

Dissertation

Semantic Processing of Digital Documents

Christian Schönberg

`mail@cschoenberg.de`

`http://uri.cschoenberg.de?contact`

Submitted to the Faculty for Informatics and Mathematics, University of Passau.

13th November 2013

1st EXAMINER: Prof. Dr. Burkard Freitag
2nd EXAMINER: Prof. Dr. Dietmar Seipel

Abstract In this thesis we present a novel approach to modelling and processing digital documents. In contrast to other modelling approaches, we model the structure of documents as indicated by the content, not as defined by technical attributes like the file format. Additionally, our meta-model can be applied to a wide range of different documents, not just to a small set of documents with a predefined set of features. The models include semantic data and content relationships, which can be further extended with domain knowledge. All of this makes our document models immediately suitable for a wide range of applications, including document consistency verification and knowledge extraction.

Our new separation of technical and semantic document models fuels a standardised method for obtaining semantic models. This method is effective, suitable for live processing, and easily transferable to other document types and other domains. As it makes extensive use of background knowledge, we also present techniques for obtaining such knowledge, and for representing complex forms of knowledge with multiple meta-layers.

A flexible technique for obtaining relevant data from our document models completes the approach. This includes the ability to obtain various verification models, suitable for different types of consistency criteria and for different validation formalisms.

We conclude this thesis with an evaluation that shows the viability and effectiveness of the proposed approach. We present runtime results that are adequate for live processing, and we provide and successfully apply techniques for measuring the quality of both document models and background knowledge.

Acknowledgements

I would like to thank my supervisor Prof. Burkhard Freitag for many fruitful discussions, for valuable criticisms, and for his support whenever it was needed;

Prof. Harald Kosch and Prof. Mario Döllner for sharing their expertise about multimedia documents and hypervideos;

Prof. Ursula Reutner for insightful discussions about what Wikipedia can be (and what it cannot be);

Prof. Manfred Hinz for his insights about what documents can be (if we let them);

Prof. Rüdiger Harnisch for sharing his knowledge about language taxonomies;

Franz Weitzl and Mirjana Jakšić for many interesting discussions and for the collaboration on multiple papers;

Stephan Kiemle, Sabine Engelbrecht, and Eberhard Mikusch of the German Aerospace Center (DLR) for access to their most sacred documents and for their help with figuring them out;

Gerard de Melo of the Max Planck Institute for Informatics for a valuable discussion about Wikipedia's category structure;

Ulrike Sattler of the University of Rostock for granting me access to a large corpus of documents;

my friends and colleagues for many helpful discussions, for their proofreading, and for their understanding of my lack of spare time;

and last but not least my parents for their unconditional support.

The work for this thesis was partially funded by grants by the Deutsche Forschungsgemeinschaft (DFG) under grant numbers FR 1021/7-1 and -2.

Contents

I Preliminaries	3
1 Introduction	5
1.1 Motivation	5
1.2 Problem Description	6
1.3 Approach	7
1.4 Contributions	7
2 Other Work	9
2.1 Documents	9
2.1.1 Document Types	9
2.1.2 Digital Video	10
2.2 Document Modelling and Verification	10
II Foundations	13
3 Technologies	15
3.1 XML Technologies	15
3.1.1 XML	15
3.1.2 XPath	18
3.1.3 XQuery	20
3.2 Description Logics	21
3.2.1 Syntax of Description Logics	22
3.2.2 Semantics of Description Logics	25
3.2.3 Description Logic Languages	30
3.3 Semantic Web Technologies	33
3.3.1 RDF	33
3.3.2 RDF Schema	37
3.3.3 OWL	40
3.3.4 SKOS	43
3.3.5 SPARQL	45
3.3.6 RDF Frameworks	49
3.4 Rule Languages	50
3.5 Model Checking	51
3.5.1 CTL	51
3.5.2 $\mathcal{ALC}CTL$	54

4	Modelling Digital Documents	57
4.1	Terminology	57
4.1.1	What is a Document?	57
4.1.2	Our Notion of a Document	61
4.1.3	Interactivity in Documents	65
4.1.4	Other Types of Documents	66
4.1.5	Layers of Abstraction on Documents	67
4.1.6	Formalised Notion of a Document	71
4.2	Modelling Semantic Data	76
4.2.1	Modelling Documents	76
4.2.2	Modelling Processes	92
5	Processing Digital Documents	97
5.1	Extracting Data Models	97
5.2	Inference on Document Models	117
5.3	Modelling Towers of Meta	119
6	Background Knowledge	131
6.1	Obtaining Knowledge from Wikipedia	132
6.1.1	Preparing Wikipedia for Knowledge Extraction	134
6.1.2	Harvesting Wikipedia	143
6.1.3	Practical Considerations	146
6.2	Obtaining Knowledge from other Sources	149
III	Implementation	153
7	System Architecture	155
7.1	System Overview	155
7.2	Preprocessing	163
7.3	Semantic Processing	163
7.4	Postprocessing	165
8	Implementing Document Models	171
8.1	Implementation Basics	171
8.2	Implementing Document Fragments	172
8.3	Implementing Relations	172
8.3.1	Implementing Include Relations	172
8.3.2	Implementing Reference Relations	173
8.3.3	Implementing Has-Part Relations	174
8.3.4	Implementing Literal-Valued Relations	175
8.4	Document Lifecycle	177
8.5	Implementing Process Models	178
9	Processing Document Models	179
9.1	Extracting Data Models	179
9.1.1	Program-based Extraction	180
9.1.2	Query-based Extraction	183
9.1.3	Rule-based Extraction	186
9.2	Inference on Document Models	215

<i>CONTENTS</i>	III
9.3 Views on a Document Model	219
10 Use Cases	229
10.1 Document Verification	229
10.1.1 Realistic Technical Documentation Use Case	229
10.1.2 Real E-Learning Use Case	234
10.1.3 Real Lecture Notes Use Case	236
10.2 Process Verification	236
10.3 Other Use Cases	241
IV Evaluation	243
11 Quantitative Evaluation	245
11.1 Generated Documents	245
11.2 Real Documents from Different Domains	249
11.2.1 E-Learning Documents	250
11.2.2 Technical Documentation	251
11.2.3 Process Descriptions (NPB)	252
11.2.4 Process Descriptions (DLR)	253
11.3 Comparison with Alternative Methods	256
11.4 Adaptability of the Processing Rules	263
11.5 Inference on Large Ontologies	264
12 Qualitative Evaluation	271
12.1 Expressive Power of Transformation Rules and Background Knowledge	271
12.2 Quality of the Document Models	273
12.3 Quality of the Background Knowledge	275
12.4 Relationship between Qualities	276
12.5 Applicability to Different Types of Documents	277
V Conclusion	281
13 Conclusion	283
14 Extensions and Future Work	285
Bibliography	287
List of Figures	297
List of Tables	301
Index	304
VI Appendix	309
A Model Vocabulary	311
A.1 Common Vocabulary	311

A.1.1	Object Properties	311
A.1.2	Datatype Properties	312
A.1.3	Classes	318
A.2	Vocabulary for Documents	319
A.2.1	Object Properties	319
A.2.2	Datatype Properties	319
A.2.3	Classes	319
A.3	Vocabulary for Processes	322
A.3.1	Object Properties	322
A.3.2	Datatype Properties	322
A.3.3	Classes	323
B	Use Cases	325
B.1	Technical documentations in DocBook format	325
B.2	Process specifications in BPMN format (NPB)	332
B.3	Process specifications in Visio format (NPB)	335
B.4	Process specifications in XML format (NPB)	338
B.5	Process specifications in Word format (DLR)	342
B.6	Game of Life	350

Thesis Outline

This thesis is divided into six parts. The first part holds the introduction and an overview of the relevant literature.

Part II deals with the foundations required for this work, starting with the technologies that are used. This is followed by a discussion on the nature of documents and how they can be modelled. The next chapter contains information about document processing, how to obtain a document model from a source document, how to augment this model with additional domain knowledge, and how to use it. Finally, in chapter 6, various methods for obtaining background knowledge are discussed.

The third part describes the implementation of the proposed approach, starting with an overview of the system architecture. Then follows a chapter on how to implement document models, succeeded by a chapter on how to implement document processing. Part III is concluded by a discussion of several use cases for the proposed approach.

The evaluation is contained in part IV, including a quantitative evaluation and a qualitative evaluation.

Part V concludes this theses, with a conclusion, a chapter on future work, and the bibliography.

Finally, the sixth and final part holds the appendix.

Part I

Preliminaries

Chapter 1

Introduction

In this thesis, we will concern ourselves with digital documents. We will explore the limits of what is and what is not a document. We will then attempt to find a metamodel for documents that is suitable for different documents, for different domains, and for different applications. We will also try to find a way to obtain instances of such a metamodel that is effective, efficient and transferable between different types of documents. Finally, we will investigate techniques on how to augment an existing model with additional information, and on how to derive from this general model more specialised models that are suitable for specific tasks.

1.1 Motivation

The verification of consistency criteria for documents is an increasingly common [Jel02, ABF04, Wei08] and very useful application. Yet for formal validation techniques, equally formalised document models are required, which are not always easy to obtain. Especially for a multitude of documents in different file formats and from different domains, it is both complex and expensive to obtain document models that meet specific requirements. This problem becomes increasingly worse for a growing number of different formats, for a growing number of different domains, and also for a growing number of different target applications such as different verification formalisms that require different verification models. But even for a single target formalism more than a single verification model per document is often necessary, as we will explore in more detail in section 9.3.

As an example that we will revisit throughout this thesis, consider an e-learning document with five chapters. The first chapter is an introduction. It is followed by a chapter (chapter 2) that contains the definition of a “data structure”. From this chapter, two other chapters can be reached: chapter 3 that contains an example of a “data structure”, and chapter 4 that contains both the definition and an illustration of a “binary tree”. Both chapters 3 and 4 then reference the final chapter 5, which contains a conclusion.

Now imagine a consistency criterion stating that after every definition, there must always be an example with the same topic. At first glance, this criterion is violated by the example document because after the “data structure” definition in chapter 2, the reader can follow up with chapter 4 which does not contain an appropriate example. At second glance, however, this perception changes. Chapter 4 does indeed not contain an example with the topic “data structure”, but it does contain an illustration of a “binary tree”. It can be argued that a “binary tree” is a specific “data structure”, and that an “illustration” is a specialisation of an “example”. Therefore, with additional knowledge about both the content and the structure of a document

infused into a document model, the verification process can not only be made less complex but also more precise.

It is also possible that the document's reader has previous knowledge, for example from required reading that is a prerequisite for this document. The contents of these prerequisite documents can be formalised and be made available to the verification process, thereby improving its fact base. Previous knowledge about binary trees might also serve to defuse the criteria violation explained above.

An abstract document model that contains all relevant information from the original source document can solve these issues. There is no longer a need for defining transformation instructions from every document format into every target format. Instead, a single set of transformation instructions must be defined for each format, and a standardised way to access this model allows the easy derivation of more specialised models for specific tasks. The standardisation of the target format (for the extraction from source documents) and the standardisation of the source format (for the generation of task-specific models) also allows for a stronger standardisation of the respective transformation processes, which makes them more easily transferable to new formats, domains or applications. In addition, further processing of such a distinct document model allows its enhancement with new facts that are part of the domain knowledge but that are not explicitly incorporated in the source document.

Next to document verification, another application for a document model that contains only the most relevant information from a source document, but that may also be augmented with further domain knowledge is information retrieval. With such models, it is not only possible to search for documents that contain specific keywords, but it is also possible to extend the search to semantically related keywords, to structurally limit the search to keywords that occur in the same document part, or to limit the results to the relevant parts of a document. It is also possible to judge the relevance of a document for a keyword query based on the prominence of the keyword usage: a keyword that is used several times in an obscure sub-section might be less relevant than a keyword that is only used once but in a top-level headline.

1.2 Problem Description

To the best of our knowledge, there exists no model for documents that goes beyond a simple reproduction of content and references and that is at the same time independent of the document's format and domain. We will attempt to create such a model. The first question that we will have to answer is what exactly a document is. In the computer science literature, it is usually implied that the term is either well-understood or self-evident. Yet most publications have a different understanding of documents, ranging from small data records over single files to complex structures spanning multiple files. After answered this, we must then identify properties that a document must have (or must *not* have), in order to be viable for generic modelling.

A model that is useful as a go-between for different document formats and different domains on the one hand side, and different applications on the other, must contain all pertinent information from the source document as well as relevant domain knowledge. This includes information about the content and the structure of the document, but also about relationships within the content and the structure, such as related terms or structural hierarchies. A metamodel must be created that takes these requirements into account.

We will also need to define standardised methods for obtaining an instance of such a metamodel, as well as for processing a model further. The latter includes deriving other, more specialised models. These methods may employ techniques known from the fields of information extraction and information retrieval, but we will not attempt to create new extraction, text mining, or

retrieval techniques.

1.3 Approach

We will first attempt to identify a set of features or attributes that are common to documents that can be used for verification or similar applications. We will then try to develop a technical understanding of documents, which we will juxtapose with a more abstract understanding that is based on the structure and content semantics of a document. Two metamodels will be created that can be used to represent documents in each form, the technical or the semantic form, respectively. We will also define a formal semantics for the semantic metamodel, which will be required for further processing of semantic models.

Based on these metamodels we then plan to define a technique to obtain instances of the latter metamodel from instances of the former, i.e., to create semantic models from technical models. Similarly, we plan to define a technique for obtaining other types of models from the created semantic models.

We will implement the proposed approach to test and analyse its viability, its effectiveness and also the runtime cost of the implementation for real-world tasks. For our evaluation, we will use, among others, e-learning documents obtained from a national research and industry program; technical documentations created for several open source projects; process descriptions specified by the German Aerospace Center; and workflow specifications created by various public institutions. Finally, we will attempt to find ways to measure the quality of the obtained models based on standard information extraction techniques like precision and recall.

1.4 Contributions

Our main contributions are the following:

1. Consolidation and formalisation of the notion of a document. This will be done both empirically and under consideration of the former and current discourse in the domains of computer science and literary science.
2. Creation of a coherent, multi-purpose metamodel for documents. Instances of this metamodel can represent the semantics of the content as well as the structure as interpreted by a reader.
 - ▶ Creation of a description logics-based implementation and formal semantics of such a model.
 - ▶ Development of a technique for obtaining such models using domain knowledge, with a focus on maintainability in the face of changed to the document format, and transferability to other formats and domains.
 - ▶ Transferring the approach from documents to process specifications.
3. Development of an approach for automated retrieval of domain knowledge from Wikipedia, complementing existing techniques, and design of a process for obtaining such knowledge semi-automatically from other sources.
4. Introduction of measures for the quality of document models.

Chapter 2

Other Work

Documents are a topic of interest in many domains and applications, and are frequently the subject of research. In this chapter, we will provide an overview of the primary research areas. We will insert additional literature reviews in appropriate places throughout this thesis.

2.1 Documents

First, we will summarise some of the most important document types and formats for text-centred documents, followed by a summary of video formats.

2.1.1 Document Types

The *Hypertext Markup Language (HTML)* [HTMa] is the standard document format for web documents. In combination with *Cascading Style Sheets (CSS)* and *Javascript*, HTML is a powerful specification format (see below).

