only plausible way we could model these repetitive entities. In modelling human chromosome bands there is no shortcut we could implement; we have to list all the possible bands. While The Karyotype Ontology is possibly exceptional in this regard, it is by no means unique. Other potential repetitive domains that this type of pattern-driven approach would be applicable to are: countries[1], stellar classification [92], human languages[2], and chess pieces [63].

Through the use of patterns, we show that we can easily extend The Karyotype Ontology (see Chapter 6). For example, karyotypic events, such as deletions and insertions that affect a sequence of bands, are not modelled in our ontology. We have discussed three ways we could achieve this. *A priori*, it is difficult to determine which of these will work best (for various sizes), particularly with respect to non-functional characteristics such as reasoning time. The extensibility of Tawny-OWL enables us to test this by generating multiple test versions of the ontology. With 10^5 classes this would otherwise be impractical.

To our knowledge, this (programmatic) application of patterns is completely novel. There is relatively little work on addressing this form of non-functional characteristic (reasoning time), largely because once the ontology has been built, it is too late to test.

We have demonstrated that patterns benefited the development of Tawny-Pizza, The Karyotype Ontology and The Mitochondrial Disease Ontology. However, we wish to understand whether patterns are generally useful. Therefore, we have also applied this methodology to SIO (see Chapter 8), which we chose after the construction of Tawny-OWL, and which was built without knowledge of Tawny-OWL.

We find that patterns are generally less useful within Tawny-SIO than The Karyotype Ontology. However, a number of patterns were identified that could increase the consistency and concision of this ontology. The lack of patterns for construction

[1] `http://www.iso.org/iso/country_codes`
[2] `http://www.iso.org/iso/language_codes`

is probably because SIO is an upper level ontology that does not have much self-repetition. The only exception is in the definition of atoms; here a pattern is vital and bears more resemblance to a middle level ontology. However even with SIO we have identified numerous annotation patterns that are more or less independent of the domain. For example, an annotation that utilises SIO term(s) is obviously not independent of the domain while an annotation that states the authors is independent. Furthermore, we (and SIO authors) have identified and implemented additional patterns, which whilst are not themselves used in Tawny-SIO, are potentially useful for downstream users of Tawny-SIO.

In this thesis, we have described the application of Tawny-OWL to four ontologies which have allowed us to test the usage of patterns. It is clear that the utility of the pattern-driven approach will depend on the nature of the ontology. For example, a structurally simple ontology may use few patterns. However, we note that they are not limited to the logical component of OWL; within Tawny-SIO we have used a number of annotation patterns. We believe that once patterns are easy to use and integrate within the ontology environment, patterns will cease to become exceptional and start to become a routine utility for authors and downstream users of ontologies. They can be used in many different parts of the life cycle and appear to be a promising approach that can substantially improve ontology engineering.

10.3 The Karyotype Ontology

From around 1960, karyotypes have been described as semantically meaningful strings[3], as defined by the standard nomenclature, ISCN. Historically, this representation has worked for human-to-human (written) communication; however the number of ISCN Strings is continuously growing, thus outstripping our human capabilities. As a result, we question the representation, scalability and extensibility of ISCN.

The specification mixes epistemology with ontology. For example ISCN Strings generally represent the canonicalised karyotype, however when describing mosaic karyotypes we count the number of clones. This confusion is progressively increases when incorporating FISH and sequence knowledge. This is not necessarily a problem, but it does mean that the ISCN is going to get increasingly more complex over time as more techniques are invented. This increase of scope means that the necessity for a machine interpretable version becomes more important as the current format, a standalone book, is not extensible. Any existing data, or other specifications related to techniques such as FISH, cannot be simply integrated with the ISCN. The specification is a standalone book, so the only way to compose it with other specifications is to rewrite it.

What we need is a representation that is computationally amenable, scalable, extensible and can integrate with other specifications. The Karyotype Ontology is all of these things; it has formally defined semantics and specification; something which we have used during this thesis. We have shown that it is scalable to realistic numbers of ISCN Strings. The use of Tawny-OWL adds a layer of extensibility over and above that already provided by OWL. Finally, as part of the Semantic Web, OWL has been built to integrate with other resources available on the open web; the semantics of OWL mean that this is truer of The Karyotype Ontology than would have been the case with either a relational or eXtensible Markup Language (XML) based representation. While we have not explored this aspect of OWL as part of the thesis, integrating The Karyotype Ontology with resources such as GO or The

[3]known in this thesis as ISCN Strings

Sequence Ontology (SO) [37] is obvious future work for The Karyotype Ontology.

Unfortunately, one key resource that The Karyotype Ontology does not cleanly integrate with is the ISCN specification, or any existing ISCN Strings. Lacking a formal computational specification for ISCN makes it difficult to extract knowledge stored in ISCN Strings and representing it using The Karyotype Ontology; this is a process which adds structure to the knowledge and thus remains difficult to automate. It is, however, possible to apply heuristics to parse the ISCN Strings and represent them in The Karyotype Ontology. One unexpectedly useful implication of our use of Tawny-OWL, is that we can integrate code implementing these heuristics in Clojure with The Karyotype Ontology. Inevitably, the heuristic nature of this process will be error-prone; even here, though, the extensible and integratable nature of The Karyotype Ontology helps. We can, for instance, use The PROV Ontology (PROV-O) to describe and maintain links back to the original source, enabling any updates to be percolated cleanly, something which has been difficult to do in other resources [14].

In this thesis, we have experimented heavily with a fully programmatic environment and the impact it has on ontology development. It is possible that this work can also enable the reverse: integrating ontologies into software development. If possible this would greatly help with one further extension that is critically required by the cytogeneticists; they require the ability to visualise these ISCN Strings as ideograms.

One other extension of The Karyotype Ontology we are interested in and would like to develop further, is the use of reasoning to identify and classify syndromes. For example Cri du Chat Syndrome (CdCS) [66] is a genetic disease that is caused by a partial deletion (of variable size) occurring in the short arm of chromosome 5, also known as `5p monosomy`. Consider the class definition in Listing 10.1. Here, we define a male individual with CdCS by explicitly defining the chromosome 5 deletion event (breakpoints `p15` and `pTer`). Obviously this definition is of a primitive class, but if it were defined, potentially all individuals with CdCS could be inferred to be subclasses which has obvious diagnostic implications.

```
(defclass k46_XY_del(5p15)
 :super
  (some derivedFrom k46_XY)
  (exactly hasEvent 1
   (and
    Deletion
    (some hasBreakPoint HumanChromosome5Bandp15)
    (some hasBreakPoint HumanChromosome5BandpTer))))
```

Listing 10.1: Tawny-OWL definition of an individual with CdCS.

In summary, with The Karyotype Ontology, we have: produced a useful computational artefact that can be used to maintain and query karyotypes (which fulfils RQ1); pioneered a new paradigm for ontology development (see Section 10.2); and lastly, left the research community with lots of new questions to contemplate.

10.4 The Mitochondrial Disease Ontology

In this thesis, we have described the development process for an ontology of mitochondrial pathology. Unlike The Karyotype Ontology, there is no existing specification that we wish to formalise. Instead, our knowledge about mitochondria is contained in numerous sources e.g. databases and papers. Therefore, we first had to apply some form of knowledge capture to elicit the relevant information. In this thesis, we describe three different techniques for knowledge elicitation; we found that term extraction from published papers was the most useful and appropriate knowledge source for the first iteration of our ontology.

Once we had extracted a list of possible terms for our ontology, it was necessary to apply some form of refinement. This stage was twofold as it required canonicalisation of all terms (i.e. removing acronyms and other forms of duplications that were not obvious in the initial capture stage), then the filtering of disease relevant terms only. In order to aid us in the canonicalisation process, simple heuristics (string searches) were employed to identify possible duplicate terms.

However, this work was conducted previous to the patternised described in Section 10.2. It would be interesting to consider how this workflow might be changed with a patternised approach. For example could we have had users building Google docs or spreadsheets and re-import this live data from them on demand? Our data cleaning steps would have been encoded in Clojure and then directly transformed with Tawny-OWL patterns into OWL; we could also have recycled the knowledge to the users by using spreadsheets in a similar manner to RightField and Populous.

Next we started to build The Mitochondrial Disease Ontology for its purpose; to provide a Knowledge Base (KB) for exploration, rather than a reference ontology, which is the current state of the ontology. This commonly occurs as ontology engineering is a lengthy process. An example "ontology"[4] that facilitates a KB for exploration (and accomplishes what we wish to achieve), is discussed in the drug discovery work conducted in [26].

Ultimately, though, the mitochondrial work was limited by the tooling. The mito-

[4]The author is careful not to describe his work as an ontology.

chondrial work came to a standstill, and it was not until Tawny-OWL was developed (in co-ordination with the karyotype work) that the mitochondrial work was able to be updated and continued. Even with the shortfall, we believe that the agile engineering approach we followed was strong, and that it can be extended, by further exploiting parallels with software engineering.

As future work, we wish to continue the construction of the ontology and evaluate of the ontology in the context of an otherwise useful task; for example, classifying and clarifying of mitochondrial disease by their symptomatic or genomic definition. This use has been proven to be successful when classifying phosphatase proteins [166], or comparing annotation similarity with sequence similarity [69]. We believe this work would be of particular interest to the biologists and, as with the classification of phosphatases, it raises the possibility of computerising the discovery of new biological understanding, such that, like the robot scientist Adam [64], we could remove some of the labour intensive (and sometimes boring) testing of hypothesis.

In summary, with The Mitochondrial Disease Ontology, we have produced a useful computational artefact that will potentially have the ability to classify and clarify mitochondrial disease by their symptomatic and/or genomic definition (which fulfils RQ2); and clearly identified the future work required of this model.

10.5 Re-purposing software engineering

For this work, we made extensive use of patterns. As mentioned in Chapter 2, patterns were originally popularised in the context of software engineering. Here, we reflect on other software engineering techniques that we have re-purposed in order to aid the ontology engineering. This mean that there is no research in this section; instead we conduct a reflective report on our success (or failure) with the re-purposed techniques.