A standard for (mostly) static and non-changeable documents is Adobe's *Portable Document Format (PDF)* [PDF]. It is often used as a publishing format, but it is rarely the native file format of any document.

For electronic books, the *ePub* format [EPU] and Amazon's Kindle format are very common. Both have capabilities that are a subset of HTML and CSS. But HTML is also often used directly as an e-book format, as is PDF.

Another large host of documents is encoded in office formats, such as Microsoft's *Word*, *Powerpoint*, *Excel*, or *Visio* formats, as well as their free counterparts.

XML-based open source formats like *DocBook* [Wal09] or *DITA* [DEAJ10] are popular for documentations, in particular among the open source community. Documents in either format are rarely published directly in their source format, but are rather compiled to PDF or HTML.

L^AT_EX is a standard format for small and large texts in academia, often used for research papers, lecture notes, and even books. It is compiled into a target document format, usually PDF.

Two sophisticated e-learning formats are *LMML* [SF99] and *ML3* [TLV03]. Both offer support for a broad range of content and can be exported to HTML and (in the case of LMML) PDF.

2.1.2 Digital Video

There are many different technical representation schemas for digital video: there are different container formats that can hold the actual encoded video or audio data like *Audio Video Interleave (AVI)*, *Matroska* or *Quicktime*. There are even more encoding/decoding formats for the video data like *MPEG-2*, *MPEG-4*, *Theora* or *Flash Video*.

In addition, meta formats like *MPEG-7* [Kos04] allow the codification of video metadata. This includes low level descriptors like image colour, texture, or object shapes and motions. It also includes high level semantic descriptors like information about persons and locations shown in the video, as well as information about relationships between depicted entities. [Got06]

Several digital video formats provide support for user interaction. The *Synchronized Multimedia Integration Language (SMIL)* [SMI] allows the specification of time-synchronised multimedia content like audio, video, text, or images, similar to the proprietary Adobe Flash format. SMIL can be scripted, which includes reacting to simple user inputs. It can also interact with web services to facilitate complex behaviour such as database access.

HTML5 [HTMb] documents use multiple standards: the content, such as text, images, video and audio, can be specified in HTML, CSS can be used to define layout and visualisation, and Javascript can be used for interaction. Interaction options include navigation, playback control of audio and video content (e.g., pause or adjust volume), and dynamically displaying additional content (e.g., textual descriptions for specific video sequences). HTML5 allows splitting video content into sequential sections to ease navigation, especially to improve barrier-free access.

SVG (Scalable Vector Graphics) [SVG] is a vector-based image format that supports animation both natively and through Javascript. Through Javascript, it also supports interaction. SVG documents can be embedded in SMIL and in HTML5 documents.

The *SIVA (Simple Interactive Video Authoring Suite)* [MMGK12] tool and format allows the specification of hypervideos including a table of contents for video sequences, multiple choice elements (i.e., navigation based on user answers), video hotspots (e.g., objects with additional information that can be displayed on user interaction), and a keyword-based search. An HTML5-based implementation exists for the hypervideo model.

2.2 Document Modelling and Verification

There are several approaches to document verification that are based on the XML DOM model of the source documents.

Schematron, developed by [Jel02], allows for XPath-based verification. The advantage of this approach is that it only relies on readily available, standardised technologies. The disadvantage is that the entire process is slaved to the file format, i.e., for different file formats – possibly even just for different files – the components need to be re-implemented.

A similar method is employed by [NCEF02] in the *xlinkit* tool, which is also closely enmeshed with the source document’s XML structure. The authors use XPath to find inconsistencies in the data, then generate links to either “proof” (witnesses) for consistency or to counter-examples that show inconsistencies.

The Verdi approach [ABF04] is a single-purpose method for verifying web documents. It is restricted to a single document format and the underlying rule-based specification language for consistency criteria does not support order criteria such as “before”, “after”, or “until”.

Also based on the XML-structure of documents, [Sch04] uses full first order logics for verifying the consistency of collections of interrelated documents

Other approaches focus on modelling the structure of documents, sometimes also allowing

for verification. Most of these publications followed in the wake of the advent of hypertext as a serious theoretical and practical concept.

Coming from the field of discourse analysis, [vDK83] attempt to capture the structure of discourse in so-called “discourse superstructures”. They are defined as grammars, but their focus is mainly on the cognitive aspects of how the structure is captured “on the fly” by a reader.

[Gar87] defines a model for hypertexts that is based on a graph-view. This graph is represented as a set of edges between document nodes.

Furuta and Stotts provide several models for hypertexts. One is based on Petri nets, which allows for checks of simple pre-conditions when traversing a hypertext [SF89]. They also developed a metamodel that describes different layers of abstraction for hypertexts, but they focus less on the content and more on the broad underlying structure and navigation [FS90]. In [SFC98], they model hypertexts as automata, but again limited to the structure [SFC98].

[HS94] presents a graph-based hypertext model that focusses on the technical aspects of hypertext. It is an early attempt to capture the general properties of hypertexts.

Van der Aalst et. al. use Petri nets to represent formal structures, in particular processes and workflows. Their approach is limited to the control flow (i.e., the structure), however, ignoring the data flow [vdA03, OVvdA⁺07].

A third type of approaches puts the primary emphasis on the document’s content, to the detriment of the structure.

[ESS05] describe a verification technique based on description logics. It is built on top of Schema¹, which treats all documents as collections of XML fragments that can be re-arranged into new documents. Its focus is on dissecting and re-combining document content instead of the document’s structure [Gru06].

The Text Encoding Initiative [TEI07] has developed a set of guidelines and tools for encoding the texts of humanity. Their XML-based structural metamodel offers no generalisations, but instead a large number of very concrete types for the classification of document elements. The metamodel is therefore very broad, but also very flat and provides little abstraction. Its focus lies on the layout and on the content, less on the structure. While it covers many domains, it lacks an underlying semantic formalism.

Finally, there are approaches to document modelling that attempt to represent the document layout as faithfully as possible, or to render it as effectively as possible.

The group around David Brailsford has proposed a document layout model defined by distinct atomic objects that are arranged on a page [BBH03]. They also developed a model consisting of multiple pre-typeset document fragments that can be used for the dynamic display and layout of e-books [PBB11].

[PCC⁺11] suggest a physics-based approach to semi-automatically layout documents, supporting document creators. It uses graph-layout techniques applied to atomic document elements.

The focus of current document modelling or verification techniques is either on the content or on the structure. In the latter case, the approach is either format-driven, i.e., slaved to the schema of the document format, or captures only very broad structural elements like “file” instead of smaller and more specific elements like “paragraph” or “definition”. A combination of these different emphases is still missing.

¹<http://www.schema.de>

Part II

Foundations

Chapter 3

Technologies

In this chapter, we will discuss several basic formalisms and technologies that we will use in the course of these theses.

3.1 XML Technologies

In this section, we will give a brief introduction into three XML technologies used in these theses. For further details, see [BPSM⁺08, CD99, BCF⁺10].

3.1.1 XML

XML, the *extensible markup language*, is a semi-structured markup language and a W3C recommendation since early 1998. It is based on unicode, allowing for international letters in both content and markup [BPSM⁺08].

The three primary building blocks of XML are elements, attributes, and text. An element consists of an opening and a closing tag with identical names. It may have a number of attributes and it may have content that consists of text, XML elements, or a mixture of both. The opening tag is indicated by `<...>`, and the closing tag is indicated by `</...>`. An attribute consists of a name and a textual value, written as `name="value"`. While an element may have several child elements with identical names, each of its attributes must have a different name.

To avoid confusion between XML elements that have the same name but a different intended meaning, XML supports *namespaces*. Assigning XML elements to a unique namespace removes the ambiguity between elements from different namespaces. A namespace is represented by its name, which must be a URI such as `http://purl.org/dc/terms/`.

XML elements in a namespace are identified by their complete name, for example `http://purl.org/dc/terms/subject`.

By defining a shorthand prefix for a namespace, elements can be written as for example `<dc:subject xmlns:dc="http://purl.org/dc/terms/">`. In addition, a default namespace can be declared that encompasses all XML elements below and including the element where it was defined, for example `<subject xmlns="http://purl.org/dc/terms/">`.

Attributes may have a namespace as well, but it must be used explicitly because the default namespace is disregarded in attribute definitions.

Example 3.1.1 (XML). *The following XML fragment models basic data about two dances, the slow waltz and the Viennese waltz. It lists some figures for the dances, each with a name that*

is unique within the scope of that dance. The description of a figure may include references to figures that can be danced after this figure and a textual description.

```

1 <dances xmlns="http://www.dancedescriptions.net/">
2   <dance>
3     <name>Slow Waltz</name>
4     <figure name="Natural Turn">
5       <follow-with>Natural Turn</follow-with>
6       <follow-with>Closed Change 1</follow-with>
7       <description>
8         The basic figure of the waltz is a turn
9         to the right, as opposed to the
10        <reference>Reverse Turn</reference>, which
11        turns to the left.
12      </description>
13    </figure>
14    <figure name="Reverse Turn">
15      <follow-with>Reverse Turn</follow-with>
16      <follow-with>Closed Change 2</follow-with>
17    </figure>
18    <figure name="Closed Change 1">
19      <follow-with>Reverse Turn</follow-with>
20      <follow-with>Closed Change 2</follow-with>
21      <description>
22        The closed change is the simplest segue between the
23        <reference>Natural Turn</reference> and the
24        <reference>Reverse Turn</reference>.
25      </description>
26    </figure>
27    <figure name="Closed Change 2">
28      <follow-with>Natural Turn</follow-with>
29      <follow-with>Closed Change 1</follow-with>
30      <description>
31        The closed change is the simplest segue between the
32        <reference>Natural Turn</reference> and the
33        <reference>Reverse Turn</reference>.
34      </description>
35    </figure>
36  </dance>
37  <dance>
38    <name>Viennese Waltz</name>
39    <figure name="Natural Turn">
40      <description>
41        The steps of the basic figure of the Viennese waltz
42        are similar to the steps of the natural turn in
43        the slow waltz.
44      </description>
45    </figure>
46  </dance>
47 </dances>

```

An XML document is *well-formed* if it only contains legal characters, if all opening and closing element names match, if element names contain only certain characters (basically alphanumeric letters, dash, underscore, dot, and colon), and if the elements form a tree-structure with a distinct root element. XML parsers have to reject documents that are not well-formed.

Other XML building blocks are comments, indicated by `<!-- -->`, processing instructions, and entities. Processing instructions contain information for parsers or other processors, such as encoding information. They are indicated by `<? ?>`. Entities act like macros that are resolved

by the XML parser, which replaces the macro with its definition. Character entities such as `@` refer to the unicode character with the given number. Named entities such as `'` refer to a string.

As a convenience for using XML markup characters in text and attribute values, the entities `'`, `"`, `&`, `<`, and `>` for `'`, `"`, `&`, `<`, and `>` are predefined in XML. Named entities can be defined in a DTD (see below).

Example 3.1.2 (XML (continued)). *The following XML code adds XML version and encoding information to a fragment from example 3.1.1. It also shows the use of comments and entities.*

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <dances xmlns="http://www.dancedescriptions.net/">
3   <dance>
4     <name>Slow Waltz</name>
5     <figure name="Natural Turn">
6       <!-- add quotation to the text -->
7       <description>
8         The basic figure of the &quot;waltz&quot; is a turn
9         to the &right;, as opposed to the
10        <reference>Reverse Turn</reference>, which
11        turns to the &left;.
12      </description>
13    </figure>
14  </dance>
15 </dances>

```

A Document Type Definition Document Type Definition, or *DTD*, is a schema definition for an XML document. In a DTD, element and attribute names can be defined, including where they may be used and what values they may have. In addition, named entities can be defined.

`<!ELEMENT figure (follow-with+, description?)>` is a declaration that defines an element named “figure”, which may have one or more (+) `<follow-with>` child elements and an optional (?) `<description>` child element. A star (*) indicates zero or more child elements, and `#PCDATA` represents text.

`<!ATTLIST figure name CDATA #REQUIRED>` is a declaration that defines an attribute “name” for the `<figure>` element with a textual value (CDATA). The attribute is required (`#REQUIRED`), not optional (`#IMPLIED`).

`<!ENTITY right ‘‘right’’>` defines a named entity with the value “right”. It can be referenced with `&right;`. The content of external files can be referenced with `<ENTITY external SYSTEM ‘‘filename.xml’’>`.

A document type definition can be written to an external file and referenced from within an XML document with `<!DOCTYPE dances SYSTEM ‘‘dances.dtd’’>` before the root element. A DTD can also be embedded into an XML document with `<!DOCTYPE dances [...]>`.

Example 3.1.3 (DTD). *The following lists a DTD schema for the XML document from example 3.1.1. It makes use of the default namespace defined there in line 1.*

```

1 <!ELEMENT dances (dance*)>
2 <!ELEMENT dance (name, figure*)>
3 <!ELEMENT name (#PCDATA)>
4 <!ELEMENT figure (follow-with+, description?)>
5 <!ATTLIST figure name CDATA #REQUIRED>
6 <!ELEMENT follow-with (#PCDATA)>
7 <!ELEMENT description (#PCDATA | reference)*>
8 <!ELEMENT reference (#PCDATA)>

```

```

9 <!ENTITY right "right">
10 <!ENTITY left "left">

```

XML Schema is another language for specifying XML schema definition. It is more powerful than DTD, but also far more complex and verbose. For further information on XML Schema, we refer the reader to [TBMM04, BM04b].

In addition to well-formedness, XML parsers can check an XML document against a schema and check if it satisfies the requirements specified there. If an XML document can be successfully verified against a schema, it is called *valid* with respect to that schema.

There are two primary types of XML parsers, SAX and DOM parsers. *SAX* (*Simple API for XML*) parsers linearly scan a document and provide handlers whenever an XML building block such as an opening element, a comment, a string, or a closing element is encountered. They are fast and require few resources, but they leave the work of assembling a model for the parsed data to the handler.

DOM (*Document Object Model*) parsers read an entire XML file into a memory model that closely represents the XML structure. They provide a high level of comfort for the user, but at the cost of high resource consumption.

Since Java 1.4, Java provides a *Java API for XML Processing* (JAXP) that includes interfaces for both SAX and DOM. There are multiple implementations for this API, including one from Oracle that is bundled with Java, and one from the Apache Project called Xerces.

XML is used in many different ways, for example as a document format in DocBook, various office formats, and XHTML. It is also used to specify graphics (SVG) or annotated video (MPEG7), and for data interchange (SOAP).

3.1.2 XPath

XPath is a selector language on XML, recommended by the W3C in late 1999 [CD99]. For path selectors in general, and how they are evaluated, see also definitions 5.2.2 and 5.2.4.

XPath expressions are a sequence of path expressions, separated by /, or the union of two XPath expressions, separated by |. A path expression consists of an optional axis expression, a path match, and an optional condition.

An axis expression changes the scope of the following path match:

<code>ancestor</code>	applies the path match to the transitive closure of parent elements of the current element,
<code>ancestor-or-self</code>	includes the current element with the ancestor elements,
<code>attribute</code>	applies the path match to all attributes of the current element,
<code>child</code>	applies the path match to all child elements for the current element (this is the default behaviour when no axis is defined),
<code>descendant</code>	applies the path match to the transitive closure of child elements of the current element,
<code>descendant-or-self</code>	includes the current element with the descendant elements,
<code>following</code>	applies the path match to all elements that come after the current element in document order,
<code>following-sibling</code>	applies the path match to all following elements on the same level in the tree structure and with the same parent element as the current element,
<code>namespace</code>	applies the path match to all namespace definitions of the current element,
<code>parent</code>	applies the path match to the parent element of the current element,
<code>preceding</code>	applies the path match to all elements that come before the current element in document order,
<code>preceding-sibling</code>	applies the match to all preceding elements on the same level in the tree structure and with the same parent element as the current element, and
<code>self</code>	applies the match only to the current element.

Axis expressions always apply the path match to elements starting at the current element and moving away from it. For example, the first element on the `ancestor` axis is the current element's parent, and the last element is the document's root element. Axis expressions are syntactically separated from path matches by `::`.

Path matches can match an element or an attribute by name (`element`, `@attribute`), they can match the root element (`/`), they can match the current element (`.`), and they can match the current element's parent (`..`). They can also match any element, attribute, text node or simply any node using wildcards (`*`, `@*`, `text()`, `node()`).

Conditions are expressions with a Boolean value, such as constants, function calls, equality or inequality assertions, or Boolean relations on other expressions. Syntactically, they are enclosed in `[]`.

Example 3.1.4 (XPath). *The following XPath expressions select nodes and values from the XML document from example 3.1.1.*

Select the names of all figures:

```
/descendant-or-self::figure/@name
```

Select the names of all figures, using the shorthand // for descendant-or-self:

```
//figure/@name
```

Select the names of all figures in the slow waltz:

```
//dance[name = 'Slow Waltz']/figure/@name
```

Select all `<description>` elements that contain `<reference>` elements:

```
//description[.//reference]
```

Select the first `<follow-with>` element of each figure:

```
//figure/follow-with[not(preceding-sibling::follow-with)]
```

XPath defines several functions, including numerical functions like `abs()` and `round()`, string functions like `concat()`, `upper-case()`, `substring()`, and `translate()` (which replaces occurrences of one character in a string with another character), node and node sequence functions such as `name()` (which returns the name of a node), `position()` (which returns the position of a node within a node sequence), and `last()` (which returns the last node of a node sequence), as well as aggregation functions like `max()` and `sum()`.

Example 3.1.5 (XPath (continued)). *The following XPath expressions select nodes and values from the XML document from example 3.1.1.*

Select all `<dance>` elements with more than one figure:

```
//dance[count(figure) > 1]
```

Select the names of all figures that end in “turn”:

```
//figure[ends-with(lower-case(@name), 'turn')]/@name
```

Select the first `<follow-with>` element of each figure:

```
//figure/follow-with[position() = 1]
```

Select the first `<follow-with>` element of each figure, using a shorthand:

```
//figure/follow-with[1]
```

3.1.3 XQuery

In contrast to XPath, *XQuery* is a Turing-complete XML query language, recommended by the W3C since early 2007 [BCF⁺10]. It does rely heavily on XPath, though. Conceptually, it is a functional programming language, but without support for higher-order functions.

It provides so-called FLWOR-expressions: FOR, LET, WHERE, ORDER BY, and RETURN. This is illustrated in example 3.1.6.

Example 3.1.6 (XQuery). *The following is an XQuery program that returns the names of all dances ordered alphabetically. The `doc()` function returns the DOM of an external file.*

```
1 declare default element namespace = "http://www.dancedescriptions.net/";
2 for   $dance in doc("dances.xml")//dance
3 let   $name := $dance/name
4 where count($dance/figure) > 0
5 order by $name
6 return $name
```

XQuery allows for the definition of custom functions, as illustrated in example 3.1.7. In addition to iteration with FOR, it supports recursion.

Example 3.1.7 (XQuery (continued)). *The following XQuery program returns an XML element sequence of named figures. For every figure, if it has a description, the text is included and the references are resolved. For each reference, both the name of the dance and the name of the figure are returned.*

```

1 declare default element namespace = "http://www.dancedescriptions.net/";
2 declare function local:getDescription($desc) {
3   for $node in $desc/node()
4     return if (local-name($node) = "reference")
5              then ($node/ancestor::dance/name/text(), " ", $node/text())
6              else $node
7 };
8 for $figure in doc("dances.xml")//figure
9 return
10  <figure name="{ $figure/@name }">
11    {
12      if ($figure/description)
13        then <desc>
14           { local:getDescription($figure/description) }
15         </desc>
16      else ()
17    }
18  </figure>

```

While there exist several XQuery processors, the number of freeware or open-source processors of decent quality is rather limited. We will make use of the GNU Qexo processor, which not only implements the XQuery language, but also provides the possibility for Java-callbacks from XQuery programs. The processor is written in Java, it can be accessed directly from Java code, and calls to Java methods can be integrated into the XQuery code. This is illustrated in example 3.1.8.

Example 3.1.8 (XQuery and Java). *The following XQuery program calls a Java method to spell-check descriptions. The namespace declaration syntax is used to reference Java classes.*