One example of a software engineering process re-purposed for ontologies is versioning; while the management of this process (revision control), allows collaborative development, it can be controlled by client-server systems known as Concurrent Versions System (CVS). There are existing bespoke tools that attempt to emulate these systems, such as ContentCVS [129], which is OWL specific. There are also collaborative environments such as WebProtégé [153], which provides some CVS functionality, i.e. a collaborative engineering of ontologies, but WebProtégé does not include features such as history. However, there is one well-known problem with versioning ontologies; these revision control tools depend implicitly on diff[5] and irrespective of the ontology's syntax, one small change in the ontology can create many changes in the file. Resource Description Framework (RDF) serialisations, in fact, are the worst perpetrators for this change [98]. We could utilise bespoke diff tools for OWL, such as `ecco` [45], which applies both semantic and syntactic diff techniques, but find that all we really need is a human-readable, line-oriented syntax, such as Manchester Syntax or the OBO flat file format [98], which Tawny-OWL emulates. In addition, with the use of patterns, we find that semantic diff is less useful as one pattern can change many entities. Instead, we find that a non-semantic diff works fine as an end user tool.

During this thesis we made extensive use of revision control; this was accomplished using the revision control system Git and GitHub (an online community Git repository). With GitHub, each project has its own repository. Version control was paramount as all ontologies were built in a modular fashion, with numerous com-

[5] A utility that shows the changes between two versions of the same file.

mits over a long period of time.

Another example of a re-purposed software engineering technique is Unit testing [159]. In software engineering, unit tests are generally used to aid development and maintenance by verifying that units of code behave as intended [13]. In our case, the unit tests provided a strict way of isolating problems early in the ontology construction and pattern generation. As Tawny-OWL is built on Clojure, a general purpose language, we naturally inherited the ability to implement tests via `clojure.test`. Further, as Tawny-OWL is extensible, regression testing accelerated and facilitated change. In addition, we have the ability to automatically generate unit tests. For example, with the use of the Incanter library[6], we auto-generated each OWL Karyotype unit test, using the data stored in the spreadsheet.

Once tests are implemented, we also have the ability to integrate with CI systems, specifically TravisCI[7], to automatically run all tests after every push to the repository.

In software engineering, Maven is used to automatically build and manage projects. This is achieved using a Project Object Model (POM), specifically an XML file which describes the software project being built, its dependencies, the build order, directories and any associated plug-ins. Maven uses this POM to dynamically download associated libraries and stores them to a local cache. Through the use of Maven, we get all of the associated software e.g. HermiT [137], ELK [170] and Tawny-OWL itself. Unlike OntoMaven [112], an extension of Maven, this functionality was naturally occurring in the development of Tawny-OWL.

Maven also provides us with a new way of distributing ontologies by allowing the reuse of existing ontology components into local development repositories. This mean that for each repository the open source libraries can be uploaded to clojars[8], a free online Clojure community repository, and pulled in via Maven dependencies.

As mentioned in Section 2.3, the interaction required by Graphical User Interface (GUI) ontology engineering tools can be time-consuming, especially when there are

[6] `https://github.com/incanter/incanter`
[7] `https://travis-ci.org/`
[8] `https://clojars.org/`

many changes to make. In this regard, Tawny-OWL is superior. However, Protégé is still one of the most popular editors for building ontologies [161]; part of this is most likely due to its visualisation capabilities, something that Tawny-OWL lacks. However, with numerous extensions[9] this limitation is alleviated [77].

In contrast to the GUI visualisation, Tawny-OWL also has Integrated Development Environment (IDE) interoperability with the Emacs text editor, and supports a full textual representation of the Clojure code. We use the `tawny-mode`[10] Emacs package that, with a full featured Read-Eval-Print Loop (REPL), allows us to dynamically explore, create and reason ontologies. In addition, it supports the encoding of ontologies in Tawny-OWL by providing documentation lookup and applying custom syntax highlighting. Further, we use the `omn-mode`[11] Emacs package to visualise the ontology in Manchester Syntax format[12] [71, 73].

As briefly mentioned in Section 1.1, lints also originated from software engineering. One well-known ontology lint used by the community is OOPS!, which is currently in the implementation stage[13]. However, we need to consider not only the ontology but the code that was used to generate the ontology. While we have tools such as eastwood[14], which help with the static analysis of Clojure code, it would be nice to have a lint for Tawny-OWL itself. In addition, if this Tawny-OWL lint could operate at the level of the patterns and allow the flexible redefinition of code, we could use it to help with the refactoring of SIO (Section 8.1 Step 2) and systematically handle the syntax tree. We could attempt this by defining rules, which is possible through standard Clojure lints, such as kibit[15].

Further, there are many other software engineering techniques that we have not considered. For example, is it possible to incorporate Tawny-OWL into a highly collaborative editing environment, such as WebProtégé? Additionally, documentation is an important but time-consuming part of any computational artefact. In order to aid the documentation process, could we implement some form of auto-

[9] See `Protege-Nrepl`, `Tawny-Protege` and `Lein-sync`
[10] `https://github.com/phillord/tawny-owl/blob/master/emacs/tawny-mode.el`
[11] Found at `https://github.com/phillord/phil-emacs-packages`.
[12] By saving the ontology in Manchester Syntax format
[13] `https://github.com/jaydchan/tawny-owl/tree/oops-service`
[14] `https://github.com/jonase/eastwood`
[15] `https://github.com/jonase/kibit`

matic live [115] documentation or employ literate documentation [75]? Could we provide the ability to import or create live spreadsheets, thus emulating RightField and Populous? As briefly discussed in Chapter 6, we believe that the best way of modelling the `affects` relation is dependent on the user's preference. Could we implement this through the use of contextual imports [24]? Lastly, could we improve Tawny-OWL itself; for example implement tools that explain the origin of pattern generated ontology components or provide ontology statistics?

In summary, we have discussed the 5 and 8 re-purposed software engineering techniques and technologies respectively, used in the creation of our ontologies. Furthermore, 3 techniques and 4 technologies have been identified as future work. Thus this section concludes our research into the advantages and benefits of applying a pattern-driven and programmatic approach to ontology engineering (RQ3). See Chapter E for a summary of all advantages and benefits discussed in this thesis.

10.6 Improving the ontology engineering process

During the work presented in this thesis, we have identified numerous features and properties regarding the usage of a pattern-driven approach to ontology engineering. Within this section, we summarise each topic that can help enhance the (computing science) ontology community.

As discussed in Section 2.3, there are a variety of bespoke tools available for the construction of OWL ontologies (e.g. Protégé and Populous). However, in this thesis, rather than use bespoke tools, we have successfully used numerous re-purposed software engineering techniques and technologies for use with Tawny-OWL, in order to build our two domain ontologies. For example, Git and GitHub were used to manage the versioning of the ontology, whilst `clojure.test` and TravisCI enabled the implementation of unit testing and the automatic running of our tests after every push to the project repository. In total, we discuss 13 re-purposed and 7 potential software engineering tools and techniques (see Section 10.5). The usage of these tools was personally, of great benefit in the ontology engineering, thus we propose that, rather than invent further bespoke ontology tools, we should re-purpose the already existing software engineering tools.

Secondly, we see that ontology patterns are useful in ontology engineering; all the more so for ontologies that are highly repetitive. For example, whilst The Karyotype Ontology is probably an exceptional case, (97.7% of the logical axioms were generated through the use of patterns, see Table 9.1), it was shown to not be unique. However, if ontology patterns are to be useful, we need them to be able to encode them computationally and subsequently generate ontologies with them. Existing tools that allow the encoding of patterns, may require repetitive and time-consuming GUI interactions and/or the separation of non-patternised and patternised parts of the ontology. With Tawny-OWL these issues are circumvented.

In this thesis, we see that localised patterns are of more importance than generic patterns. This was especially true for The Karyotype Ontology. However, this statement is already obvious, as in software engineering, the implementation of functions and removal of code replication occurred way before we (the computing science

community) started talking about design patterns.

Finally, in this work, we have shown that it is possible to incorporate software to build our ontologies. An example of this is shown in the construction of The Mitochondrial Disease Ontology; The Mitochondrial Disease Ontology is built with Clojure, Tawny-OWL the OWL API, and is based on numerous text files and spreadsheets. Here, we consider whether the converse is also possible i.e. can ontology elements be built into the software. An example of this would be a visual tool for the construction of karyotype classes. In fact through the use of Clojure and Tawny-OWL, this would be possible as we can embed our ontology directly into our programming language.

Taken together, this approach to ontology engineering defines a new paradigm, and allows us to move ontologies beyond tools for generating reference canonicalisations, toward becoming tools for the exploration of knowledge and the creation of new understanding.

A

Recast of The Pizza Ontology

```
(defontology pizzaontology
  :iri "http://www.ncl.ac.uk/pizza"
  :prefix "piz:")

(defdproperty hasCalorificContentValue)
(defaproperty myOpinion)
(defoproperty hasTopping
  :label "rdfs-label test")

(as-disjoint
 (defclass Pizza
  :label "Pizza")
 (defclass PizzaComponent))

(as-subclasses PizzaComponent
 :cover :disjoint
 (defclass PizzaTopping)
 (defclass PizzaBase))

(defclass NamedPizza
  :annotation (annotation myOpinion "rdfs-annotation test"))
(defclass TomatoTopping
  :annotation (see-also "rdfs-seealso test"))
(defclass MozzarellaTopping
  :comment "rdfs-comment test")

(defclass MargheritaPizza
 :super
  NamedPizza
  (some-only hasTopping
  TomatoTopping MozzarellaTopping))

(defindividual ExampleMargheritaPizza
  :type MargheritaPizza
  :fact (fact hasCalorificContentValue 300))
```

Listing A.1: A subset of The Pizza Ontology

B

Tawny-OWL: Supplementary Material

Contents

B.1 Tawny-OWL restriction exemplars 202

B.2 Tawny-OWL entity exemplars 204

B.3 Tawny-OWL frames 206

B.4 Defining a namespace in Tawny-OWL 209

B.1 Tawny-OWL restriction exemplars

In Tawny-OWL exact cardinality restrictions are declared using the `exactly` function. An example exact cardinality restriction can be seen in Listing B.1. This restriction is used by the `FourCheesePizza` defined class to find all `Pizza`s that have exactly four `CheeseTopping`s.

```
(defclass FourCheesePizza
  :equivalent
  (and Pizza
       (exactly 4 hasTopping CheeseTopping)))
```

Listing B.1: An example exact cardinality restriction definition.