```

1 declare default element namespace = "http://www.dancedescriptions.net/";
2 declare namespace SpellChecker = "class:net.dancedescriptions.SpellChecker";
3 for $desc in doc("dances.xml")//description
4 return SpellChecker:check($desc)

```

3.2 Description Logics

The following section is based in large parts on [BCM⁺03] and [Fre11], extended with state of the art developments and some proofs not contained in the original research. We will give a compact overview, using a syntax and terminology that we unified across the literature.

We will begin with a language-independent discussion of the syntax of descriptions logics, followed by a discussion of its semantics. We will then introduce several description logic languages of varying expressiveness, and conclude with a brief discussion of decidability.

Description logics are commonly used in knowledge representation. They offer an expressive formalism combined with efficient reasoning services.

3.2.1 Syntax of Description Logics

The basic building blocks of description logics are individuals, concepts, and roles.

Definition 3.2.1 (Individual). *An individual is a distinct object in a domain. It is defined and identified solely by its name.*

Example 3.2.2 (Individual). *chapter2 and datastructure are individual names.*

Definition 3.2.3 (Atomic Concept). *An atomic concept represents a set of individuals. It is defined and identified solely by its name.*

Example 3.2.4 (Atomic Concept). *Chapter and Topic are atomic concept names.*

Definition 3.2.5 (Atomic Role). *An atomic role represents a binary relation on a set of individuals. It is defined and identified solely by its name.*

Example 3.2.6 (Atomic Role). *successor and hasTopic are atomic role names.*

Definition 3.2.7 (Complex Concept). *A complex concept represents a set of individuals. For an atomic concept A , an individual i , complex concepts C_1 and C_2 , a complex role R (see below), and a number $n \in \mathbb{N}_0$, the following constructs are complex concepts:*

\top	(top or universal concept),
\perp	(bottom concept),
A	(atomic concept),
$\{i\}$	(set constructor),
$\neg C_1$	(complement),
$C_1 \sqcap C_2$	(intersection),
$C_1 \sqcup C_2$	(union),
$\exists R.C_1$	(existential quantification),
$\forall R.C_1$	(universal quantification or value restriction),
$\geq nR.C_1$	(qualified number restriction),
$\leq nR.C_1$	(qualified number restriction),
$= nR.C_1$	(qualified number restriction), and
$\exists R.\nabla$	(self reference).

Remark 3.2.8. *Note that the set constructor is a convenient way to define a concept in an extensional way. Complement, intersection, union, quantification, number restriction and self reference are ways to define concepts in an intensional way.*

Example 3.2.9 (Complex Concept). *$A \sqcap B$, $\neg\text{Definition} \sqcup \text{Example}$, and $\exists\text{relatedTo}.\nabla$ are complex concepts.*

Definition 3.2.10 (Complex Role). *A complex role represents a binary relation on a set of individuals. For an atomic role A and complex roles R_1 and R_2 , the following constructs are complex roles:*

\top	(top or universal role),
\perp	(bottom role),
A	(atomic role),
$\neg R_1$	(complement),
$R_1 \sqcap R_2$	(intersection),
$R_1 \sqcup R_2$	(union),
R_1^+	(transitive closure),
R_1^-	(inverse), and
$R_1 \circ R_2$	(composition).

Example 3.2.11 (Complex Role). $relatedTo^+$, $hasTopic \sqcup hasSubject$, and $hasPart \circ hasTopic$ are complex roles.

Next to individuals, description logics can also deal with concrete data types.

Definition 3.2.12 (Concrete Role). Let \mathbf{D} be a set of concrete data types.

A concrete role represents a binary relation between a set of individuals and \mathbf{D} . It is defined and identified solely by its name.

Let R_{C_1} and R_{C_2} be concrete roles, $d \in \mathbf{D}$ a concrete data type, and R a complex role. Then concrete roles and data types can be used in the following constructs:

- $\neg d$ (data type complement),
- $\exists R_{C_1}.d$ (data type existence),
- $\forall R_{C_1}.d$ (data type restriction),
- $\geq nR_{C_1}.d$ (qualified number restriction),
- $\leq nR_{C_1}.d$ (qualified number restriction),
- $= nR_{C_1}.d$ (qualified number restriction),
- $\neg R_{C_1}$ (complement),
- $R_{C_1} \sqcap R_{C_2}$ (intersection),
- $R_{C_1} \sqcup R_{C_2}$ (union), and
- $R \circ R_{C_1}$ (role composition).

Note that $\neg d$ can be used anywhere in place of d .

Example 3.2.13 (Concrete Data Type). $\mathbb{N} \in \mathbf{D}$ is a concrete data type. age is a concrete role. $\forall age.\mathbb{N}$ is a complex concept using the concrete role.

Individuals, concepts, roles and concrete data types can be used in more complex statements to assert facts in a knowledge base.

Definition 3.2.14 (Terminological Axiom). A terminological axiom defines a relationship between two concepts or between two roles. For complex concepts C_1 and C_2 and for complex or concrete roles R_1 and R_2 , the following constructs are terminological axioms:

- $C_1 \equiv C_2$ (equality),
- $C_1 \sqsubseteq C_2$ (inclusion, subsumption),
- $R_1 \equiv R_2$ (equality), and
- $R_1 \sqsubseteq R_2$ (inclusion, subsumption).

Equality axioms with solely an atomic concept or an atomic role on the left hand side of the equality operator are called **definitions**. In a definition, the atomic concept is called the **symbolic name** for the complex concept on the other side of the equality operator.

In definitions, the symbol \equiv is often replaced with \doteq .

Inclusion axioms with solely an atomic concept or an atomic role on the left hand side of the subsumption operator are called **specialisations**.

Example 3.2.15 (Terminological Axiom). The following statements are terminological axioms:

- Illustration* \sqsubseteq *Example*,
- List* \equiv *OrderedList* \sqcup *UnorderedList*, and
- successor* \sqsubseteq *references*.

The following statements are definitions:

- List* \doteq *OrderedList* \sqcup *UnorderedList* and
- partOf* \doteq *hasPart*⁻.

Definition 3.2.16 (Instance Assertion). An *instance assertion* defines a relationship between an individual and a concept, or between two individuals and a role. For individuals i_1 and i_2 , a complex concept C , a complex role R , a value v belonging to a concrete data type $d \in \mathbf{D}$, and a concrete role R_C , the following constructs are instance assertions:

$$\begin{aligned} C(i_1) & \quad (\text{concept assertion}), \\ R(i_1, i_2) & \quad (\text{role assertion}), \text{ and} \\ R_C(i_1, v) & \quad (\text{value assertion}). \end{aligned}$$

Example 3.2.17 (Instance Assertion). $\text{Chapter}(\text{chapter2})$, $\text{Topic}(\text{datastructure})$, and $(\text{hasPart} \circ \text{hasTopic})(\text{chapter2}, \text{datastructure})$ are instance assertions.

Definition 3.2.18 (TBox). A TBox $\mathcal{T} = (C, R, X_T)$ specifies terminological information. It consists of a finite set of concepts C , a finite set of roles R , and a finite set of terminological axioms X_T using C and R that only contains definitions and specialisations, and that does not define any symbolic name more than once.

Example 3.2.19 (TBox). Let

$$\begin{aligned} C & = \{\text{Chapter}, \text{Topic}\} \\ R & = \{\text{hasPart}, \text{hasTopic}\} \\ X_T & = \{\text{Chapter} \sqcap \text{Topic} \sqsubseteq \perp, \\ & \quad \text{hasTransTopic} \doteq \text{hasPart}^+ \circ \text{hasTopic}\} \end{aligned}$$

Then $\mathcal{T} = (C, R, X_T)$ is a TBox.

Definition 3.2.20 (ABox). An ABox $\mathcal{A} = (I, X_I)$ specifies assertions about individuals. It consists of a finite set of domain objects I and a finite set of instance assertions X_I that use I and the terminology specified in a TBox.

Example 3.2.21 (ABox). Let

$$\begin{aligned} I & = \{\text{chapter2}, \text{paragraph2.1}, \text{datastructure}\} \\ X_I & = \{\text{Chapter}(\text{chapter2}), \\ & \quad \text{Topic}(\text{datastructure}), \\ & \quad \text{hasPart}(\text{chapter2}, \text{paragraph2.1}), \\ & \quad \text{hasTopic}(\text{paragraph2.1}, \text{datastructure})\} \end{aligned}$$

Then $\mathcal{A} = (I, X_I)$ is an ABox.

Definition 3.2.22 (Description Logic System). A *description logic system* is a knowledge base consisting of a TBox and an ABox that uses the TBox. It provides inference services (see below) on both the TBox and the ABox.

Definition 3.2.23 (Ontology). In the domain of knowledge management and informatics, an *ontology* is a knowledge base that defines a terminology for a domain and facts about a domain using the terminology.

In the context of this thesis, we will regard and use the term “ontology” as an equivalent to “description logic system”, even though in general there exist ontologies that are not based on description logics.

Formally, an ontology $\mathcal{O} = (C, R, I, X)$ contains a TBox $\mathcal{T} = (C, R, X_T)$ and an ABox $\mathcal{A} = (I, X_I)$, with $X = X_T \cup X_I$. \mathcal{T} defines the terminology that can be used to make assertions in \mathcal{A} .

Example 3.2.24 (Ontology). Let $\mathcal{T} = (C, R, X_T)$ be the TBox from example 3.2.19. Let $\mathcal{A} = (I, X_I)$ be the ABox from example 3.2.21.

Then $\mathcal{O} = (C, R, I, X)$ with $X = X_T \cup X_I$ is an ontology containing \mathcal{T} and \mathcal{A} .

A special type of ontology that defines a hierarchy on its individuals is called a *taxonomy*.

Definition 3.2.25 (Taxonomy). *An ontology $\mathcal{X} = (C, R, I, X)$ is a taxonomy, iff it contains at least one role $r \in R$ with $\nexists x \in I : r^+(x, x)$, where r^+ is the transitive closure of r .*

In other words, r defines an acyclic directed graph on I .

r typically models a relationship similar to a generalisation.

Example 3.2.26 (Taxonomy). *Let*

$$\begin{aligned} C &= \emptyset, \\ R &= \{\text{hasNarrower}\}, \\ I &= \{\text{mammal, bird, animal}\}, \text{ and} \\ X &= \{\text{hasNarrower}(\text{animal, mammal}), \text{hasNarrower}(\text{animal, bird})\}. \end{aligned}$$

Then $\mathcal{X} = (C, R, I, X)$ is a taxonomy.

Description 3.2.27 (Knowledge Base). *A knowledge base is an ontology or a taxonomy. Sometimes, a knowledge base is defined as an ontology $\mathcal{O} = (\emptyset, R, I, X)$ without concepts, also written as (R, I, X) .*

3.2.2 Semantics of Description Logics

Having defined the syntax of description logics, we will now discuss its semantics.

Definition 3.2.28 (Interpretation). *Let \mathcal{O} be an ontology.*

Then $I_{\mathcal{O}} = (\Delta^I, ()^I)$ is an interpretation of \mathcal{O} , where Δ^I is a non-empty set of objects called the domain of $I_{\mathcal{O}}$, and $()^I$ is a function in postfix superscript notation. Let Δ_D be the domain of all data types $d \in \mathbf{D}$, with $d^D \subseteq \Delta_D$ the domain of d .

Δ^I and Δ_D are assumed to be disjoint. Then $()^I$ assigns

1. *to every individual from \mathcal{O} a member of Δ^I ,*
2. *to every atomic concept from \mathcal{O} a subset of Δ^I ,*
3. *to every atomic role from \mathcal{O} a subset of $\Delta^I \times \Delta^I$,*
4. *to every concrete role from \mathcal{O} a subset of $\Delta^I \times \Delta_D$,*
5. *to every concrete data type $d \in \mathbf{D}$ a subset d^D of Δ_D , and*
6. *to every value v belonging to a concrete data type d a member of d^D .*

For complex concepts C_1 and C_2 , for complex roles R_1 and R_2 , for concrete roles R_{C_1} and R_{C_2} , for a concrete data type d , and for $n \in \mathbb{N}_0$, $()^I$ is inductively extended to complex concepts and roles as follows:

\top^I	$:= \Delta^I$
\perp^I	$:= \emptyset$
$(\neg C_1)^I$	$:= \Delta^I \setminus C_1^I$
$(C_1 \sqcap C_2)^I$	$:= C_1^I \cap C_2^I$
$(C_1 \sqcup C_2)^I$	$:= C_1^I \cup C_2^I$
$(\exists R_1.C_1)^I$	$:= \{x \in \Delta^I \mid \text{for all } y \in \Delta^I : (x, y) \in R_1^I \text{ implies } y \in C_1^I\}$
$(\forall R_1.C_1)^I$	$:= \{x \in \Delta^I \mid \text{there exists } y \in \Delta^I : (x, y) \in R_1^I \text{ and } y \in C_1^I\}$
$(\geq n R_1.C_1)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta^I \mid (x, y) \in R_1^I \text{ and } y \in C_1^I\} \geq n\}$
$(\leq n R_1.C_1)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta^I \mid (x, y) \in R_1^I \text{ and } y \in C_1^I\} \leq n\}$
$(= n R_1.C_1)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta^I \mid (x, y) \in R_1^I \text{ and } y \in C_1^I\} = n\}$
$(\exists R_1.\nabla)^I$	$:= \{x \in \Delta^I \mid (x, x) \in R_1^I\}$
$\top\top^I$	$:= \Delta^I \times \Delta^I$
$\perp\perp^I$	$:= \emptyset$
$(\neg R_1)^I$	$:= (\Delta^I \times \Delta^I) \setminus R_1^I$
$(R_1 \sqcap R_2)^I$	$:= R_1^I \cap R_2^I$
$(R_1 \sqcup R_2)^I$	$:= R_1^I \cup R_2^I$
$(R_1^+)^I$	$:= \text{transitive closure of } R_1^I$
$(R_1^-)^I$	$:= \{(y, x) \in \Delta^I \times \Delta^I \mid (x, y) \in R_1^I\}$
$(R_1 \circ R_2)^I$	$:= \{(x, z) \in \Delta^I \times \Delta^I \mid \text{there exists } y : (x, y) \in R_1^I \text{ and } (y, z) \in R_2^I\}$
d^I	$:= d^D$
$(\neg d)^I$	$:= \Delta_D \setminus d^I$
$(\exists R_{C_1}.d)^I$	$:= \{x \in \Delta^I \mid \text{for all } y \in \Delta_D : (x, y) \in R_{C_1}^I \text{ implies } y \in d^I\}$
$(\forall R_{C_1}.d)^I$	$:= \{x \in \Delta^I \mid \text{there exists } y \in \Delta_D : (x, y) \in R_{C_1}^I \text{ and } y \in d^I\}$
$(\geq n R_{C_1}.d)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta_D \mid (x, y) \in R_{C_1}^I \text{ and } y \in d^I\} \geq n\}$
$(\leq n R_{C_1}.d)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta_D \mid (x, y) \in R_{C_1}^I \text{ and } y \in d^I\} \leq n\}$
$(= n R_{C_1}.d)^I$	$:= \{x \in \Delta^I \mid \{y \in \Delta_D \mid (x, y) \in R_{C_1}^I \text{ and } y \in d^I\} = n\}$
$(\neg R_{C_1})^I$	$:= (\Delta^I \times \Delta_D) \setminus R_{C_1}^I$
$(R_{C_1} \sqcap R_{C_2})^I$	$:= R_{C_1}^I \cap R_{C_2}^I$
$(R_{C_1} \sqcup R_{C_2})^I$	$:= R_{C_1}^I \cup R_{C_2}^I$
$(R_1 \circ R_{C_1})^I$	$:= \{(x, z) \in \Delta^I \times \Delta_D \mid \text{there exists } y : (x, y) \in R_1^I \text{ and } (y, z) \in R_{C_1}^I\}$

Example 3.2.29 (Interpretation). Let $\mathcal{O} = (C, R, I, X_T \cup X_I)$ be the ontology from example 3.2.24.

Then $I_{\mathcal{O}} = (\Delta^I, ()^I)$ is an interpretation of \mathcal{O} , with

Δ^I	$= \{\text{Chapter 2, Paragraph 2.1, Data Structure}\}$
chapter2^I	$= \text{Chapter 2,}$
paragraph2.1^I	$= \text{Paragraph 2.1,}$
datastructure^I	$= \text{Data Structure}$
Chapter^I	$= \{\text{Chapter 2}\},$
Topic^I	$= \{\text{Data Structure}\}$
hasPart^I	$= \{(\text{Chapter 2, Paragraph 2.1})\},$
hasTopic^I	$= \{(\text{Paragraph 2.1, Data Structure})\}$

Description 3.2.30 (Unique Name Assumption). An interpretation $I_{\mathcal{O}}$ respects the unique name assumption iff

$$i_1 \neq i_2 \Rightarrow i_1^I \neq i_2^I$$

for individuals i_1 and i_2 , i.e., if distinct individual names must have distinct interpretations.

In the context of this thesis, we will assume that every interpretation respects the unique name assumption unless indicated otherwise.

Definition 3.2.31 (Model). *An axiom or assertion x can be valid in an interpretation $I_{\mathcal{O}}$, written as $I_{\mathcal{O}} \models x$.*

For individuals i_1 and i_2 , for complex concepts C_1 and C_2 , for complex or concrete roles R_1 and R_2 , for a complex role R , for a value v belonging to a concrete data type $d \in \mathbf{D}$, and for a concrete role R_C , this is defined as follows:

$$\begin{aligned} I_{\mathcal{O}} \models C_1 \sqsubseteq C_2 &\Leftrightarrow C_1^I \subseteq C_2^I \\ I_{\mathcal{O}} \models C_1 \equiv C_2 &\Leftrightarrow C_1^I = C_2^I \\ I_{\mathcal{O}} \models R_1 \sqsubseteq R_2 &\Leftrightarrow R_1^I \subseteq R_2^I \\ I_{\mathcal{O}} \models R_1 \equiv R_2 &\Leftrightarrow R_1^I = R_2^I \\ I_{\mathcal{O}} \models C_1(i_1) &\Leftrightarrow i_1^I \in C_1^I \\ I_{\mathcal{O}} \models R(i_1, i_2) &\Leftrightarrow (i_1^I, i_2^I) \in R^I \\ I_{\mathcal{O}} \models R_C(i_1, v) &\Leftrightarrow (i_1^I, v^I) \in R_C^I \end{aligned}$$

*An interpretation $I_{\mathcal{O}}$ is also called a **model** of an axiom or assertion x iff x is valid in $I_{\mathcal{O}}$.*

*An interpretation $I_{\mathcal{O}}$ is a **model** of an ontology $\mathcal{O} = (C, R, I, X)$ iff all axioms and assertions $x \in X$ are valid in $I_{\mathcal{O}}$, written as $I_{\mathcal{O}} \models \mathcal{O}$.*

Example 3.2.32 (Model). *Let $\mathcal{O} = (C, R, I, X_T \cup X_I)$ be the ontology from example 3.2.24. Then the interpretation $I_{\mathcal{O}} = (\Delta^I, ()^I)$ from example 3.2.29 is a model of \mathcal{O} .*

Proposition 3.2.33 (Concept Equivalences). *For complex concepts C_1 and C_2 , and a complex role R , the following equivalences hold in any model:*

$$\begin{aligned} C_1 \sqcup C_2 &\equiv \neg(\neg C_1 \sqcap \neg C_2) & (a) \\ \exists R.C_1 &\equiv \neg \forall R. \neg C_1 & (b) \end{aligned}$$

Proof of Proposition 3.2.33 (a). Let \mathcal{O} be an ontology containing the two complex concepts $C_1 \sqcup C_2$ and $\neg(\neg C_1 \sqcap \neg C_2)$, for complex concepts C_1 and C_2 . Let $I_{\mathcal{O}}$ be a model of \mathcal{O} . Then

$$\begin{aligned} (\neg(\neg C_1 \sqcap \neg C_2))^I &= \Delta^I \setminus (\neg C_1 \sqcap \neg C_2)^I \\ &= \Delta^I \setminus ((\neg C_1)^I \cap (\neg C_2)^I) \\ &= \Delta^I \setminus (\Delta^I \setminus C_1^I \cap \Delta^I \setminus C_2^I) \\ &= C_1^I \cup C_2^I \\ &= (C_1 \sqcup C_2)^I \end{aligned}$$

□

Proof of Proposition 3.2.33 (b). Let \mathcal{O} be an ontology containing the two complex concepts $\exists R.C$ and $\neg \forall R. \neg C$, for a complex concept C and a complex role R . Let $I_{\mathcal{O}}$ be a model of \mathcal{O} . Then

$$\begin{aligned} (\neg \forall R. \neg C)^I &= \Delta^I \setminus (\forall R. \neg C)^I \\ &= \Delta^I \setminus \{x \in \Delta^I \mid \text{exists } y \in \Delta^I : (x, y) \in R^I \text{ and } y \in (\neg C)^I\} \\ &= \Delta^I \setminus \{x \in \Delta^I \mid \text{exists } y \in \Delta^I : (x, y) \in R^I \text{ and } y \in \Delta^I \setminus C^I\} \\ &= \Delta^I \setminus \{x \in \Delta^I \mid \text{for all } y \in \Delta^I : (x, y) \notin R^I \text{ or } y \notin \Delta^I \setminus C^I\} \\ &= \Delta^I \setminus \{x \in \Delta^I \mid \text{for all } y \in \Delta^I : (x, y) \in R^I \text{ implies } y \in C^I\} \\ &= (\exists R.C)^I \end{aligned}$$

□

Definition 3.2.34 (Expansion). *Let $\mathcal{O}_D = (C, R, I, X_T \cup X_I)$ be an ontology that only contains definitions in X_T . Let \mathcal{O}'_D be the expansion of \mathcal{O}_D . \mathcal{O}'_D is obtained from \mathcal{O}_D as follows:*

- For every definition $x \in X_T$ of the form $A \doteq D$, where $A \in C \cup R$ is a symbolic concept or role name and D is a complex concept or role:
 - ▷ in every other concept or role axiom $y \in X_T \cup X_I$ where A is not the symbolic concept or role name:
 - replace all occurrences of A with D .

For ontologies without cyclic definitions, i.e., where A does not occur in the expansion of D , this process terminates after a finite number of steps.

For ontologies that also contain specialisations, a more complex inference procedure is required.

For inference tasks, it is often helpful to be able to disregard the TBox, i.e., to only regard an ABox $\mathcal{A} = (I, X_I)$ with respect to a TBox $\mathcal{T} = (C, R, \emptyset)$. This can be achieved by

1. expanding the TBox, and then
2. replacing all concept and role names in X_I with their expansion.

For an ontology, different *inference services* can be defined. For the TBox, a satisfiability check can identify “empty” concepts, and a subsumption check can identify hierarchical relationships between concepts. For the ABox, its consistency with a given TBox can be checked, and new assertions that are logical consequences of existing assertions and axioms can be inferred. For a TBox and an ABox, all individuals belonging to a concept can be retrieved, and the most specific concepts to which a given individual belongs can be found.

Definition 3.2.35 (Satisfiability). For a TBox \mathcal{T} , a concept or role C is *satisfiable with respect to \mathcal{T}* iff there exists a model $I_{\mathcal{T}}$ with $C^I \neq \emptyset$.

Example 3.2.36 (Satisfiability). For the TBox \mathcal{T} from example 3.2.19, all concepts and roles are satisfiable with respect to \mathcal{T} . The concept **ChapterAndTopic** \doteq **Chapter** \sqcap **Topic** is not satisfiable with respect to \mathcal{T} because **Chapter** and **Topic** are defined to be disjoint.

Definition 3.2.37 (Subsumption). For a TBox \mathcal{T} , a concept or role C_1 is *subsumed with respect to \mathcal{T}* by a concept or role C_2 iff $C_1^I \subseteq C_2^I$ for every model $I_{\mathcal{T}}$ of \mathcal{T} .

For a TBox \mathcal{T} , two concepts C_1 and C_2 are *equivalent with respect to \mathcal{T}* iff $C_1^I = C_2^I$ for every model $I_{\mathcal{T}}$ of \mathcal{T} .

Example 3.2.38 (Subsumption). For the TBox \mathcal{T} from example 3.2.19, the concept **ChapterAndTopic** \doteq **Chapter** \sqcap **Topic** is subsumed by the bottom concept with respect to \mathcal{T} : **ChapterAndTopic** \sqsubseteq \perp .

Definition 3.2.39 (Consistency). For a TBox \mathcal{T} , an ABox \mathcal{A} is *consistent with respect to \mathcal{T}* iff there exists a model $I_{\mathcal{T},\mathcal{A}}$.

Example 3.2.40 (Consistency). The ABox from example 3.2.21 is consistent with respect to the TBox from example 3.2.19, as shown by the model given in example 3.2.29.

Definition 3.2.41 (Instantiation). For a TBox \mathcal{T} and an ABox \mathcal{A} , a concept or role assertion x is a *logical consequence of \mathcal{T} and \mathcal{A}* iff $I_{\mathcal{T},\mathcal{A}} \models x$ for every model $I_{\mathcal{T},\mathcal{A}}$.

Example 3.2.42 (Instantiation). For the ontology \mathcal{O} from example 3.2.24, the role assertion **hasTransTopic**(chapter2, datastructure) is a logical consequence of \mathcal{O} .

Proof. To show: for every model I of \mathcal{O} :

$$I \models \text{hasTransTopic}(\text{chapter2}, \text{datastructure}).$$

Therefore, to show: for every model I of \mathcal{O} :

$$(\text{chapter2}^I, \text{datastructure}^I) \in \text{hasTransTopic}^I.$$

W.l.o.g., let I be any model of \mathcal{O} . The assertions and axioms

$$\begin{aligned} & \text{hasPart}(\text{chapter2}, \text{paragraph2.1}), \\ & \text{hasTopic}(\text{paragraph2.1}, \text{datastructure}), \text{ and} \\ & \text{hasTransTopic} \doteq \text{hasPart}^+ \circ \text{hasTopic} \end{aligned}$$

are all part of \mathcal{O} . Since $I \models \mathcal{O}$, the following must also be true:

$$\begin{aligned} & (\text{chapter2}^I, \text{paragraph2.1}^I) \in \text{hasPart}^I, \\ & (\text{paragraph2.1}^I, \text{datastructure}^I) \in \text{hasTopic}^I, \text{ and} \\ & \text{hasTransTopic}^I = (\text{hasPart}^+ \circ \text{hasTopic})^I. \end{aligned}$$

Resolving \circ leads to

$$\begin{aligned} & (\text{hasPart}^+ \circ \text{hasTopic})^I = \\ & \{(x, z) \mid \text{there exists } y : (x, y) \in (\text{hasPart}^+)^I \text{ and } (y, z) \in \text{hasTopic}^I\}, \end{aligned}$$

which contains $(\text{chapter2}^I, \text{datastructure}^I)$. □

Definition 3.2.43 (Individual Retrieval). *For an ontology \mathcal{O} , a model $I_{\mathcal{O}}$ of \mathcal{O} , and a concept C from \mathcal{O} , the individual retrieval problem is to find all individuals i in \mathcal{O} such that $I_{\mathcal{O}} \models C(i)$.*

Example 3.2.44 (Individual Retrieval). *For the ontology and model from example 3.2.29, the individual `chapter2` can be retrieved for the concept `Chapter`.*

Definition 3.2.45 (Concept Realisation). *For an ontology \mathcal{O} , a model $I_{\mathcal{O}}$ of \mathcal{O} , and an individual i , the concept realisation problem is to find the most specific concepts C in \mathcal{O} such that $I_{\mathcal{O}} \models C(i)$.*

Example 3.2.46 (Concept Realisation). *For the ontology and model from example 3.2.29, the concept `Chapter` can be found for the individual `chapter2`.*

In contrast to, for example, database systems that follow the *closed world assumption*, description logics adhere to the *open world assumption*.

Description 3.2.47 (Closed World Assumption). *Under the closed world assumption (CWA), the data represented in an information system is regarded as complete. It is assumed that any information that is not represented in the system has been explicitly omitted and is therefore missing for a reason. In other words, any datum that cannot be retrieved from an information system is regarded as logically false under the CWA.*

Example 3.2.48 (Closed World Assumption). *Let an information system contain only the single assertion that “The moon is made of rock”. Under the closed world assumption, we can infer that the moon is not made out of cheese, because this datum is not contained in the information system.*

Description 3.2.49 (Open World Assumption). *Under the open world assumption (OWA), the data represented in an information system is regarded as incomplete. It is assumed that any information that is not represented in the system may simply be missing, may have been overlooked, or will be inserted at a later time. In other words, only if a datum can be inferred to be logically false, can it be regarded as logically false under the OWA.*

Example 3.2.50 (Open World Assumption). *Let an information system contain only the single assertion that “The moon is made of rock”. Under the open world assumption, we can neither infer that the moon is made out of cheese, nor that it is not made out of cheese. The information system does not contain any indication one way or the other. Formally, there exist models for this information system in which the moon is made of cheese, and other models in which the moon is not made of cheese.*

If, however, the information system contains a more precise assertion that “The moon is made entirely of rock and nothing else”, then we can logically infer that the moon is not made of cheese, and there exist no models in which it is.

3.2.3 Description Logic Languages

There exist several description logics languages with varying degrees of expressiveness. As a base language with rudimentary capabilities, we will use the description logics language \mathcal{AL} .

Definition 3.2.51 (\mathcal{AL}). *The \mathcal{AL} description logics language supports the top and bottom concepts (\top and \perp), atomic concept negation ($\neg A$ for an atomic concept A), concept intersection ($C_1 \sqcap C_2$ for complex concepts C_1 and C_2), value restriction ($\forall R.C$ for an atomic role R and a complex concept C), restricted existential quantification ($\exists R.\top$ for an atomic role R), complex concept axioms ($C_1 \sqsubseteq C_2$, $C_1 \equiv C_2$ for complex concepts C_1 and C_2), and concept and role assertions ($C(i_1)$, $R(i_1, i_2)$ for an atomic concept C , an atomic role R , and individuals i_1 and i_2).*

Description 3.2.52 (\mathcal{AL} Language Extensions). *The following symbols indicate extensions for the \mathcal{AL} description logic language:*

- C : complex complement ($\neg C$ for a complex concept C)
- (\mathcal{D}) : concrete data types
- \mathcal{E} : full existential quantification ($\exists R.C$ for an atomic role R and a complex concept C). Recall that full existential quantification can be expressed using universal quantification and complex complement (cf. proposition 3.2.33)
- \mathcal{F} : functional roles ($\leq 1R.C$ for an atomic role R and a complex concept C)
- \mathcal{H} : role hierarchies ($R_1 \sqsubseteq R_2$, $R_1 \equiv R_2$ for complex roles R_1 and R_2)
- \mathcal{I} : inverse roles (R^- for an atomic role R)
- \mathcal{N} : number restrictions ($\leq nR.\top$, $\geq nR.\top$, $= nR.\top$ for an atomic role R and $n \in \mathbb{N}_0$)
- \mathcal{O} : nominals, set constructor ($\{i\}$ for an individual i)
- \mathcal{Q} : qualified number restrictions ($\leq nR.C$, $\geq nR.C$, $= nR.C$ for an atomic role R , a complex concept C , and $n \in \mathbb{N}$)
- \mathcal{R} : disjoint roles ($R_1 \sqcap R_2 \sqsubseteq \perp$), reflexive and irreflexive roles, negated role assertions ($\neg R(i_1, i_2)$), role inclusion axioms of the form $R_1 \circ R_2 \sqsubseteq R_1$ and $R_2 \circ R_1 \sqsubseteq R_1$, a universal role (\top), and a self concept ($\exists R_1.\top$), for atomic roles R_1 and R_2 , and individuals i_1 and i_2

- S*: shorthand for \mathcal{AL} with complex complement and transitive roles
(R^+ for an atomic role R)
- U*: concept union ($C_1 \sqcup C_2$ for two complex concepts C_1 and C_2).
Recall that concept union can be expressed using concept intersection
and complex complement (cf. proposition 3.2.33).

Proposition 3.2.53 (Reflexive and Irreflexive Roles). *For an atomic role R , reflexivity can be expressed as $\top \sqsubseteq \exists R.\nabla$ (a), and irreflexivity can be expressed as $\top \sqsubseteq \neg\exists R.\nabla$ (b).*

Proof of Proposition 3.2.53 (a). Let R be an atomic role and $I_{\mathcal{O}}$ a model for R .

$$\begin{aligned} R \text{ is reflexive} &\Leftrightarrow \forall x \in \Delta^I : (x, x) \in R^I \\ &\Leftrightarrow \Delta^I \subseteq \{x \in \Delta^I \mid (x, x) \in R^I\} \\ &\Leftrightarrow I_{\mathcal{O}} \models \top \sqsubseteq \exists R.\nabla \end{aligned}$$

□

Proof of Proposition 3.2.53 (b). Let R be an atomic role and $I_{\mathcal{O}}$ a model for R .

$$\begin{aligned} R \text{ is irreflexive} &\Leftrightarrow \forall x \in \Delta^I : (x, x) \notin R^I \\ &\Leftrightarrow \Delta^I \subseteq \Delta^I \setminus \{x \in \Delta^I \mid (x, x) \in R^I\} \\ &\Leftrightarrow I_{\mathcal{O}} \models \top \sqsubseteq \neg\exists R.\nabla \end{aligned}$$

□

One of the most common description logic languages is \mathcal{ALC} .

Definition 3.2.54 (\mathcal{ALC}). *The \mathcal{AL} extension with complex complement is called \mathcal{ALC} .*

Description logics are a fragment of first order predicate logic. A major challenge with description logic languages is to make that fragment as expressive as possible, while still keeping it decidable. Three expressive but decidable descriptions logic languages are \mathcal{SHIF} , \mathcal{SHOIN} , and \mathcal{SROIQ} . They are of particular importance in the context of the semantic web (see below).

Definition 3.2.55 (\mathcal{SHIF}). *The \mathcal{AL} extension with transitive, inverse, hierarchical and functional roles is called \mathcal{SHIF} . \mathcal{SHIF} extended with concrete data types is called $\mathcal{SHIF}^{(\mathcal{D})}$. [HPSH03]*

Definition 3.2.56 (\mathcal{SHOIN}). *The \mathcal{AL} extension with transitive, inverse, and hierarchical roles, nominals, and number restrictions is called \mathcal{SHOIN} . \mathcal{SHOIN} extended with concrete data types is called $\mathcal{SHOIN}^{(\mathcal{D})}$. [HPSH03]*

Definition 3.2.57 (\mathcal{SROIQ}). *The \mathcal{AL} extension with extended expressiveness for roles, with nominals, and with qualified number restrictions is called \mathcal{SROIQ} . \mathcal{SROIQ} extended with concrete data types is called $\mathcal{SROIQ}^{(\mathcal{D})}$. [HKS06]*

All \mathcal{AL} languages presented here are decidable. Decidability can be shown by providing a sound and complete algorithm for calculating a model (iff one exists). One type of such algorithms are tableau algorithms. We will sketch a tableau algorithm for \mathcal{ALC} (for \mathcal{ALU} , to be precise). A tableau for the very expressive \mathcal{SROIQ} language can be found in [HKS06].

Note that for reasons of simplicity, in the discussion of tableau algorithms we will treat an ABox as though it were solely a set of individual assertions $\mathcal{A} = X_I$ instead of $\mathcal{A} = (I, X_I)$.

Definition 3.2.58 (Tableau Algorithm for \mathcal{ALC}). *A description logics tableau algorithm checks if a complex concept C is satisfiable, i.e., if there exists an interpretation I_C for which $I_C \models C(p)$, for some individual p .*

A description logics tableau is a set of ABoxes that contain complex concepts parameterised with “new” individuals.

The tableau algorithm for \mathcal{ALC} takes a complex concept C in negated normal form¹ and returns a set of ABoxes. It starts with a tableau containing only the assertion $C(p)$ for some “new” individual p . This tableau is successively extended by applying rules from table 3.1, replacing ABoxes until no more rules can be applied.

An ABox contains a clash if it either contains the assertion $\perp(p)$, or if it contains both the assertion $A(p)$ and $\neg A(p)$, for an individual p and an atomic concept A . If all of the ABoxes returned by the tableau algorithm contain a clash, then the concept C is unsatisfiable, else if at least one ABox does not contain a clash, C is satisfiable.

Remark 3.2.59. *The tableau algorithm requires “new” individuals at several points. These are individuals that are not part of any ontology from which the concepts are taken. New individuals are required to avoid confusion with existing ones. They serve as place holders for existing individuals from an ontology, asserting the existence but not the identity of an individual with certain properties.*

Remark 3.2.60. *By using the equivalence*

$$C_1 \sqsubseteq C_2 \Leftrightarrow C_1 \sqcap \neg C_2 \text{ is unsatisfiable}$$

for complex concepts C_1 and C_2 , the tableau algorithm can be used to determine the validity of subsumptions: the subsumption is valid iff the intersection is unsatisfiable, i.e., iff all ABoxes returned by the tableau algorithms contain a clash.

\sqcap -rule	if \mathcal{A} contains $(C_1 \sqcap C_2)(p)$ but not both $C_1(p)$ and $C_2(p)$: return $\mathcal{A}' = \mathcal{A} \cup \{C_1(p), C_2(p)\}$
\sqcup -rule	if \mathcal{A} contains $(C_1 \sqcup C_2)(p)$ but neither $C_1(p)$ nor $C_2(p)$: return $\mathcal{A}' = \mathcal{A} \cup \{C_1(p)\}$ and $\mathcal{A}'' = \mathcal{A} \cup \{C_2(p)\}$
\exists -rule	if \mathcal{A} contains $\exists R.C(p)$ but neither $C(p')$ nor $R(p, p')$ for some p' : return $\mathcal{A}' = \mathcal{A} \cup \{C(p'), R(p, p')\}$
\forall -rule	if \mathcal{A} contains $\forall R.C(p)$ and $R(p, p')$, but not $C(p')$ for some p' : return $\mathcal{A}' = \mathcal{A} \cup \{C(p')\}$

Table 3.1: \mathcal{ALC} tableau rules for an ABox \mathcal{A} , for complex concepts C_1 and C_2 , an atomic role R , and individuals p and p' , and a “new” individual q . p , p' and q are parameters.

Example 3.2.61 (Tableau Algorithm for \mathcal{ALC}). *In order to check the validity of the subsumption*

$$\exists R.A \sqsubseteq \forall R.A$$

for an atomic role R and an atomic concept A , we check the unsatisfiability of

$$\exists R.A \sqcap \neg \forall R.A.$$

¹a complex concept where negation is only applied to atomic concepts; cf. proposition 3.2.33

Transformed into negated normal form, the concept reads

$$C_0 \doteq \exists R.A \sqcap \exists R.\neg A.$$

If C_0 is satisfiable, there must be an individual p_0 such that $C_0(p_0)$. We therefore start the tableau algorithm with a set containing a single ABox $\{\{C_0(p_0)\}\}$. Applying the \sqcap -rule yields $\{\{C_0(p_0), \exists R.A(p_0), \exists R.\neg A(p_0)\}\}$. Applying the \exists -rule twice leads to $\{\{\dots, A(p_1), R(p_0, p_1), \neg A(p_2), R(p_0, p_2)\}\}$. At this point, no further rules can be applied.

Not all of the ABoxes contain a clash, which means that C_0 is satisfiable. Therefore, the initial subsumption is invalid.

3.3 Semantic Web Technologies

This section is partially based on [Fre11]. Additional details can be found in [KC04, BG04, BvHH⁺, PS08].

3.3.1 RDF

The *Resource Description Framework* (RDF), a W3C recommendation since early 2004, is – as the name implies – a language for describing resources [KC04].

Definition 3.3.1 (RDF Resource). *An RDF resource is either a named RDF resource or a blank node.*

A named RDF resource is anything that can be identified by a URI (or by an IRI, a URI with international symbols). The URI serves primarily as a unique identifier, but best practice is to use URLs that point to content that is relevant for or that describes the resource. Namespace prefixes can be used to increase clarity.

Blank nodes are placeholders in an artificial “_” namespace. They are anonymous resources that are identified uniquely only in a local context, fulfilling a role similar to that of existential quantification.

Example 3.3.2 (RDF Resource). $\langle \text{http://www.uni-passau.de} \rangle$, $\langle \text{http://www.dancedescriptions.net/waltz} \rangle$, and $\langle \text{http://www.w3.org/2001/XMLSchema\#integer} \rangle$ are named RDF resources. $_:a$ and $_:b01645$ are blank nodes.

Definition 3.3.3 (RDF Literal). *An RDF literal is either an untyped literal, a typed literal, or a language literal.*

Untyped literals are literal, i.e., uninterpreted, values without data type information.

Typed literals are literal values with type information. RDF makes use of the XML Schema data types such as `integer`, `string`, or `date`. In addition, typed literals can have the `XMLLiteral` type, which refers to well-formed XML data. Formally, a typed literal is a tuple (l, t) , where l is an untyped literal and t is a data type.

Language literals are textual values with language information. The language information should be written in a standardised manner, for example using ISO 639 language codes such as “en” or “de”. Formally, a typed literal is a tuple (l, a) , where l is an untyped literal and a is a language code.

Note that an RDF literal cannot have both type and language information at the same time. In particular, language literals are not of type `string`.

Typed literals are not validated when they are parsed. Only when, for example, arithmetic operations are performed on integer-typed literals, does the processing system check if the literal values are actually integers. This is called “lazy validation”, in contrast to “eager validation” at parse-time.

Example 3.3.4 (RDF Literal). `"Waltz"`, `"ID638-238-910"`, and `"23½"` are untyped literals.

`("Waltz", xsd:string)`, `("85", xsd:integer)`, and
`("<dance>Waltz</dance>", XMLLiteral)` are typed literals.

`("Waltz", en)` and `("Langsamer Walzer", de)` are language literals.

Definition 3.3.5 (RDF Statement). An RDF statement is a triple consisting of subject, predicate and object. The subject can be any RDF resource, the predicate must be a named resource, and the object can be either an RDF resource or an RDF literal.

A statement (s, p, o) with a predicate p is also called a p -statement, and o is called the p -property value of s .

Example 3.3.6 (RDF Statement). The triple

`(dnc:waltz, dnc:name, ("Waltz", en))` is an RDF statement, with `dnc` a prefix for the namespace `<http://www.dancedescriptions.net/>`.

Definition 3.3.7 (RDF Graph). An RDF graph is a collection of RDF statements. We call R_N the set of named resources, R_B the set of blank nodes, and $R = R_N \cup R_B$ the set of RDF resources in these statements. We call L_U the set of untyped literals, L_T the set of typed literals, L_L the set of language literals, and $L = L_U \cup L_T \cup L_L$ the set of RDF literals in these statements.

As a graph structure, an RDF graph is a tuple (N, E) , where $N = R \cup L$ and $E \subseteq R \times R_N \times (R \cup L)$. An RDF graph is an edge-annotated directed graph.

Example 3.3.8 (RDF Graph). Figure 3.1 shows an example of an RDF graph. It contains ten named resources, one blank node (indicated by a dashed circle), one untyped literal, one typed literal, four language literals, and a total of ten statements. Of the ten named resources, six are used as predicates in the statements.

We will briefly discuss two serialisation methods for RDF graphs here: turtle and XML. In turtle format, all statements are listed one by one, separated by a single dot “.”. URIs are enclosed in `<>`, literals are enclosed in quotation marks, type information is appended to a literal after “^^”, and language information is appended to a literal after “@”.

Namespace prefixes can be defined using the `@prefix` keyword. There are shorthands for multiple statements with the same subject and for blank nodes.

Example 3.3.9 (Turtle). The following code shows the RDF graph from example 3.3.8 in turtle

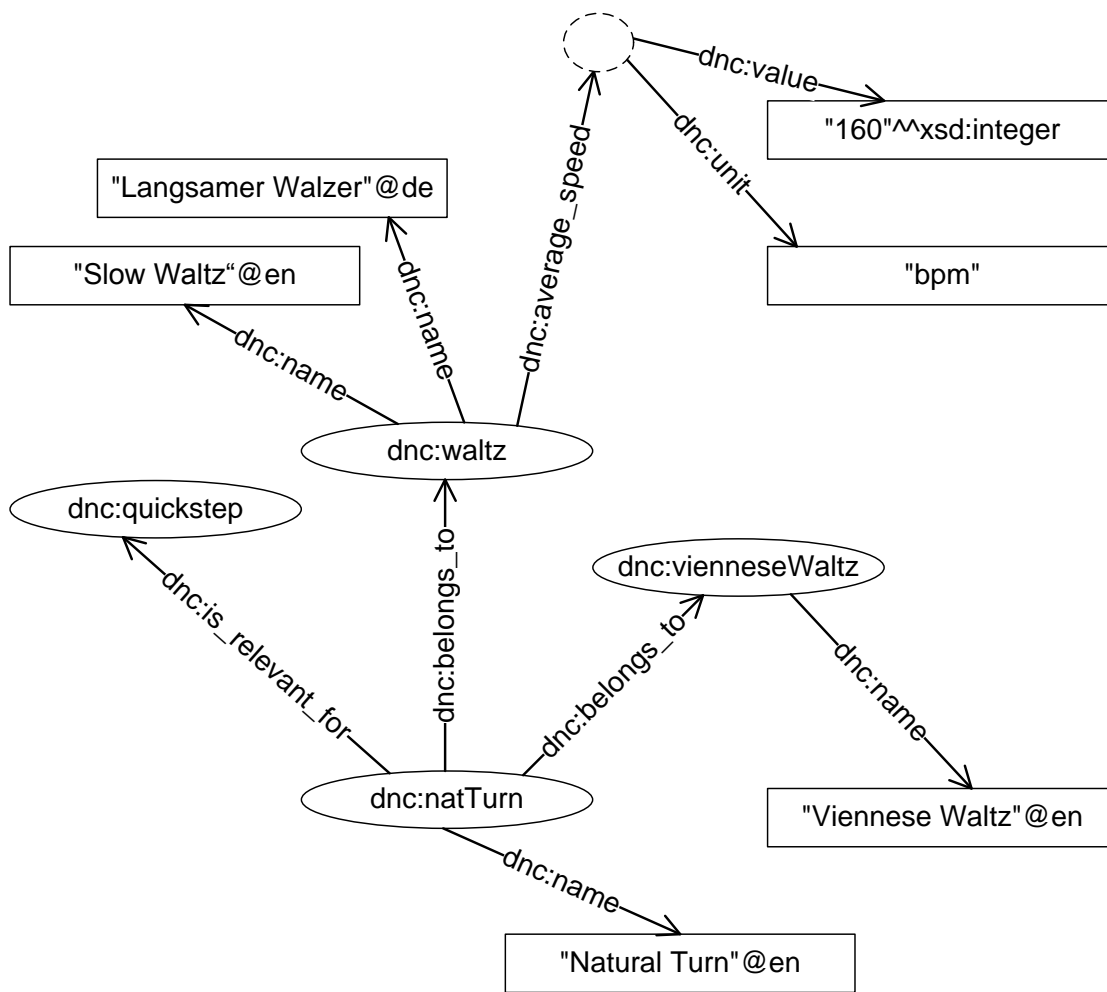


Figure 3.1: RDF Graph

syntax.