In Tawny-OWL minimum cardinality restrictions are declared using the `at-least` functions. An example minimum cardinality restriction can be seen in Listing B.2. This restriction is used to by the `InterestingPizza` defined class to find all `Pizza`s that have at least three `CheeseTopping`s. Conversely maximum cardinality restrictions are declared using the `at-most` function.

```
(defclass InterestingPizza
  :equivalent
  (and Pizza
       (at-least 3 hasTopping PizzaTopping)))
```

Listing B.2: An example minimum cardinality restriction definition.

In Tawny-OWL has-value restrictions are declared using the `has-value` function. An example has-value restriction can be seen in Listing B.3. This restriction declares that the calorific content value is 150.

```
(has-value hasCalorificContentValue 150)
```

Listing B.3: An example basic class definition

In Tawny-OWL datatype restrictions are declared using the `span` function. An example datatype restriction is shown in Listing B.4. Using the `>=<` symbol, we define a min and max inclusive datatype restriction. This restriction is used by the `MediumCaloriePizza` defined class to find all `Pizza`s that have a calorific value between 400 and 700.

```
(defclass MediumCaloriePizza
  :equivalent
  (and Pizza
       (some hasCalorificContentValue
             (span >=< 400 700))))
```

Listing B.4: An example existential restriction definition using a datatype property.

B.2 Tawny-OWL entity exemplars

Once a class is defined in Tawny-OWL, we can start to define OWL individuals. As an example basic individual definition see Listing B.5. The `defindividual` function builds on `individual`, such that `individual` creates the OWL API OWLIndividual object and `defindividual` creates a symbol, in this case the `ExampleMargheritaPizza`. Subsequent use of this symbol will resolve to the individual object.

```
(defindividual ExampleMargheritaPizza
 :type MargheritaPizza
 :fact (fact hasCalorificContentValue 300))
```

Listing B.5: An example basic individual definition.

The `:type` and `:fact` frames are used to add one or more class assertion axioms and facts respectively to the individual. In Tawny-OWL, facts can be generated using the `fact` or `is` functions. All available individual frames are shown in Table B.3.

In an ontology an ontologist can define three types of properties: object property, annotation property and datatype property. In Tawny-OWL these are declared majorly using the `defoproperty`, `defaproperty` and `defdproperty` respectively. In this section we show an example declaration of each property type and supply a brief description of some of the associated frames for each.

```
(defoproperty hasBase
 :super hasIngredient
 :characteristic :functional
 :range PizzaBase
 :domain Pizza)
```

Listing B.6: An example basic object property definition.

The `:super` frame is used to add one or more superproperties to the property using the `SubPropertyOf` axiom. In Listing B.6 we assert that the `hasBase` object property is a subproperty of the `hasIngredient` object property. The `:characteristic` frame is used to add a list of characteristics to the property. In Listing B.6, we are defining the `hasBase` object property to be a functional property. This states that every `Pizza` can only have one `PizzaBase`. The `:range` and `:domain` frames are used to add one or more property range or domain axioms to the property. All available object property

frames are shown in Table B.4.

As an example basic annotation property definition see Listing B.7. The `defaproperty` function builds on the `annotation` function, such that `annotation` creates the OWL API OWLAnnotationProperty object and `defaproperty` creates a symbol. in this case `myOpinion`.

```
(defaproperty myOpinion
 :super owl-comment-property
 :label "My Opinion"
 :comment "Do I think this is a good pizza to eat?")
```

Listing B.7: An example basic annotation property definition

Like the `defoproperty`, the `:super` frame is used to add one or more superproperties to the property using the `SubPropertyOf` axiom. In Listing B.7 we assert that the `myOpinion` annotation property is a subproperty of the `owl-comment-property`, which is a predefined Tawny-OWL symbol that refers to the `rdfs:comment` annotation property. All available annotation property frames are shown in Table B.5.

As an example basic datatype property definition see Listing B.8. The `defdproperty` function builds on the `datatype-property` function, such that `datatype-property` creates the OWL API OWLDataProperty object and `defdproperty` creates a symbol, in this case `hasCalorificContentValue`.

```
(defdproperty hasCalorificContentValue
 :range :XSD_INTEGER)
```

Listing B.8: An example basic datatype property definition

Like the `defoproperty`, the `:range` frame is used to add one or more property range axioms to the property. In Listing B.8 we assert that the `hasCalorificContentValue` datatype property has a property range axiom value of `:XSD_INTEGER`, which is a transformed Tawny-OWL keyword that refers to the OWL2Datatype constant[1]. All available datatype property frames are shown in Table B.6.

[1] `http://owlapi.sourceforge.net/javadoc/org/semanticweb/owlapi/vocab/OWL2Datatype.html#XSD_INTEGER`

B.3 Tawny-OWL frames

Table B.1: Table showing the variety of frames available to define an OWL ontology in Tawny-OWL. Data taken from the Tawny-OWL repository.

Ontology frame	Description
`:iri`	Sets the iri for the ontology
`:iri-gen`	Adds an IRI gen function to the ontology options
`:prefix`	Sets the prefix for the ontology
`:name`	Adds a tawny-name annotation axiom to the ontology, unless the `:noname` ontology option is specified
`:seealso`	Adds a see also annotation axiom to the ontology
`:comment`	Adds a comment annotation axiom to the ontology
`:versioninfo`	Adds a version information annotation axiom to the ontology

Table B.2: Table showing the variety of frames available to define an OWL class in Tawny-OWL. The `:subclass` frame, previously used to define `SubClassOf` restrictions, has not included as it has now been deprecated. Data taken from the Tawny-OWL repository.

Class frame	Description
`:sub`	Adds one or more subclasses to the class
`:super`	Adds one or more superclasses to the class
`:haskey`	Adds a haskey axiom to the class
`:annotation`	Adds an annotation axiom to the class
`:comment`	Adds a comment annotation axiom to the class
`:label`	Adds a label annotation axiom to the class
`:equivalent`	Adds an equivalent class axiom to the ontology
`:disjoint`	Adds a disjoint class axiom to the ontology

Table B.3: Table showing the variety of frames available to define an OWL individual in Tawny-OWL. Data taken from the Tawny-OWL repository.

Individual frame	Description
`:type`	Adds one or more class assertion axioms to the individual
`:fact`	Adds one or more facts to the individual
`:same`	Adds one or more individuals as same individual axioms to the ontology
`:different`	Adds one or more individuals as different individual axioms to the ontology

Table B.4: Table showing the variety of frames available to define an OWL object property in Tawny-OWL. The `:subproperty` frame and `:subpropertychain` frame, previously used to define `SubPropertyOf` and `SubPropertyChainOf` restrictions respectively, has not included as it has now been deprecated. Data taken from the Tawny-OWL repository.

Property frame	Description
`:domain`	Adds one or more domain axioms to the property
`:range`	Adds one or more range axioms to the property
`:inverse`	Adds one or more inverse axioms to the property
`:sub`	Adds one or more subproperties to the property
`:super`	Adds one or more superproperties to the property
`:subchain`	Adds a property chain to property
`:characteristic`	Adds a list of characteristics to the property. Available characteristics include `:transitive`, `:functional`, `:inversefunctional`, `:symmetric`, `:asymmetric`, `:irreflexive` and `:reflexive`
`:disjoint`	Adds a disjoint property axiom to the ontology
`:equivalent`	Adds an equivalent property axiom to the ontology
`:annotation`	Adds an annotation axiom to the property
`:comment`	Adds a comment annotation axiom to the property
`:label`	Adds a label annotation axiom to the property

Table B.5: Table showing the variety of frames available to define an OWL annotation property in Tawny-OWL. The `:subproperty` frame, previously used to define `SubPropertyOf` restrictions, has not included as it has now been deprecated. Data taken from the Tawny-OWL repository.

Property frame	Description
`:super`	Adds one or more superproperties to the property
`:annotation`	Adds an annotation axiom to the property
`:comment`	Adds a comment annotation axiom to the property
`:label`	Adds a label annotation axiom to the property

Table B.6: Table showing the variety of frames available to define an OWL datatype property in Tawny-OWL. The `:subproperty` frame, previously used to define `SubPropertyOf` restrictions, has not included as it has now been deprecated. Data taken from the Tawny-OWL repository.