```

@prefix dnc: <http://www.dancedescriptions.net/> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
dnc:natTurn      dnc:name          "Natural Turn"@en .
dnc:waltz        dnc:name          "Slow Waltz"@en ;
                 dnc:name          "Langsamer Walzer"@de ;
                 dnc:average_speed [
                   dnc:value        "160"^^xsd:integer ;
                   dnc:unit         "bpm"
                 ] .
dnc:vienneseWaltz dnc:name        "Viennese Waltz"@en .
dnc:natTurn      dnc:belongs_to   dnc:waltz .
dnc:natTurn      dnc:belongs_to   dnc:vienneseWaltz .
dnc:natTurn      dnc:is_relevant_for dnc:quickstep .

```

First, two namespace prefixes are defined. Then, the ten statements are listed one after the other. For `dnc:waltz`, several statements with the same subject are written in abbreviated form. The blank node is also used in abbreviated syntax, instead of introducing a `_:x` resource and writing the statements as

```

dnc:waltz dnc:average_speed _:x .
_:x       dnc:value         "160"^^xsd:integer .
_:x       dnc:unit         "bpm" .

```

The XML serialisation is more verbose than the turtle syntax, and it offers many alternatives for specifying the same data. The XML root element is always `<rdf:RDF>`, in the `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` namespace. RDF statements are encoded as `<rdf:Description>` elements, with an `rdf:about` attribute that specifies the subject resource. There also exist several options for abbreviating the XML syntax. Additional details can be found in [BM04a].

Example 3.3.10 (RDF XML). *The following XML code is a serialisation the RDF graph from example 3.3.8.*

```

1 <!DOCTYPE RDF [
2   <!ENTITY dnc "http://www.dancedescriptions.org/">
3   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
4 ]>
5 <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
6   xmlns:dnc="http://www.dancedescriptions.org/"
7   <rdf:Description rdf:about="@dnc:natTurn">
8     <dnc:name xml:lang="en">Natural Turn</dnc:name>
9     <dnc:belongs_to rdf:resource="@dnc:waltz"/>
10    <dnc:belongs_to
11      <rdf:Description rdf:about="@dnc:vienneseWaltz">
12        <dnc:name xml:lang="en">Viennese Waltz</dnc:name>
13      </rdf:Description>
14    </dnc:belongs_to>
15    <dnc:is_relevant_for
16      <rdf:Description rdf:about="@dnc:quickstep"/>
17    </dnc:is_relevant_for>
18  </rdf:Description>
19  <rdf:Description rdf:about="@dnc:waltz">
20    <dnc:name xml:lang="en">Slow Waltz</dnc:name>
21    <dnc:name xml:lang="de">Langsamer Walzer</dnc:name>
22    <dnc:average_speed rdf:parseType="Resource">
23      <dnc:value rdf:datatype="@xsd:integer">160</dnc:value>
24      <dnc:unit>bpm</dnc:unit>

```

```

25     </dnc:average_speed>
26     </rdf:Description>
27 </rdf:RDF>

```

First, two XML entities are defined to improve the legibility of the code. In lines 7 and 19, two `<description>` elements are opened that serve as the subjects for several statements. In line 11, another `<description>` element is opened that serves as both the object for one statement and as the subject for another statement. Line 22 starts a blank node.

3.3.2 RDF Schema

RDF Schema (RDFS) is also a W3C recommendation as of early 2004 [BG04]. It is a language for specifying RDF vocabulary. It is built on top of RDF and introduces semantics based on first-order predicate logic. It adheres to the open world assumption.

RDFS introduces a resource named `rdfs:Class` in the `<http://www.w3.org/2000/01/rdf-schema#>` namespace for classifying resources. If an RDF graph contains a statement (c , `rdf:type`, `rdfs:Class`), then c is considered to be a “class”. If, in turn, the same RDF graph also contains a statement (r , `rdf:type`, c), then the resource r is considered to instantiate the class c . A resource can instantiate multiple classes, and each class represents the set of resources that instantiate it.

Similarly, `rdf:Property` in the `<http://www.w3.org/1999/02/22-rdf-syntax-ns#>` namespace is used for the classification of resources that are used as predicates in statements. Instantiation of `rdf:Property` is again indicated with `rdf:type`, which is a property itself and thus instantiates `rdf:Property`.

RDFS supports generalisation of both classes and property types with the properties `rdfs:subClassOf` and `rdfs:subPropertyOf`. Both properties are transitive and reflexive. The class `rdfs:Resource` serves as a common super-class for all RDF classes.

Example 3.3.11 (RDFS). *The following RDF graph in turtle syntax defines two classes, one a sub-class of the other, a class instantiation, and two properties in a generalisation relationship.*

```

dnc:StandardDance    rdf:type          rdfs:Class .
dnc:Dance            rdf:type          rdfs:Class .
dnc:StandardDance    rdfs:subClassOf  dnc:Dance .
dnc:waltz            rdf:type          dnc:StandardDance .
dnc:belongs_to       rdf:type          rdf:Property .
dnc:is_relevant_for  rdf:type          rdf:Property .
dnc:belongs_to       rdfs:subPropertyOf dnc:is_relevant_for .

```

Some of the behaviour of RDFS classes is similar to classes or types in other paradigms. For example, if an RDF resource r instantiates an RDFS class c , then it also instantiates all super-classes of c . In an enhanced entity-relationship (EER) model, if an entity e belongs to an entity type t , then there is an associated entity e' in the super-type of t (provided that t has a super-type). In the unified modelling language (UML), if an object o instantiates a class c , then o can be used in any place where an instance of a super-class of c is required.

However, in UML and EER, attributes are specified locally for a class or entity type. In RDFS, properties are specified globally. Also, in UML, class attributes cannot be optional: if a class defines an attribute, then all instantiating objects have this attribute. On the other hand, in RDFS, properties cannot be defined as mandatory for instances of a class.

RDFS allows the specification of restrictions on the domain and range of properties. Using `rdfs:domain` and `rdfs:range`, the classification of subjects and objects in statements with a specific property can be restricted.

For example, the statement (`rdf:type`, `rdfs:range`, `rdfs:Class`) specifies that all objects in `rdf:type`-statements are instances of `rdfs:Class`. This is consistent with the open world assumption: if a resource is used as the object in an `rdf:type`-statement, then it is assumed (or actually inferred) to be a class. Under the closed world assumption, an RDFS processor would have to check if the object is explicitly *known* to be a class, and throw an error if it is not.

It is possible to place multiple restrictions on a property. The semantics of multiple restriction is the intersection of their class requirements. So if the range of a property p is restricted to two classes c_1 and c_2 , then only resources that instantiate both c_1 and c_2 can be the object of p -statements.

RDFS introduces two types of aggregations: `rdfs:Container` and `rdfs:Collection`. There are three types of containers, namely `rdf:Bag`, which represents an unordered list, `rdf:Seq`, which represents an ordered list, and `rdf:Alt`, which represents a list of alternatives. Membership in any container is defined using the `rdfs:member` property and its sub-properties `rdf:_1`, `rdf:_2`, `...`

There is currently only one collection type, `rdf:List`. Lists are defined in a manner reminiscent of functional programming languages. The empty list is represented by the special list instance `rdf:nil`. A non-empty list is defined as a resource (usually a blank node) with a head and a tail. The head, specified using the `rdf:first` property, points to a resource, and the tail, specified using the `rdf:rest` property, points to another list. Other than containers, collections are regarded as closed, i.e., they contain only the elements that are specified and no more. This is in slight contrast to the open world assumption.

RDFS provides some limited support for reification. Reification is the re-specification of an expression in a language l using the language l itself. In RDFS, reification allows treating a statement as a resource, allowing statements about other statements.

However, the reification semantics in RDFS are very weak, because there is no logical relationship between a statement and its reified form, and neither does a regular statement entail its reified form nor vice versa.

Example 3.3.12 (Reification). *The following turtle code specifies an RDF statement about the waltz that is then reified. A note is added to the reified statement.*

```
dnc:waltz    dnc:name      "Slow Waltz"@en .
_:s         rdf:type      rdf:Statement .
_:s         rdf:subject   dnc:waltz .
_:s         rdf:predicate dnc:name .
_:s         rdf:object    "Slow Waltz"@en .
_:s         dnc:note      "Often simply called 'Waltz'" .
```

Some RDF graphs represent formal ontologies as defined in section 3.2, provided that they do not contain classes that instantiate other classes, or other things that are not supported by the description logics semantics. Vice versa, some formal ontologies can be represented as RDF graphs, provided that they do not contain transitivity axioms or other things that are not supported by the RDF semantics.

Persistent Storage

Apart from the file-based serialisations for RDF data described above, there exist several schemas for storing RDF data in relational database systems. A schema that is both simple and very

triples		
subject	predicate	object
dnc:natTurn	rdf:type	dnc:Figure
dnc:natTurn	dnc:belongs_to	dnc:waltz
dnc:natTurn	dnc:belongs_to	dnc:vienneseWaltz
dnc:natTurn	dnc:name	<i>"Natural Turn"@en</i>
dnc:waltz	rdf:type	dnc:Dance
dnc:waltz	dnc:name	<i>"Slow Waltz"@en</i>
dnc:vienneseWaltz	rdf:type	dnc:Dance
dnc:vienneseWaltz	dnc:name	<i>"Viennese Waltz"@en</i>
dnc:Figure	rdf:type	rdfs:Class
dnc:Dance	rdf:type	rdfs:Class

Table 3.2: Example: triple table

properties			
subject	rdf:type	dnc:belongs_to	dnc:name
dnc:natTurn	dnc:Figure	dnc:waltz	<i>"Natural Turn"@en</i>
dnc:natTurn	dnc:Figure	dnc:vienneseWaltz	<i>"Natural Turn"@en</i>
dnc:waltz	dnc:Dance	NULL	<i>"Slow Waltz"@en</i>
dnc:vienneseWaltz	dnc:Dance	NULL	<i>"Viennese Waltz"@en</i>
dnc:Figure	rdfs:Class	NULL	NULL
dnc:Dance	rdfs:Class	NULL	NULL

Table 3.3: Example: property table

common is based on a *triple table* with three attributes **subject**, **predicate** and **object** to store statements, as shown in table 3.2. Resource URIs are often put into a separate table and referred to by an identifier. Variations on triple tables differentiate between resource-valued objects and literal-valued objects [WSKR03], or allow for more than one RDF graph in a single table by introducing a fourth attribute for a graph name.

Another common approach are *property tables* (as shown in table 3.3), where the table consists of a **subject** attribute that represents the subjects of statements, and several attributes that represent various common properties. Such a table holds an individual in the subject-column and several property values (objects of statements) in its property-columns. This schema is useful if there are a small number of very common properties and little fluctuation in the RDF Schema. A disadvantageous selection of properties can lead to many duplicate rows (for properties with more than one value per subject) or many NULL values (for properties with no value for some subjects). Property tables are usually combined with a triple table to hold statements with properties that are not represented in the property table.

A more complex schema consists of *class-property tables*, where a separate property table exists for each RDFS class as shown in table 3.4. This is based on the assumption that instances of the same class will often share more common properties than instances of different classes, alleviating some of the disadvantages of pure property tables. Again, this approach is usually complemented with a triple table.

Vertical partitioning is an orthogonal approach to class-property tables, as it creates separate tables for common properties [AMMH07, AMMH09]. Each of these tables have two attributes

class_figure		
subject	dnc:belongs_to	dnc:name
dnc:natTurn	dnc:waltz	<i>"Natural Turn"@en</i>
dnc:natTurn	dnc:vienneseWaltz	<i>"Natural Turn"@en</i>

class_dance	
subject	dnc:name
dnc:waltz	<i>"Slow Waltz"@en</i>
dnc:vienneseWaltz	<i>"Viennese Waltz"@en</i>

Table 3.4: Example: class-property tables

property_dnc_belongs_to	
subject	object
dnc:natTurn	dnc:waltz
dnc:natTurn	dnc:vienneseWaltz

property_dnc_name	
subject	object
dnc:natTurn	<i>"Natural Turn"@en</i>
dnc:waltz	<i>"Slow Waltz"@en</i>
dnc:vienneseWaltz	<i>"Viennese Waltz"@en</i>

property_rdfs_class	
subject	object
dnc:natTurn	dnc:Figure
dnc:waltz	dnc:Dance
dnc:vienneseWaltz	dnc:Dance
dnc:Figure	rdfs:Class
dnc:Dance	rdfs:Class

Table 3.5: Example: vertical partitioning

`subject` and `object` and hold all statements with the property represented by the table. While solving most problems of property tables and of class-property tables, the partitioning by property does not align well with many retrieval scenarios that are often centred on the subject. This schema can also be combined with a triple table. An example is shown in table 3.5.

There is no clear “best” solution for storing RDF data in a relational database system. The schema that is best suited depends on the structure of the RDF data and on the usage scenario. There are several papers claiming advantages for a specific schema in a specific case [LM07, SGK⁺08, SGD⁺09]. Without detailed knowledge of the application domain, triple tables are a solid catch-all solution.

3.3.3 OWL

To address some of the shortcomings of RDF in terms of semantics, the *Web Ontology Language* (OWL) was developed and became a W3C recommendation early 2004 [BvHH⁺]. It consists of three sub-languages: OWL Lite, OWL DL, and OWL Full, each of which is a subset of the next one.

OWL Lite is based on the description logic $\mathcal{SHL}\mathcal{F}^{(\mathcal{D})}$. It is a small language and is efficient in terms of reasoning complexity. However, due to its limitations it is rarely used. OWL DL is based on the more powerful description logic $\mathcal{SHOIN}^{(\mathcal{D})}$. It is expressive and reasonably efficient, and therefore the OWL sub-language that is used the most. It is also the language that we will describe here. OWL Full contains all of RDF(S), but is undecidable and little used.

Even though RDF does not support the entire OWL semantics, all valid OWL documents are valid RDF documents, so that the RDF syntax and serialisation can be used for OWL. Additional OWL commands are written in the OWL namespace `<http://www.w3.org/2002/07/owl#>`. An overview of OWL commands and their semantics based on description logics can be found in table 3.6.

All OWL documents contain an OWL metadata section that describes the OWL document itself. We will not give details about this metadata section here except to name the `owl:imports` command that can be used to import other OWL documents into the current one.

While OWL is based on description logics, it uses a different terminology. Description logics concepts are called *classes* in OWL, individuals are called *objects*, and roles are called *properties*. This terminology is consistent with RDF. Different from RDF, but consistent with description logics, is the strict separation of types: the sets of classes, objects, properties, and data values are disjoint in OWL.

To differentiate from the less restrictive classes in RDF, OWL introduces its own class resource `owl:Class`. Instantiation and generalisation are still specified with `rdf:type` and `rdfs:subClassOf`, respectively. For class equivalence, the new `owl:equivalentClass` property is introduced. Special classes that correspond to the description logics concepts \top and \perp are defined with `owl:Thing` and `owl:Nothing`.

Classes can be defined as intersections of other classes with `owl:intersectionOf`, as a union of other classes with `owl:unionOf`, and as the complement of another class with `owl:complementOf`. Two classes can be defined as disjoint with `owl:disjointWith`.

Other ways to define classes are by enumeration with the set-constructor `owl:oneOf`, through unqualified cardinality constraints (see example 3.3.13), and through existential and universal quantification (see example 3.3.14).

Example 3.3.13 (OWL Cardinality Constraints). *The following OWL document defines a figure as something that has at least one step. In description logics, this can be expressed as*

Figure $\sqsubseteq_{\geq 1}$ has_step.

```
dnc:Figure rdfs:subClassOf _:a .
_:a rdf:type owl:Restriction .
_:a owl:onProperty dnc:has_step .
_:a owl:minCardinality "1"^^xsd:integer .
```

Example 3.3.14 (OWL Quantification). *The following OWL document defines a figure as something that only contains steps, and a dance as something that contains at least one figure. In description logics, this can be expressed as*

Figure $\sqsubseteq \forall$ contains.Step, Dance $\sqsubseteq \exists$ contains.Figure.

```

dnc:Figure    rdfs:subClassOf    _:b .
_:b           rdf:type           owl:Restriction .
_:b           owl:onProperty   dnc:contains .
_:b           owl:allValuesFrom dnc:Step .
dnc:Dance    rdfs:subClassOf    _:c .
_:c           rdf:type           owl:Restriction .
_:c           owl:onProperty   dnc:contains .
_:c           owl:someValuesFrom dnc:Figure .

```

Similar to the new resource for classes, there are new resources for properties in OWL: `owl:ObjectProperty` for properties that can have objects as values, and `owl:DatatypeProperty` for properties with literal values. Property generalisation is still expressed with `rdfs:subPropertyOf`, and property equivalence can now be expressed with `owl:equivalentProperty`.

Inverses of properties can be specified with `owl:inverseOf`. Transitive, symmetric, functional, and injective properties can be specified as sub-properties of `owl:TransitiveProperty`, `owl:SymmetricProperty`, `owl:FunctionalProperty`, and `owl:InverseFunctionalProperty`, respectively. A property r is functional if $\forall a, b, c : r(a, b) \wedge r(b, c) \Rightarrow a = b$. A property r is injective if $\forall a, b, c : r(a, c) \wedge r(b, c) \Rightarrow a = b$. Both `rdfs:domain` and `rdfs:range` can also still be used.

Individuals (objects) can be declared as equivalent or explicitly not equivalent with `owl:sameAs` and `owl:differentFrom`, respectively.

Late 2009, an enhanced version of OWL, OWL 2, became a W3C recommendation. OWL 2 is based on the description logic $SR\mathcal{OIQ}^{\mathcal{D}}$. It also introduces new syntactical and modelling conveniences.

As one of these conveniences, OWL 2 introduces the concept of *punning*, which relaxes the unique name assumption by allowing a class, an object, and a property to all share the same name. They are, however, still semantically distinct, making the usefulness of punning questionable.

In OWL 2, properties can now be specified as asymmetric, reflexive, or irreflexive by making them sub-properties of `owl:AsymmetricProperty`, `owl:ReflexiveProperty`, or `owl:IrreflexiveProperty`, respectively. A property r is asymmetric if $\forall a, b : a \neq b \wedge r(a, b) \Rightarrow \neg r(b, a)$. A property r is reflexive if $\forall a : r(a, a)$. A property r is irreflexive if $\nexists a : r(a, a)$.

Special properties that correspond to the description logics roles \top and \perp are `owl:topObjectProperty` and `owl:bottomObjectProperty` for object-valued properties, and `owl:topDataProperty` and `owl:bottomDataProperty` for literal-valued properties.

A powerful new mechanism in OWL 2 is the `owl:propertyChainAxiom` for property composition. This corresponds to role composition ($r_1 \sqsubseteq r_2 \circ r_3$) in description logics.

In addition, OWL 2 now supports *qualified* cardinality constraints with `owl:minQualifiedCardinality`, `owl:maxQualifiedCardinality`, and `owl:qualifiedCardinality` on a class specified with `owl:onClass`.

OWL 2 also allows declaring that two resources are *not* in a specific relationship, as shown in example 3.3.15.

Example 3.3.15 (OWL 2 Negative Property Assertion). *The following OWL document states that the natural turn does not belong to the tango. In description logics, this can be expressed as*

$$\{natTurn\} \sqcap \exists belongs_to. \{tango\} \sqsubseteq \perp.$$

OWL	Description logics
<code>C1 rdfs:subClassOf C2</code>	$C_1 \sqsubseteq C_2$
<code>r1 rdfs:subPropertyOf r2</code>	$r_1 \sqsubseteq r_2$
<code>C1 owl:equivalentClass C2</code>	$C_1 \equiv C_2$
<code>r1 owl:equivalentProperty r2</code>	$r_1 \equiv r_2$
<code>owl:Thing</code>	\top
<code>owl:Nothing</code>	\perp
<code>owl:intersectionOf <C1, C2></code>	$C_1 \sqcap C_2$
<code>owl:unionOf <C1, C2></code>	$C_1 \sqcup C_2$
<code>owl:complementOf C1</code>	$\neg C_1$
<code>C1 owl:disjointWith C2</code>	$C_1 \sqcap C_2 \sqsubseteq \perp$
<code>owl:oneOf <i1, i2></code>	$\{i_1, i_2\}$
<code>owl:inverseOf r1</code>	r_1^{-}
<code>r1 rdf:type owl:TransitiveProperty</code>	$r_1 \equiv r_1^+$
<code>r1 rdf:type owl:SymmetricProperty</code>	$r_1 \equiv r_1^{-}$
<code>r1 rdf:type owl:FunctionalProperty</code>	$\top \sqsubseteq \leq 1r_1$
<code>r1 rdf:type owl:InverseFunctionalProperty</code>	$\top \sqsubseteq \leq 1r_1^{-}$
<code>r1 rdfs:domain C1</code>	$\exists r_1. \top \sqsubseteq C_1$
<code>r1 rdfs:range C1</code>	$\top \sqsubseteq \forall r_1. C_1$
<code>i1 owl:sameAs i2</code>	$\{i_1\} \equiv \{i_2\}$
<code>i1 owl:differentFrom i2</code>	$\{i_1\} \sqcap \{i_2\} \sqsubseteq \perp$
<code>owl:propertyChainAxiom <r1, r2> (OWL 2 only)</code>	$r_1 \circ r_2$

Table 3.6: OWL constructs and their corresponding description logics expressions, with classes C_1 and C_2 , concepts C_1 and C_2 , properties r_1 and r_2 , roles r_1 and r_2 , objects i_1 and i_2 , and individuals i_1 and i_2 . $\langle \dots \rangle$ denotes a list.