Property frame	Description
`:domain`	Adds one or more data domain axioms to the property
`:range`	Adds one or more data range axioms to the property
`:sub`	Adds one or more subproperties to the property
`:super`	Adds one or more superproperties to the property
`:characteristic`	Adds a list of characteristics to the property. Available characteristics include `:functional`
`:disjoint`	Adds a disjoint property axiom to the ontology
`:equivalent`	Adds an equivalent property axiom to the ontology
`:annotation`	Adds an annotation axiom to the property
`:comment`	Adds a comment annotation axiom to the property
`:label`	Adds a label annotation axiom to the property

B.4 Defining a namespace in Tawny-OWL

Before an OWL ontology can be defined, a Clojure namespace must be defined. This utility is used to prevent name clashes. Similar to Java, Clojure namespaces can also contain packages which map to directories. The namespace defined in Listing B.9 maps to a file in the `src/pizza/pizza.clj`. Listing B.9 shows an example of defining a namespace using the Clojure `ns` macro. By default it will create a new namespace that contains mappings to functions in `clojure.core` and classnames in `java.long` and `clojure.lang.Compile`. Further mappings can be included and/or excluded using the namespace keywords such as `:refer-clojure`, `:use` and `:require`.

```
(ns pizza.pizza
  (:refer-clojure :exclude [and or not some])
  (:use [tawny.owl]
        [tawny.english])
  (:require [tawny.reasoner :as r]))
```

Listing B.9: An example basic namespace definition

The `:use` keyword is used to import another namespace's code as though it was declared in your namespace i.e. it does not need qualification. In Listing B.9 we are importing all `tawny.owl` and `tawny.english`[2] symbols. In order ensure that no namespace clashes occur when we import `tawny.english`, the `:refer-clojure` keyword is used apply restrictions to `clojure.core` mappings, such that `and`, `or`, `not` and `some` functions are not imported into the current namespace. The `:require` keyword however is used to refer to another namespace's code without importing it into the current namespace. In Listing B.9 we define a shortcut for the `tawny.reasoner` library by using the `:as` option.

Once a namespace is available, we can start to manipulate ontologies in Tawny-OWL.

[2]Shortcut aliases for `tawny.owl` functions with an `owl-` disambiguation.

C

Mitochondria: Supplementary Material

Table C.1: Table highlighting certain statistics of the papers used in the term capture stage. Here we show the order of the papers, and for each paper provide the publishing year, the number of terms manually extracted (with and without collection duplicates) and its associated reference.

Paper No.	Year	Batch No.	Terms	Unique terms	Reference
1	2011	1	101	101	[111]
2	2011	1	245	245	[120]
3	2004	1	284	268	[123]
4	2006	1	178	152	[11]
5	2006	1	208	186	[39]
6	2009	2	122	103	[124]
7	2005	2	433	416	[148]
8	2000	2	90	86	[16]
9	2003	2	178	174	[104]
10	2003	2	187	161	[21]
11	2006	3	54	54	[151]
12	2005	3	82	77	[152]
13	2004	3	383	314	[171]
14	2009	3	105	103	[53]
15	2000	3	61	54	[54]
16	2008	4	109	82	[25]
17	2001	4	80	73	[67]
18	2006	4	62	51	[84]
19	2003	4	2	1	[164]
20	2007	4	56	44	[117]
21	2006	5	106	78	[133]
22	2010	5	108	97	[9]
23	2005	5	68	62	[8]
24	2004	5	36	34	[61]
25	2002	5	19	17	[138]

Continued on next page...

Table C.1 – continued from previous page

Paper No.	Year	Batch No.	Terms	Unique terms	Reference
26	2011	6	130	122	[108]
27	2011	6	11	10	[107]
28	2010	6	79	66	[83]
29	2010	6	22	19	[110]
30	2011	6	67	61	[48]
		Total:	3666	3311	

Table C.2: Table showing variety of mitochondrial disease terms taken from the UMDF website. Adapted to include related OMIM links.

Name	OMIM ID	Long Name
Alpers Disease	203700	–
Barth Syndrome	302060	–
Beta-oxidation Defects	–	–
Cernitine-Acyl-Carnitine Deficiency	–	–
Cernitine Deficiency	212160	–
Creatine Deficiency Syndromes	–	–
Co-Enzyme Q10 Deficiency	–	–
Complex I Deficiency	252010	NADH dehydrogenase (NADH-CoQ reductase) deficiency
Complex II Deficiency	252011	Succinate dehydrogenase deficiency
Complex III Deficiency	516020	Ubiquinone-cytochrome c oxidoreductase deficiency
Complex IV Deficiency	220110	COX deficiency
Complex V Deficiency	516060	ATP synthase deficiency
CPEO	157640	Chronic Progressive External Ophthalmoplegia Syndrome
CPT I Deficiency	–	–
CPT II Deficiency	600650	–
KSS	530000	–
Lactic Acidosis	–	–
LBSL	611105	–
LCAD	–	–
LCHAD	609016	–
Leigh Disease	256000	–
Luft Disease	–	–

Continued on next page...

Table C.2 – continued from previous page

Name	OMIM ID	Long Name
MAD	231680	–
MCAD	201450	–
MELAS	540000	–
MERRF	545000	–
MIRAS	–	Mitochondrial Recessive Ataxia Syndrome
Mitochondrial Cytopathy	–	–
Mitochondrial DNA Depletion	–	–
Mitochondrial Encephalopathy	–	–
Mitochondrial Myopathy	251900	–
MNGIE	603041	–
NARP	551500	–
Pearson Syndrome	557000	–
Pyruvate Carboxylase Deficiency	266150	–
Pyruvate Dehydrogenase Deficiency	312170	–
POLG Mutations	–	–
Respiratory Chain	–	–
SCAD	201470	–
SCHAD	601609	–
VLCHAD	201475	–

D

Classification: Supplementary Material

Table D.1: Table showing the variety of ODP classification categories and their description. Data taken from the Ontology Design Patterns wiki (see `http://ontologydesignpatterns.org/wiki/OPTypes`).

Name	Code	Description
Structural	Str	Structural ODPs includes Logical ODPs and Architectural ODPs.
Logical	Log	A Logical OP is a formal expression, whose only parts are expressions from a logical vocabulary, e.g. OWL Description Logic (DL), that solves a problem of expressivity.
Architectural	Arc	Architectural ODPs affect the overall shape of the ontology: their aim is to constrain 'how the ontology should look like'.
Correspondence	Cor	Correspondence OPs include Reengineering OPs and Alignment OPs.
Re-engineering	Ren	Reengineering OPs are transformation rules applied in order to create a new ontology (target model) starting from elements of a source model.
Alignment	Ali	Alignment OPs refer to correspondences between ontologies. Each pattern models a relation between two entities or sets of entities in two ontologies. Instantiation of an Alignment OP results in a correspondence between elements of two given ontologies.
Content	Con	CPs are distinguished networked ontologies and have their own namespace. They cover a specific set of competency questions (requirements), which represent the problem they provide a solution for. Furthermore, CPs show certain characteristics, i.e. they are: computational, small, autonomous, hierarchical, cognitively relevant, linguistically relevant, and best practices. See evaluation principles for more details.
Reasoning	Rea	Reasoning OPs are applications of Logical OPs oriented to obtain certain reasoning results, based on the behavior implemented in a reasoning engine.
Presentation	Pre	Presentation ODPs deal with usability and readability of ontologies from a user perspective.
Naming	Nam	Naming OPs are conventions on how to create names for namespaces, files, and ontology elements in general (classes, properties, etc.).
Annotation	Ann	Annotation ODPs provide annotation properties or annotation property schemas that are meant to improve the understandability of ontologies and their elements.
Lexico-Syntactic	Lex	Lexico-Syntactic OPs are linguistic structures or schemas that consist of certain types of words following a specific order, and that permit to generalize and extract some conclusions about the meaning they express.

Table D.2: Table showing the variety of ODP classification categories and their description. Data taken from the Ontology Design Patterns (ODPs) Public Catalog (see `http://www.gong.manchester.ac.uk/odp/html/`).

Name	Code	Description
Extension ODPs	E	patterns that by-pass the limitations of OWL.
Good Practice ODPs	G	patterns that obtain a more robust, cleaner and easier to maintain ontology.
Domain Modelling ODPs	D	patterns that are solutions to concrete modelling problems in biology.

E

Summary of research questions

Contents

E.1 Summary of research questions 222

E.1 Summary of research questions

Throughout this thesis we have discussed the research questions and how and when they have been fulfilled. Due to the distributed manner of this knowledge, in this chapter we collate the evidence and summarise each research question.

RQ1 How can we build a computational representation of the ISCN using a pattern-driven and programmatic approach?

In this thesis, we have shown the step-by-step construction of the computational representation of the ISCN.

1. Use a programmatic environment that has the ability to build ontologies: In Chapter 3, we introduce Tawny-OWL and Clojure (what it is built on) using exemplars from The Pizza Ontology
2. Ensure that the programmatic environment has the ability to encode patterns. In Chapter 4, we detail how patterns can be encoded function using Tawny-OWL using further exemplars from The Pizza Ontology.
3. Use the software engineering life cycle to build the computational model. In Chapter 5: we analyse the current specification and identify the requirements; detail the design and implement of The Karyotype Ontology; and lastly test The Karyotype Ontology using exemplars from the ISCN.
4. Improve the community's understanding of localised patterns. In Chapter 9 we introduce a novel classification of localised patterns and provide basic stats of the localised patterns used to build The Karyotype Ontology.

RQ2 Can we apply this approach to model new areas of biology and to produce useful computational artefacts?

In this thesis, we have shown the construction of two novel bio-ontologies using the pattern-driven and patternised approach.

1. In Chapter 5, we show that this approach is applicable to a bounded and mature biological domain, specifically karyotypes. The resulting computational artefact, The Karyotype Ontology, will potentially be valuable for cytogeneticists by transforming collections of karyotypes to a form that is easy to query, validate and maintain.

2. In Chapter 7, we show that this approach is applicable to a complex developing biological domain, specifically mitochondria. The resulting computational artefact, The Mitochondrial Disease Ontology, is our first step to building an ontology that will potentially have the ability to classify and clarify mitochondrial disease by their symptomatic and/or genomic definition.

RQ3 What are the advantages and benefits of applying this approach to ontology engineering?

In this thesis, we have shown 13 advantages and benefits of applying a pattern-driven and patternised approach to ontology engineering.

1. Compatibility with existing structured data. This separate the knowledge from the axiomitisation. In Chapter 4 we show that we can interact with external files. In Chapter 7 we incorporate knowledge into The Mitochondrial Disease Ontology from a variety of sources and sources.

2. Easy to iterate over all entities. In Chapter 4 we use the `whodunit` pattern to apply a creator annotation to all entities in the ontology. In addition, we use the load-labels pattern to apply a Italian name annotation to all entities (where possible).

3. Complex patterns can be abstracted over for downstream users. In Chapter 4 we show numerous examples where once encoded, these patterns can be potentially used without requiring the user to know the technical detail underneath.

4. Aid developer comprehensibility. In Chapter 4, we show that the patternised and non-patternised parts of the ontology are expressed in the

same syntax. In addition, the patternised and non-patternised parts of the ontology are expressed in the same environment. In Chapter 8 we hide the SIO IDs and provide the developer with usable Clojure names. Similarly in Chapter 9, we introduce the OBO ID pattern which hides the OBO IDs.

5. Enforces consistency: a computer is less error-prone (than humans) and is not likely to get bored. In Chapter 5 we show that by encoding the ISCN (and exemplars) we were able to identify two missing and identify several exemplars that refer to a band that does not exist. In Chapter 8 we show that by refactoring SIO there are various errors within the ontology. In addition, by encoding the downstream patterns we found errors in the SIO wiki (which documents these downstream patterns).
6. Automatically (re-)generate ontologies. A computer can accomplish this quicker and easier than a human. In addition, any changes made to the code (including patterns) also effect the ontology. Thus ontologies are potentially easy to maintain. In Chapter 6 we use this ability to programmatically generate numerous random ontologies (not always applicable).
7. Means to test the (reasoning) performance of an ontology. In Chapter 6, we test the performance by calculating the mean time taken to reason our ontology.
8. Means to test the scalability of an ontology (not always applicable). In Chapter 6 we use auto-generated random ontologies of various sizes and the reasoning performance to find that The Karyotype Ontology can comfortably scale to 10^5.
9. Means to test the extensibility i.e. what effect does different modelling representations on an ontology. In Chapter 6 we use auto-generated random ontologies of various axiomitisations (and sizes) as well as the reasoning performance to find that all three representations could be valid though it is dependent on the value of `K`.
10. Encourages concision. In Chapter 8 we show this by carefully refactoring

SIO into a patternised form and noting the differences for size of file, number of lines and load time.

11. The ability to encode patterns for downstream usage: potentially encourage downstream usage and/or useful to downstream users. In Chapter 8, we discuss the downstream patterns documented on the SIO wiki. As shown in Chapter 9, each event and feature has an associated localised pattern. Uses these patterns we were able to implement the ISCN exemplars as a form of application-dependent testing (results are discussed in Chapter 5).

12. Useful in the construction of ontologies and downstream usage of ontologies. In Chapter 9 we show document the localised patterns used in all three bio-ontologies. More than 70% of the ontologies were constructed using localised patterns. In each ontology, there is at least one pattern that generates approximately >50% of the logical axioms.

13. Compatibility with existing software tools and software engineering techniques. In Chapter 10 we discuss 13 re-purposed and 7 potential software engineering tools and techniques e.g. Git, GitHub, `clojure.test` and CI.

References

[1] DCMI Metadata Terms. `http://dublincore.org/documents/dcmi-terms/`. Accessed: June 2015.

[2] Abbreviations and symbols for nucleic acids, polynucleotides and their constituents. *European Journal of Biochemistry*, 15(2):203–208, 1970.

[3] R Apweiler, A Bateman, M. J Martin, C O'Donovan, M Magrane, Y Alam-Faruque, E Alpi, R Antunes, J Arganiska, E Barrera Casanova, B Bely, M Bingley, C Bonilla, R Britto, B Bursteinas, W Mun Chan, G Chavali, E Cibrian-Uhalte, A Da Silva, M De Giorgi, F Fazzini, P Gane, L. G Castro, P Garmiri, E Hatton-Ellis, R Hieta, R Huntley, D Legge, W Liu, J Luo, A MacDougall, P Mutowo, A Nightingale, S Orchard, K Pichler, D Poggioli, S Pundir, L Pureza, G Qi, S Rosanoff, T Sawford, A Shypitsyna, E Turner, V Volynkin, T Wardell, X Watkins, H Zellner, M Corbett, M Donnelly, P van Rensburg, M Goujon, H McWilliam, R Lopez, I Xenarios, L Bougueleret, A Bridge, S Poux, N Redaschi, L Aimo, A Auchincloss, K Axelsen, P Bansal, D Baratin, P. A Binz, M. C Blatter, B Boeckmann, J Bolleman, E Boutet, L Breuza, C Casal-Casas, E de Castro, L Cerutti, E Coudert, B Cuche, M Doche, D Dornevil, S Duvaud, A Estreicher, L Famiglietti, M Feuermann, E Gasteiger, S Gehant, V Gerritsen, A Gos, N Gruaz-Gumowski, U Hinz, C Hulo, J James, F Jungo, G Keller, V Lara, P Lemercier, J Lew, D Lieberherr, T Lombardot, X Martin, P Masson, A Morgat, T Neto, S Paesano, I Pedruzzi, S Pilbout, M Pozzato, M Pruess, C Rivoire, B Roechert, M Schneider, C Sigrist, K Sonesson, S Staehli, A Stutz, S Sundaram, M Tognolli, L Verbregue, A. L Veuthey, C. H Wu, C. N Arighi, L Arminski, C Chen, Y Chen, J. S Garavelli, H Huang, K Laiho, P McGarvey, D. A Natale, B. E Suzek, C Vinayaka, Q Wang, Y Wang, L. S Yeh, M. S Yerramalla, and J Zhang. Activities at the Universal Protein Resource (UniProt). *Nucleic Acids Res.*, 42(Database issue):D191–198, Jan 2014.

[4] M Aranguren, S Bechoffer, P Lord, U Sattler, and R Stevens. Understanding and using the meaning of statements in a bio-ontology: recasting the Gene Ontology in OWL. *BMC Bioinformatics*, 8:57+, February 2007.

[5] M Ashburner, C. A Ball, J. A Blake, D Botstein, H Butler, J. M Cherry, A. P Davis, K Dolinski, S. S Dwight, J. T Eppig, M. A Harris, D. P Hill, L Issel-Tarver, A Kasarskis, S Lewis, J. C Matese, J. E Richardson, M Ringwald, G. M Rubin, and G Sherlock. Gene ontology: tool for the unification of biology. The Gene Ontology Consortium. *Nature Genetics*, 25(1):25–29, May 2000.

[6] M Bada, R Mcentire, C Wroe, and R Stevens. Goat: The gene ontology annotation tool. In *Proceedings of the 2003 UK e-Science All Hands Meeting*, pages 514–519, 2003.

[7] M Bada, R Stevens, C Goble, Y Gil, M Ashburner, J. A Blake, J. M Cherry, M Harris, and S Lewis. A short study on the success of the Gene Ontology. *Web Semantics Science Services and Agents on the World Wide Web*, 1(2):235–240, 2004.

[8] S. M Bailey, A Landar, and V Darley-Usmar. Mitochondrial proteomics in free radical research. *Free Radical Biology and Medicine*, 38(2):175–188, 2005.

[9] R Banjerdpongchai, P Kongtawelert, O Khantamat, C Srisomsap, D Chokchaichamnankit, P Subhasitanont, and J Svasti. Mitochondrial and endoplasmic reticulum stress pathways cooperate in zearalenone-induced apoptosis of human leukemic cells. *Journal of hematology and oncology*, 3:50, 2010.

[10] R Banks, V Khomenko, and L. J Steggles. A case for using signal transition graphs for analysing and refining genetic networks. *Electronic Notes in Theoretical Computer Science*, 227(0):3–19, 2009.

[11] S Basu, E Bremer, C Zhou, and D. F Bogenhagen. Migenes: A searchable interspecies database of mitochondrial proteins curated using gene ontology annotation. *Bioinformatics*, 22(4):485–492, 2006.

[12] W. A Baumgartner, K. B Cohen, L. M Fox, G Acquaah-Mensah, and L Hunter. Manual curation is not sufficient for annotation of genomic databases. *Bioinformatics*, 23(13):i41–i48, Jul 2007.

[13] K Beck. Simple smalltalk testing: with patterns. Technical Report 4 (2), 1994.

[14] M. J Bell. *Provenance, Propagation and Quality of Biological Annotation.* PhD thesis, Newcastle University, 2014.

[15] C Beneteau, S Baron, A David, F Jossic, D Poulain, S Schmitt, M.-D Leclair, P Piloquet, and C Le Caignec. Constitutional telomeric association (y;7) in a patient with a female phenotype. *American Journal of Medical Genetics Part A*, 161(6):1436–1441, 2013.

[16] J. L Blanchard and M Lynch. Organellar genes. why do they end up in the nucleus? *Trends in Genetics*, 16(7):315–320, 2000.

[17] E Blomqvist, A Gangemi, and V Presutti. Experiments on pattern-based ontology design. In *Proceedings of the Fifth International Conference on Knowledge Capture*, K-CAP '09, pages 41–48, New York, NY, USA, 2009. ACM.

[18] E Blomqvist, V Presutti, E Daga, and A Gangemi. Experimenting with extreme design. In P Cimiano and H Pinto, editors, *Knowledge Engineering and Management by the Masses*, volume 6317 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 2010.

[19] C Boelling, M Dumontier, M Weidlich, and H.-G Holzhütter. Role-based representation and inference of biochemical processes. In R Cornet and R Stevens, editors, *ICBO*, volume 897 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2012.

[20] E Camon, M Magrane, D Barrell, D Binns, W Fleischmann, P Kersey, N Mulder, T Oinn, J Maslen, A Cox, and R Apweiler. The Gene Ontology Annotation (GOA) project: implementation of GO in SWISS-PROT, TrEMBL, and InterPro. *Genome Res.*, 13(4):662–672, Apr 2003.

[21] P. F Chinnery and E. A Schon. Mitochondria. *Journal of neurology, neurosurgery, and psychiatry*, 74(9):1188–1199, 2003.

[22] P. F Chinnery. Mitochondrial disorders overview. 2000 Jun 8 [Updated 2014 Aug 14]. In: Pagon RA, Adam MP, Ardinger HH, et al., editors. GeneReviewsÂő [Internet]. Seattle (WA): University of Washington, Seattle; 1993-2015. Available from: `http://www.ncbi.nlm.nih.gov/books/NBK1224/`.

[23] M Copeland, A Brown, H Parkinson, R Stevens, and J Malone. The SWO Project: A Case Study of Applying Agile Ontology Engineering Methods in Community Driven Ontologies. *ICBO 2012*, 7 2012.

[24] M Courtot, F Gibson, A. L Lister, J Malone, D Schober, R. R Brinkman, and A Ruttenberg. Mireot: The minimum information to reference an external ontology term. *Appl. Ontol.*, 6(1):23–33, January 2011.

[25] M Crimi and R Rigolio. The mitochondrial genome, a growing interest inside an organelle. *International journal of nanomedicine*, 3(1):51–57, 2008.

[26] S Croset. *Drug repositioning and indication discovery using description logics*. PhD thesis, University of Cambridge, 2014.

[27] J Day-Richter, M. A Harris, M Haendel, and S Lewis. Obo-edit – an ontology editor for biologists. *Bioinformatics*, 23(16):2198–2200, August 2007.

[28] R de Almeida Falbo, G Guizzardi, A Gangemi, and V Presutti. Ontology patterns: Clarifying concepts and terminology. In *Proceedings of the 4th Workshop on Ontology and Semantic Web Patterns co-located with 12th International Semantic Web Conference (ISWC 2013), Sydney, Australia, October 21, 2013.*, 2013.

[29] S DiMauro and M Hirano. MELAS. 2001 Feb 27 [Updated 2013 Nov 21]. In: Pagon RA, Adam MP, Ardinger HH, et al., editors. GeneReviewsÂő [Internet]. Seattle (WA): University of Washington, Seattle; 1993-2015. Available from: `http://www.ncbi.nlm.nih.gov/books/NBK1233/`.

[30] M. D Donaldson, E. J Gault, K. W Tan, and D. B Dunger. Optimising management in Turner syndrome: from infancy to adult transfer. *Arch. Dis. Child.*, 91(6):513–520, Jun 2006.

[31] N Drummond, A Rector, R Stevens, G Moulton, M Horridge, H. H Wang, and J Seidenberg. Putting OWL in order: Patterns for sequences in OWL. *Concrete*, pages 1–10, 2006.

[32] M Duerst and M Suignard. Internationalized resource identifiers (iris). `http://tools.ietf.org/html/rfc3987`, 2005. Accessed: June 2015.

[33] M Dumontier, C Baker, J Baran, A Callahan, L Chepelev, J. C Toledo, N Del Rio, G Duck, L Furlong, N Keath, D Klassen, J McCusker, N. Q Rosinach, M Samwald, N. V Rosales, M Wilkinson, and R Hoehndorf. The Semanticscience Integrated Ontology (SIO) for biomedical research and knowledge discovery. *Journal of Biomedical Semantics*, 5(1):14+, 2014.

[34] EBI Web Team. ununbium atom (CHEBI:33517). `http://www.ebi.ac.uk/chebi/chebiOntology.do?chebiId=CHEBI:33517`. Accessed: June 2015.

[35] M Egaña, A Rector, R Stevens, and E Antezana. Applying ontology design patterns in bio-ontologies. In *Proceedings of the 16th International Conference on Knowledge Engineering: Practice and Patterns*, EKAW '08, pages 7–16, Berlin, Heidelberg, 2008. Springer-Verlag.

[36] M Egana Aranguren, R Stevens, and E Antezana. Transforming the axiomisation of ontologies: The ontology pre-processor language. *Nature Precedings*, Dec 2009.

[37] K Eilbeck, S Lewis, C Mungall, M Yandell, L Stein, R Durbin, and M Ashburner. The sequence ontology: a tool for the unification of genome annotations. *Genome Biology*, 6(5):R44, 2005.

[38] C Ford, K Jones, P Polani, J. D Almeida, and J Briggs. A SEX-CHROMOSOME ANOMALY IN A CASE OF GONADAL DYSGENESIS (TURNER'S SYNDROME). *The Lancet*, 273(7075):711 – 713, 1959. Originally published as Volume 1, Issue 7075.

[39] T Gabaldón. Computational approaches for the prediction of protein function in the mitochondrion. *American journal of physiology. Cell physiology*, 291(6):C1121–C1128, 2006.

[40] E Gamma, R Helm, R Johnson, and J Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[41] F Gandon and G Schreiber. RDF 1.1 XML Syntax. Technical report, 2014.

[42] A Garcia, K O'Neill, L. J Garcia, P Lord, R Stevens, O Corcho, and F Gibson. Developing ontologies within decentralised settings. In H Chen, Y Wang, and K.-H Cheung, editors, *Semantic e-Science*, volume 11 of *Annals of Information Systems*, pages 99–139. Springer US, 2010.

[43] C Goble and C Wroe. The Montagues and the Capulets. *Comp. Funct. Genomics*, 5(8):623–632, 2004.

[44] A Gómez-Pérez, M Fernández-López, and O Corcho. *Ontological Engineering*, volume 36. Springer, 2004.

[45] R. S Gonçalves, B Parsia, and U Sattler. Ecco: A Hybrid Diff Tool for OWL 2 ontologies. In *OWLED 2012*, 2012.

[46] B. M Good, E. M Tranfield, P. C Tan, M Shehata, G. K Singhera, J Gosselink, E. B Okon, and M. D Wilkinson. Fast, cheap and out of control: a zero curation model for ontology development. *Pacific Symposium On Biocomputing*, 11:128–139, 2006.

[47] Y Goto, I Nonaka, and S Horai. A mutation in the tRNA(Leu)(UUR) gene associated with the MELAS subgroup of mitochondrial encephalomyopathies. *Nature*, 348(6302):651–653, Dec 1990.

[48] J Gross and D Bhattacharya. Endosymbiont or host: who drove mitochondrial and plastid evolution? *Biology direct*, 6:12, 2011.

[49] T. R Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

[50] J Hallinan, K James, and A Wipat. Network approaches to the functional analysis of microbial proteins. In R. K Poole, editor, *Advances in Microbial Physiology*, volume 59 of *Advances in Microbial Physiology*, pages 101 – 133. Academic Press, 2011.

[51] H. A Hashim and A A. The Spiral_OWL Model – Towards Spiral Knowledge Engineering. 4(2):759 – 764, 2010.

[52] J Hastings, P de Matos, A Dekker, M Ennis, B Harsha, N Kale, V Muthukrishnan, G Owen, S Turner, M Williams, and C Steinbeck. The ChEBI reference database and ontology for biologically relevant chemistry: enhancements for 2013. *Nucleic Acids Research*, 41(D1):D456–D463, 2013.

[53] T Hayashi, R Rizzuto, G Hajnoczky, and T. P Su. Mam: more than just a housekeeper. *Trends in Cell Biology*, 19(2):81–88, 2009.

[54] J. M Herrmann and W Neupert. Protein transport into mitochondria. *Current Opinion in Microbiology*, 3(2):210–214, 2000.

[55] R Hoehndorf. What is an upper level ontology? `http://ontogenesis.knowledgeblog.org/740`, 2010. Accessed: June 2015.

[56] M Horridge, N Drummond, J Goodwin, A Rector, R Stevens, and H Wang. The Manchester OWL Syntax. In *OWLED 2006 Second Workshop on OWL Experiences and Directions*, Athens, GA, USA, 2006.

[57] M Horridge and P. F Patel-Schneider. Manchester syntax for OWL 1.1. *OWLED 2008*, 2008.

[58] M Horridge and P. F Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax (Second Edition). Technical report, 2012.

[59] I Horrock. Oxford ontology research overview. `http://dream.inf.ed.ac.uk/events/ukont-13/2012workshop/Horrocks_UK-Ont-Net.pdf`, 2012. Accessed: June 2015.

[60] L. J Jensen and P Bork. Ontologies in quantitative biology: A basis for comparison, integration, and discovery. *PLoS Biol*, 8(5):e1000374, 05 2010.

[61] R. E Jensen, C. D Dunn, M. J Youngman, and H Sesaki. Mitochondrial building blocks. *Trends in Cell Biology*, 14(5):215–218, 2004.

[62] S Jupp, M Horridge, L Iannone, J Klein, S Owen, J Schanstra, R Stevens, and K Wolstencroft. Populous: A tool for populating ontology templates. *Proceedings of the 3rd International Workshop on Semantic Web Applications and Tools for the Life Sciences BerlinGermany December 810 2010*, 2010.

[63] T Just. *U.S. Chess Federation's official rules of chess*. Random House, New York, 2014.

[64] R. D King, J Rowland, W Aubrey, M Liakata, M Markham, L. N Soldatova, K. E Whelan, A Clare, M Young, A Sparkes, S. G Oliver, and P Pir. The Robot Scientist Adam. *Computer*, 42(8):46–54, 2009.

[65] I Laycock. Modelling the proteins of the mitochondria using ontological modelling. Master's thesis, Newcastle University, 2010.

[66] J LEJEUNE, J LAFOURCADE, R BERGER, J VIALATTE, M BOESWILLWALD, P SERINGE, and R TURPIN. [3 CASES OF PARTIAL DELETION OF THE SHORT ARM OF A 5 CHROMOSOME]. *C. R. Hebd. Seances Acad. Sci.*, 257:3098–3102, Nov 1963.

[67] E. J Lesnefsky, S Moghaddas, B Tandler, J Kerner, and C. L Hoppel. Mitochondrial dysfunction in cardiac disease: ischemia–reperfusion, aging, and heart failure. *Journal of molecular and cellular cardiology*, 33(6):1065–1089, 2001.

[68] C Linnaeus. *Systema naturae sive Regna tria naturae systematice proposita per classes, ordines, genera, & species*. 1735.

[69] P. W Lord, R. D Stevens, A Brass, and C. A Goble. Investigating semantic similarity measures across the Gene Ontology: the relationship between sequence and annotation. *Bioinformatics*, 19(10):1275–83, 2003.

[70] P Lord. Components of an ontology. `http://ontogenesis.knowledgeblog.org/514`, 2010. Accessed: June 2015.

[71] P Lord. Omn-mode now released. `http://www.russet.org.uk/blog/2185`, 2012. Accessed: June 2015.

[72] P Lord. Programming OWL. `http://www.russet.org.uk/blog/2214`, 2012. Accessed: June 2015.

[73] P Lord. Some updates to omn-mode. `http://www.russet.org.uk/blog/2200`, 2012. Accessed: June 2015.

[74] P Lord. The Semantic Web takes Wing: Programming Ontologies with Tawny-OWL. *OWLED 2013*, 2013.

[75] P Lord. Further experiments with literate programming. http://www.russet.org.uk/blog/2979, 2014. Accessed: June 2015.

[76] P Lord. Manchester syntax is a bit backward. http://www.russet.org.uk/blog/2985, 2014. Accessed: June 2015.

[77] P Lord. Tawny and protege. http://www.russet.org.uk/blog/2981, 2014. Accessed: June 2015.

[78] P Lord and R Stevens. Adding a little reality to building ontologies for biology. *PLoS ONE*, 5(9):7, 2010.

[79] J Malone, E Holloway, T Adamusiak, M Kapushesky, J Zheng, N Kolesnikov, A Zhukova, A Brazma, and H Parkinson. Modeling Sample Variables with an Experimental Factor Ontology. *Bioinformatics (Oxford, England)*, March 2010.

[80] J Malone and H Parkinson. Reference and application ontologies. http://ontogenesis.knowledgeblog.org/295, 2010. Accessed: June 2015.

[81] J Malone, R Stevens, A Brown, and H Parkinson. An Agile Ontology. http://softwareontology.wordpress.com/2011/04/04/an-agile-ontology, 2011. Accessed: June 2015.

[82] J Malone, R Stevens, A Brown, and H Parkinson. The Software Ontology's Bottom Up, Agile Approach to Ontology building. http://softwareontology.wordpress.com/2011/08/08/267/, 2011. Accessed: June 2015.

[83] C Mammucari and R Rizzuto. Signaling pathways in mitochondrial dysfunction and aging. *Mechanisms of Ageing and Development*, 131(7-8):536–543, 2010.

[84] C. a Mannella. Structure and dynamics of the mitochondrial inner membrane cristae. *Biochimica et biophysica acta*, 1763(5-6):542–8, 2006.

[85] N Manwaring, M. M Jones, J. J Wang, E Rochtchina, C Howard, P Mitchell, and C. M Sue. Population prevalence of the MELAS A3243G mutation. *Mitochondrion*, 7(3):230–233, May 2007.

[86] V Marx. Biology: The big challenges of big data. *Nature*, 498(7453):255–260, 2013.

[87] D. L McGuinness and F van Harmelen. OWL Web Ontology Language Overview. Technical report, 2004.

[88] E Mikroyannidi, N Azlinayati, A Manaf, L Iannone, and R Stevens. Analysing syntactic regularities in ontologies. 2012.

[89] E Mikroyannidi, A Rector, and R Stevens. *Abstracting and Generalising the Foundational Model Anatomy (FMA) Ontology.* Bio-ontologies, 2009.

[90] M. H Moghadam, A Movafagh, M Omrani, K Ghanati, M Hashemi, F Poursafavi, H Darvish, D. Z Abdolahi, M Gholami, M. R. H Rostamy, S Safari, L HaghNejad, R Darehgazani, N. S Naeini, M. G Motlagh, and D Amani. Identification of homogeneously staining regions in leukemia patients. *Journal of Research in Medical Sciences*, 18(4), 2013.

[91] T Morgan. Turner syndrome: diagnosis and management. *Am Fam Physician*, 76(3):405–410, Aug 2007.

[92] W. W Morgan and P. C Keenan. Spectral classification. *Annual Review of Astronomy and Astrophysics*, 11(1):29–50, 1973.

[93] J. M Mortensen, M Horridge, M. A Musen, and N. F Noy. Applications of ontology design patterns in biomedical ontologies. *AMIA Annu Symp Proc*, 2012:643–652, 2012.

[94] J. M Mortensen, M Horridge, M. A Musen, and N. F Noy. Applications of ontology design patterns in biomedical ontologies. *AMIA Annu Symp Proc*, 2012:643–652, 2012.

[95] J Mortensen, M Horridge, M. A Musen, and N. F Noy. Modest use of ontology design patterns in a repository of biomedical ontologies. In *WOP'12*, pages –1–1, 2012.

[96] B Motik, B Parsia, and P. F Patel-Schneider. OWL 2 Web Ontology Language XML Serialization (Second Edition). Technical report, 2012.

[97] B Motik, P. F Patel-Schneider, and B Parsia. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). Technical report, 2012.

[98] C Mungall. The perils of managing OWL in a version control system. `https://douroucouli.wordpress.com/2014/03/30/the-perils-of-managing-owl-in-a-version-control-system/`, 2014. Accessed: June 2015.

[99] C Mungall, A Ruttenberg, I Horrocks, and D Osumi-Sutherland. OBO Flat File Format 1.4 Syntax and Semantics [DRAFT]. `http://oboformat.googlecode.com/svn/branches/2011-11-29/doc/obo-syntax.html`. Accessed: June 2015.

[100] C Mungall, A Ruttenberg, and D Osumi-Sutherland. Taking shortcuts with OWL using safe macros. *Nature Preceedings*, 2010.

[101] N. F Noy, N. H Shah, P. L Whetzel, B Dai, M Dorf, N Griffith, C Jonquet, D. L Rubin, M. A Storey, C. G Chute, and M. A Musen. BioPortal: ontologies and integrated data resources at the click of a mouse. *Nucleic Acids Res.*, 37(Web Server issue):W170–173, Jul 2009.

[102] N. F Noy and D. L mcguinness. Ontology Development 101: A Guide to Creating Your First Ontology. Online, 2001.

[103] N Noy and A Rector. Defining N-ary Relations on the Semantic Web. Technical report, 2006.

[104] T. W O'Brien. Properties of human mitochondrial ribosomes. *IUBMB Life*, 55(9):505–513, Sep 2003.

[105] M. J O'Connor, C Halaschek-Wiener, and M. A Musen. Mapping Master: A Flexible Approach for Mapping Spreadsheets to OWL. In P. F Patel-Schneider, Y Pan, P Hitzler, P Mika, L Zhang, J Pan, I Horrocks, and B Glimm, editors, *The Semantic Web – ISWC 2010*, volume 6497 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin Heidelberg, 2010.

[106] C Ogbuji. InfixOWL: An Idiomatic Interface for OWL. In C Dolbear, A Ruttenberg, and U Sattler, editors, *OWLED 2008*, volume 432 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[107] H. D Osiewacz. Regulation of the mitochondrial transition pore: impact on mammalian aging. *Aging (Albany NY)*, 3(1):10–11, Jan 2011.

[108] C Osman, D. R Voelker, and T Langer. Making heads or tails of phospholipids in mitochondria. *The Journal of cell biology*, 192(1):7–16, 2011.

[109] G O'Toole. Everything should be made as simple as possible, but not simpler – quote investigator. `http://quoteinvestigator.com/2011/05/13/einstein-simple/`, 2011. Accessed: June 2015.

[110] L. J Pallanck. Culling sick mitochondria from the herd. *The Journal of cell biology*, 191(7):1225–1227, 2010.

[111] L Partridge. Some highlights of research on aging with invertebrates, 2010. *Aging Cell*, 10(1):5–9, 2011.

[112] A Paschke. OntoMaven: Maven-based Ontology Development and Management of Distributed Ontology Repositories. *9th International Workshop on Semantic Web Enabled Software Engineering (SWESE 2013)*, 2013.

[113] P. F Patel-Schneider and I Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax Section 2. Abstract Syntax. Technical report, 2004.

[114] S. G Pavlakis, P. C Phillips, S DiMauro, D. C De Vivo, and L. P Rowland. Mitochondrial myopathy, encephalopathy, lactic acidosis, and strokelike episodes: a distinctive clinical syndrome. *Ann. Neurol.*, 16(4):481–488, Oct 1984.

[115] S Peroni, D Shotton, and F Vitali. The live owl documentation environment: A tool for the automatic generation of ontology documentation. In A ten Teije, J VÃűlker, S Handschuh, H Stuckenschmidt, M d'Aquin, A Nikolov, N Aussenac-Gilles, and N Hernandez, editors, *EKAW*, volume 7603 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2012.

[116] J Perry, S Nouri, P La, A Daniel, Z Wu, S Purvis-Smith, E Northrop, K Choo, and H Slater. Molecular distinction between true centric fission and pericentric duplication-fission. *Human Genetics*, 116(4):300–310, 2005.

[117] P Pizzo and T Pozzan. Mitochondria-endoplasmic reticulum choreography: structure and signaling dynamics. *Trends in Cell Biology*, 17(10):511–517, 2007.

[118] M Poveda-Villalón, M. C Suárez-Figueroa, and A Gómez-Pérez. Validating ontologies with oops! In *Proceedings of the 18th International Conference on Knowledge Engineering and Knowledge Management*, EKAW'12, pages 267–281, Berlin, Heidelberg, 2012. Springer-Verlag.

[119] V Presutti, E Daga, A Gangemi, and A Salvati. http: //ontologydesignpatterns.org [odp]. In C Bizer and A Joshi, editors, *International Semantic Web Conference (Posters & Demos)*, volume 401 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[120] S Rawat and T. L Stemmler. Key players and their role during mitochondrial iron-sulfur cluster biosynthesis. *Chemistry - A European Journal*, 17(3):746–753, 2011.

[121] A Rector. Representing Specified Values in OWL: value partitions and value sets. Technical report, 2005.

[122] A Rector, N Drummond, M Horridge, J Rogers, H Knublauch, R Stevens, H Wang, and C Wroe. OWL pizzas: Practical experience of teaching OWL-DL: Common errors & common patterns. In *In Proc. of EKAW 2004*, pages 63–81. Springer, 2004.

[123] A. S Reichert and W Neupert. Mitochondriomics or what makes us breathe. *Trends in Genetics*, 20(11):555–562, 2004.

[124] R Reja, A. J Venkatakrishnan, J Lee, B.-C Kim, J.-W Ryu, S Gong, J Bhak, and D Park. Mitointeractome: mitochondrial protein interactome database, and its application in âĂIJaging networkâĂİ analysis. *BMC genomics*, 10 Suppl 3:S20, 2009.

[125] R. J Roberts. Pubmed central: The genbank of the published literature. *Proceedings of the National Academy of Sciences of the United States of America*, 98(2):381–382, 2001.

[126] P Rocca-Serra, A Ruttenberg, M. J O'Connor, P. L Whetzel, D Schober, J Greenbaum, M Courtot, R. R Brinkman, S. A Sansone, R Scheuermann, R Scheuermann, and B Peters. Overcoming the ontology enrichment bottleneck with Quick Term Templates. *Appl. Ontol.*, 6(1):13–22, January 2011.

[127] C Rosse and J. L. V Mejino, Jr. A reference ontology for biomedical informatics: The foundational model of anatomy. *J. of Biomedical Informatics*, 36(6):478–500, December 2003.

[128] C Roussey, F Pinet, M Kang, and O Corcho. An introduction to ontologies and ontology engineering. In *Ontologies in Urban Development Projects*, volume 1 of *Advanced Information and Knowledge Processing*, pages 9–38. Springer London, 2011.

[129] E. J Ruiz, B. C Grau, I Horrocks, and R Berlanga. Building ontologies collaboratively using contentcvs. In *Proceedings of the 22nd International Workshop on Description Logics (DL 2009)*, 2009.

[130] U Sattler and R Stevens. Modelling in multiple dimensions is great in so many ways. `http://ontogenesis.knowledgeblog.org/1401`, 2013. Accessed: June 2015.

[131] U Sattler, R Stevens, and P Lord. (i can't get no) satisfiability. `http://ontogenesis.knowledgeblog.org/1329`, 2013. Accessed: June 2015.

[132] U Sattler, R Stevens, and P Lord. How does a reasoner work? `http://ontogenesis.knowledgeblog.org/1486`, 2014. Accessed: June 2015.

[133] A. H. V Schapira. Mitochondrial disease. *Lancet*, 368(9529):70–82, 2006.

[134] W. L Scoville. Note on capsicums. *Journal of the American Pharmaceutical Association*, 1(5):453–454, 1912.

[135] L Shaffer, M.-J J., and S M., editors. *ISCN 2013: An International System for Human Cytogenetic Nomenclature (2013)*. Karger, 2012.

[136] L Shaffer, M Slovak, and L Campbell, editors. *ISCN 2009: An International System for Human Cytogenetic Nomenclature (2009)*. Karger, 2009.

[137] R Shearer, B Motik, and I Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In A Ruttenberg, U Sattler, and C Dolbear, editors, *Proc. of the 5th Int. Workshop on OWL: Experiences and Directions (OWLED 2008 EU)*, Karlsruhe, Germany, October 26–27 2008.

[138] T. B Sherer, R Betarbet, and J. T Greenamyre. Environment, mitochondria, and parkinson's disease. *The Neuroscientist : a review journal bringing neurobiology, neurology and psychiatry*, 8(3):192–197, 2002.

[139] A. C Smith and A. J Robinson. MitoMiner, an integrated database for the storage and analysis of mitochondrial proteomics data. *Molecular cellular proteomics*, 8(6):1324–1337, 2009.

[140] M Smith, I Horrocks, M Krötzsch, and B Glimm. OWL 2 Web Ontology Language Conformance (Second Edition). Technical report, 2012.

[141] D. C Stein, F van Harmelen, I Horrocks, D. L McGuinness, P. F Patel-Schneider, and L. A Stein. DAML+OIL (March 2001) Reference Description. Technical report, 2001.

[142] R Stevens. Closing down the open world: Covering axioms and closure axioms. `http://ontogenesis.knowledgeblog.org/1001`, 2011.

[143] R Stevens. Why use an ontology? `http://ontogenesis.knowledgeblog.org/1296`, 2013. Accessed: June 2015.

[144] R Stevens, C. A Goble, and S Bechhofer. Ontology-based knowledge representation for bioinformatics. *Briefings in Bioinformatics*, 1(4):398–414, 2000.

[145] R Stevens and P Lord. Application of ontologies in bioinformatics. In S Staab and R Studer, editors, *Handbook on Ontologies in Information Systems*. Springer, second edition, 2008.

[146] K Stochholm, S Juul, K Juel, R. W Naeraa, and C HÃÿjbjerg Gravholt. Prevalence, incidence, diagnostic delay, and mortality in turner syndrome. *The Journal of Clinical Endocrinology & Metabolism*, 91(10):3897–3902, 2006. PMID: 16849410.

[147] M. C Suárez-Figueroa, A Gómez-Pérez, and M Fernández-López. The neon methodology for ontology engineering. In M. C Suárez-Figueroa, A Gómez-Pérez, E Motta, and A Gangemi, editors, *Ontology Engineering in a Networked World*, pages 9–34. Springer Berlin Heidelberg, 2012.

[148] R. W Taylor and D. M Turnbull. Mitochondrial dna mutations in human disease. *Nature reviews. Genetics*, 6(5):389–402, 2005.

[149] TIOBE. Tiobe software: The coding standards company. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Accessed: September 2014.

[150] S. H Tirmizi, S Aitken, D. A Moreira, C Mungall, J Sequeda, N. H Shah, and D. P Miranker. Mapping between the OBO and OWL ontology languages. *J Biomed Semantics*, 2 Suppl 1:S3, 2011.

[151] A Torroni, A Achilli, V Macaulay, M Richards, and H. J Bandelt. Harvesting the fruit of the human mtdna tree. *Trends in Genetics*, 22(6):339–345, 2006.

[152] A Trifunovic, A Hansson, A Wredenberg, A. T Rovio, E Dufour, I Khvorostov, J. N Spelbrink, R Wibom, H. T Jacobs, and N.-G Larsson. Somatic mtdna mutations cause aging phenotypes without affecting reactive oxygen species production. *Proceedings of the National Academy of Sciences of the United States of America*, 102(50):17993–17998, 2005.

[153] T Tudorache, C Nyulas, N. F Noy, and M. A Musen. WebProtégé: A Collaborative Ontology Editor and Knowledge Acquisition Tool for the Web. *Semant Web*, 4(1):89–99, Jan 2013.

[154] H. H TURNER. A syndrome of infantilism, congenital webbed neck, and cubitus valgus. *Endocrinology*, 23(5):566–574, 1938.

[155] S Varier, M Kaiser, and R Forsyth. Establishing, versus maintaining, brain function: A neuro-computational model of cortical reorganization after injury to the immature brain. *Journal of the International Neuropsychological Society*, 17(06):1030–1038, 2011.

[156] V Vassiliadis, J Wielemaker, and C Mungall. Processing OWL2 ontologies using Thea: An application of logic programming. In *OWLED 2009*, 2009.

[157] M Vigo, C Jay, and R Stevens. Constructing conceptual knowledge artefacts: Activity patterns in the ontology authoring process. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, pages 3385–3394, New York, NY, USA, 2015. ACM.

[158] J Visootsak and J. M Graham. Klinefelter syndrome and other sex chromosomal aneuploidies. *Orphanet journal of rare diseases*, 1:42, 2006.

[159] D Vrandečić and A Gangemi. Unit tests for ontologies. In R Meersman, Z Tari, and P Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4278 of *Lecture Notes in Computer Science*, pages 1012–1020. Springer Berlin Heidelberg, 2006.

[160] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition). Technical report, 2012.

[161] P Warren. Ontology users' survey - summary of results. Technical report, The Open University, June 2013.

[162] P Warren. Ontology patterns: a survey into their use. Technical report, The Open University, March 2014.

[163] J. D Warrender and P Lord. A pattern-driven approach to biomedical ontology engineering. *SWAT4LS 2013*, 2013.

[164] B Westermann and W Neupert. 'Omics' of the mitochondrion. *Nat. Biotechnol.*, 21(3):239–240, Mar 2003.

[165] M. M Wintrobe and J. P Greer. *Wintrobe's clinical hematology*, volume 1. 2009.

[166] K Wolstencroft, P Lord, L Tabernero, A Brass, and R Stevens. Protein classification using ontology classification. *Bioinformatics*, 22(14):e530–e538, 2006.

[167] K Wolstencroft, R Mcentire, R Stevens, L Tabernero, and A Brass. Constructing ontology-driven protein family databases. *Bioinformatics*, 21(8):1685–1692, April 2005.

[168] K. J Wolstencroft, R Stevens, L Tabernero, and A Brass. PhosphaBase: an ontology-driven database resource for protein phosphatases. *Proteins*, 58(2):290–294, Feb 2005.

[169] K Wolstencroft, S Owen, M Horridge, O Krebs, W Mueller, J. L Snoep, F du Preez, and C Goble. RightField: embedding ontology annotation in spreadsheets. *Bioinformatics*, 27(14):2021–2022, 2011.

[170] Yevgeny Kazakov and Markus Krötzsch and František Simančík. ELK: a reasoner for OWL EL ontologies. System description, University of Oxford, 2012. Available from `http://code.google.com/p/elk-reasoner/wiki/Publications`.

[171] M Zeviani and S Di Donato. Mitochondrial disorders. *Brain : a journal of neurology*, 127(Pt 10):2153–2172, 2004.