```
_:d rdf:type owl:NegativePropertyAssertion .
_:d owl:sourceIndividual dnc:natTurn .
_:d owl:assertionProperty dnc:belongs_to .
_:d owl:targetIndividual dnc:tango .
```

It is now also possible to define new data types by combining and restricting existing ones. For details, we refer the reader to [HKR09, Fre11].

Table 3.6 provides an overview of important OWL constructs and their corresponding description logics counterparts.

3.3.4 SKOS

The *Simple Knowledge Organization System* (SKOS) is a pre-defined vocabulary for taxonomies, implemented in OWL [SKO]. It uses the namespace `<http://www.w3.org/2004/02/skos/core#>`.

SKOS defines its own notion of a concept `skos:Concept`, which serves as the base class for the taxonomy. Instances of this concept can have one preferred label (`skos:prefLabel`) and several alternate labels (`skos:altLabel`).

Concept instances can also be declared as broader (`skos:broader`) or narrower (`skos:narrower`) than other instances. Neither `skos:broader` nor `skos:narrower` are transitive, but they are sub-

properties of `skos:broaderTransitive` and `skos:narrowerTransitive`, respectively, which are transitive. Because of the sub-property relationship, the transitive closure of `skos:broaderTransitive` also contains the transitive closure of `skos:broader` (analogue for `skos:narrower`). These properties should be used to model relationships that resemble generalisations or specialisations. Note that a statement (a , `skos:broader`, b) is supposed to indicate that b is broader than a , not the other way around.

Two concept instances can also be defined as related (`skos:related`), which should be used for objects that are associated with one another, but not in a hierarchical way.

Instances of the class `skos:ConceptScheme` should represent an aggregation of concepts, for example all relevant concepts for a specific domain, or for a specific application. Scheme membership can be declared for any concept instance with the `skos:inScheme` property. A scheme instance can have a top concept that is at the top of the hierarchy defined by `skos:broader`/`skos:narrower`, i.e., the top concept should be the broadest concept in the scheme. This is specified with either `skos:hasTopConcept` on the scheme instance, or with `skos:topConceptOf` on the concept instance. There are, however, no formal semantics in place to ensure any of these constraints, or to infer additional information.

For concept instances in different schemes, the properties `skos:broadMatch` and `skos:narrowMatch`, which are sub-properties of `skos:broader` and `skos:narrower`, respectively, can be used to define a hierarchy. In addition, the properties `skos:closeMatch` and `skos:exactMatch` allow for the definition of a close association and an equivalence between two concept instances, respectively.

We recognise that SKOS documents do not necessarily represent formal taxonomies as defined in definition 3.2.25, because the hierarchical relations are not irreflexive. We will, however, treat the relations as if they were irreflexive, allowing us to use SKOS as a modelling language for taxonomies in this thesis. This is compatible with the intended semantics of SKOS as described in [SKO].

Example 3.3.16 (SKOS). *The SKOS document shown in table 3.7 models the hierarchical relationships between several categories of dances. It can be written as a description logics ontology (with stronger semantics) as follows:*

$$\begin{aligned}
 C &= \{ \textit{ConceptScheme}, \textit{Concept} \}, \\
 R &= \{ \textit{inScheme}, \textit{topConceptOf}, \textit{prefLabel}, \textit{altLabel}, \textit{broader} \}, \\
 I &= \{ \textit{Dancing}, \textit{Dances}, \textit{StandardDances}, \textit{LatinDances}, \textit{SwayDances}, \\
 &\quad \textit{RhythmDances}, \textit{Dances_en}, \textit{Standard_Dances_en}, \textit{Latin_Dances_en}, \\
 &\quad \textit{Latin-American_Dances_en}, \textit{Lateinamerikanische_Tänze_de}, \\
 &\quad \textit{Schwungtänze_de} \}, \textit{and} \\
 X &= \{ \textit{ConceptScheme}(\textit{Dancing}), \textit{Concept}(\textit{Dances}), \\
 &\quad \textit{Concept}(\textit{StandardDances}), \textit{Concept}(\textit{LatinDances}), \\
 &\quad \textit{Concept}(\textit{SwayDances}), \textit{Concept}(\textit{RhythmDances}), \\
 &\quad \textit{inScheme}(\textit{Dances})(\textit{Dancing}), \textit{inScheme}(\textit{StandardDances})(\textit{Dancing}), \\
 &\quad \textit{inScheme}(\textit{LatinDances})(\textit{Dancing}), \textit{inScheme}(\textit{SwayDances})(\textit{Dancing}), \\
 &\quad \textit{inScheme}(\textit{RhythmDances})(\textit{Dancing}), \\
 &\quad \textit{topConceptOf}(\textit{Dances})(\textit{Dancing}), \\
 &\quad \textit{prefLabel}(\textit{Dances})(\textit{Dances_en}), \\
 &\quad \textit{prefLabel}(\textit{StandardDances})(\textit{Standard_Dances_en}), \\
 &\quad \textit{prefLabel}(\textit{LatinDances})(\textit{Latin_Dances_en}), \\
 &\quad \textit{prefLabel}(\textit{SwayDances})(\textit{Schwungtänze_de}), \\
 &\quad \textit{altLabel}(\textit{LatinDances})(\textit{Latin-American_Dances_en}), \\
 &\quad \textit{altLabel}(\textit{LatinDances})(\textit{Lateinamerikanische_Tänze_de}),
 \end{aligned}$$

dnc:Dancing	rdf:type	skos:ConceptScheme .
dnc:Dances	rdf:type	skos:Concept ;
	skos:inScheme	dnc:Dancing ;
	skos:topConceptOf	dnc:Dancing ;
	skos:prefLabel	"Dances"@en .
dnc:StandardDances	rdf:type	skos:Concept ;
	skos:inScheme	dnc:Dancing ;
	skos:prefLabel	"Standard Dances"@en ;
	skos:broader	dnc:Dances .
dnc:LatinDances	rdf:type	skos:Concept ;
	skos:inScheme	dnc:Dancing ;
	skos:prefLabel	"Latin Dances"@en ;
	skos:altLabel	"Latin-American Dances"@en ;
	skos:altLabel	"Lateinamerikanische Tänze"@de ;
	skos:broader	dnc:Dances .
dnc:SwayDances	rdf:type	skos:Concept ;
	skos:inScheme	dnc:Dancing ;
	skos:prefLabel	"Schwungtänze"@de ;
	skos:broader	dnc:StandardDances .
dnc:RhythmDances	rdf:type	skos:Concept ;
	skos:inScheme	dnc:Dancing ;
	skos:broader	dnc:LatinDances .

Table 3.7: Example: SKOS document.

broader(StandardDances)(Dances),
broader(LatinDances)(Dances),
broader(SwayDances)(SwayDances),
broader(RhythmDances)(LatinDances)}.

3.3.5 SPARQL

SPARQL (pronounced “sparkle”), the *SPARQL Protocol And RDF Query Language*, is a W3C recommendation since early 2008 [PS08]. It supports selection, projection, joins, filtering, ordering, grouping, and top-k queries. *SPARQL* is a query language on RDF graphs, where graph patterns that contain placeholders are specified and matched against an RDF graph. Sub-graphs that match the pattern are found, and the placeholders are filled with the appropriate values. All such valuations for the placeholders form the result of the query.

Definition 3.3.17 (RDF Subgraph). *Let $G = (N, E)$ be an RDF graph. Then $G' = (N', E')$ is an RDF subgraph of G , with $N' \subseteq N$ and $E' \subseteq E$ restricted to N' , i.e., $E' \subseteq \{(s, p, o) \mid \exists s, p, o \in N' : (s, p, o) \in E\}$.*

Example 3.3.18 (RDF Subgraph). *Let G be the RDF graph from example 3.3.8. Then both G itself and $G' = (\{dnc:natTurn, dnc:belongs_to, dnc:waltz\}, \{dnc:natTurn, dnc:belongs_to, dnc:waltz\})$ are RDF subgraphs of G .*

Definition 3.3.19 (RDF Subgraph Pattern). *Let $G' = (N', E')$ be an RDF subgraph of an RDF graph G . Then $G'_P = (N'_P, E'_P)$ is an RDF subgraph pattern of G , where P is a set of placeholders, with $N'_P = N' \cup P$ and $E'_P \subseteq \{(s, p, o) \mid \exists s, p, o, s', p', o' \in N'_P : (s', p', o') \in E' \wedge ((s = s') \vee (s \in P)) \wedge ((p = p') \vee (p \in P)) \wedge ((o = o') \vee (o \in P))\}$.*

Intuitively, G'_P is a subgraph of G , where some components of the edges E'_P have been replaced with placeholders from P .

Example 3.3.20 (RDF Subgraph Pattern). Let G be the RDF graph from example 3.3.8. Let $P = \{?x\}$ be a set of placeholders. Then $G'_P = (\{\text{dnc:natTurn}, \text{dnc:belongs_to}, \text{dnc:waltz}, ?x\}, \{(\text{dnc:natTurn}, \text{dnc:belongs_to}, ?x)\})$ is an RDF subgraph pattern of G .

Definition 3.3.21 (SPARQL Query). A SPARQL query $Q = (S, W, F, f, D, k, o)$ consists of

- S : a list of placeholders that define the query result,
- W : a combination of RDF subgraph patterns that define the actual query,
- F : a set of filter conditions that can be attached to (some of) the subgraph patterns,
- f : a relation that attaches filter conditions to subgraph patterns,
- D : a set of ordering instructions,
- k : the maximum number of rows to be returned as the query result, with $k = \infty$ by default, and
- o : the offset of rows to be returned in the query result, with $o = 0$ by default.

Syntactically, S is indicated by the **SELECT** keyword, W is indicated by the **WHERE** keyword and framed by curly braces “{ }”, elements of F are indicated by the **FILTER** keyword within the confines of the W braces, f -relationships are defined by syntactic proximity of filter expression to elements of W , D is indicated by the **ORDER BY** keyword, k is indicated by the **LIMIT** keyword, and o is indicated by the **OFFSET** keyword.

Namespace prefixes can be defined using the **PREFIX** keyword.

Example 3.3.22 (SPARQL Query). The following SPARQL query selects the first ten resources to which `dnc:natTurn` belongs and that are not blank nodes, ordered by the URI of the resource.

```

1 PREFIX dnc: <http://www.dancedescriptions.net/>
2 SELECT ?x
3 WHERE {
4     dnc:natTurn dnc:belongs_to ?x
5     FILTER (!isBLANK(?x))
6 }
7 ORDER BY ?x
8 LIMIT 10
9 OFFSET 0

```

Definition 3.3.23 (SPARQL Query Result). For a SPARQL query $Q = (S, W, F, f, D, k, o)$, the SPARQL query result is an $n + 1$ -ary relation, where n is the number of placeholders in S . The first place in the relation is taken by a number $r \in \mathbb{Z}$ that represents the “row number”, and the final n places are taken by the valuations of the n placeholders. The row number starts at zero and is increased by one for each new entry in the relation that becomes part of the query result.

Intuitively, a SPARQL query result can be seen as a table, where the rows are numbered by r and where each column is labelled with one of the placeholders. Accordingly, we will show SPARQL query results in the form of tables where convenient.

The combination of subgraph patterns and filter expressions from the query Q is matched against an RDF graph G , resulting in a number of matching RDF subgraphs. For each of these subgraphs, the values that match the placeholders become a new row in the query result. The order of these rows is either determined by the ordering instructions D , or by the query processor if D is empty.

SPARQL queries ignore the open world assumption that usually underlies RDF/OWL data in favour of the closed world assumption. The latter is a more sensible choice for querying, since it leads to concrete, tangible and finite results.

Example 3.3.24 (SPARQL Query Result). *The SPARQL query from example 3.3.22 applied to the RDF graph from example 3.3.8 results in two matching subgraphs:*

*(`dnc:natTurn`, `dnc:belongs_to`, `dnc:waltz`) and
(`dnc:natTurn`, `dnc:belongs_to`, `dnc:vienneseWaltz`).*

Therefore, the placeholder `?x` matches `dnc:waltz` and `dnc:vienneseWaltz`, resulting in a SPARQL query result with two rows:

<i>r</i>	<i>?x</i>
0	<code>dnc:waltz</code>
1	<code>dnc:vienneseWaltz</code>

In SPARQL, placeholders are usually called *variables*, and are syntactically indicated by either “?” or “\$”. In a SPARQL query result, variables can be *unbound* in some or all rows, i.e., have no values. This can happen for variables that are part of the projection *S*, but that do not occur in the subgraph patterns. It can also happen for variables that occur in *optional* subgraph patterns (see below). In query results, we will represent unbound variables by a single dash “-”.

Blank nodes can be used in subgraph patterns (*W*), but not in *S*. In a subgraph pattern, a blank node is treated just like a variable.

Blank nodes can also occur in query results, when the query matches blank nodes that occur in the original RDF graph. It is, however, not guaranteed that the blank nodes in the query result will have the same names as the blank nodes in the RDF graph. It is only guaranteed that they will be used consistently, i.e., different blank nodes in the RDF graph will have different names in the query result, and equivalent blank nodes in the RDF graph will be equivalent in the query result.

Literals as part of subgraph patterns are matched according to their type: untyped literals can only be successfully matched against other untyped literals, typed literals can only be successfully matched against other typed literals with the same data type, and language literals can only be successfully matched against other language literals with the same language.

A SPARQL query can contain more than one subgraph pattern. They can be combined in three ways: conjunctively, where the common query result contains only rows that match the combination of both subgraph patterns; disjunctively, where the common result contains all rows that match either of the subgraph patterns; and optionally, where the query result contains all rows that match the first (mandatory) subgraph pattern, but variables in these rows may hold values from the second (optional) subgraph pattern.

Smaller subgraph patterns can be combined to larger subgraph patterns that can, in turn, be combined to even larger ones. A combination of subgraph patterns is called a *group pattern*. The smallest possible subgraph pattern, consisting of a single statement, is called a *triple*.

Conjunctive subgraph pattern combination is syntactically indicated by a “.” between the patterns. Disjunctive pattern combination is indicated by the UNION keyword, and optional patterns are indicated by the OPTIONAL keyword. Group patterns are enclosed in “{ }”.

Filter conditions from *F* can be attached to any subgraph pattern with the keyword FILTER, followed by the actual condition in parentheses “()”. Filter conditions can consist of (in-)equality checks between literals, resources and variables. They may also contain a number of pre-defined functions, such as type-check functions, arithmetic operations, and string functions.

Example 3.3.25 (SPARQL Query (continued)). *The following SPARQL query returns all resources `?x` that belong to something (called `?y`), or that are relevant for something (also called*

?y). For everything that *?x* belongs to, if it has an average speed, the value of this speed is returned as *?z*.

```

1 PREFIX dnc: <http://www.dancedescriptions.net/>
2 SELECT ?x ?y ?z
3 WHERE {
4   {
5     ?x dnc:belongs_to ?y .
6     OPTIONAL {
7       ?y dnc:average_speed _:a .
8       _:a dnc:value ?z
9     }
10  } UNION {
11   ?x dnc:is_relevant_for ?y
12 }
13 }
```

Applied to the RDF graph from example 3.3.8, the query yields the following result:

<i>r</i>	<i>?x</i>	<i>?y</i>	<i>?z</i>
0	<i>dnc:natTurn</i>	<i>dnc:waltz</i>	"160"^^xsd:integer
1	<i>dnc:natTurn</i>	<i>dnc:vienneseWaltz</i>	-
2	<i>dnc:natTurn</i>	<i>dnc:quickstep</i>	-

While SPARQL does not support “proper” negation, the desired effect can sometimes be achieved by filtering for unbound variables. The function `BOUND()`, applied to a variable name, returns true in a specific result row iff the variable is bound in this row. Negated with “!”, an appropriate filter condition retains only those result rows where the variable is unbound.

Example 3.3.26 (SPARQL Query (continued)). *The following SPARQL query returns all resources that have something belong to it and that do not have an average speed.*

```

1 PREFIX dnc: <http://www.dancedescriptions.net/>
2 SELECT ?x
3 WHERE {
4   _:a dnc:belongs_to ?x .
5   OPTIONAL { ?x dnc:average_speed ?y }
6   FILTER (!BOUND(?y))
7 }
```

Applied to the RDF graph from example 3.3.8, the query yields the following result:

<i>r</i>	<i>?x</i>
0	<i>dnc:vienneseWaltz</i>

Similar to SQL, SPARQL offers a `DISTINCT` selection keyword that prevents duplicate rows in a query result. Top-k queries can be formulated with the keywords `LIMIT` and `OFFSET`, where the former limits the number of rows in the query result, and the latter specifies the number of the first row to be returned.

SPARQL provides the option of returning an RDF graph as the result of a query, instead of a regular SPARQL query result. This is achieved by specifying the structure of the resulting graph in the form of a number of subgraph patterns. Syntactically, the keyword `SELECT` is replaced by the keyword `CONSTRUCT`, followed by the subgraph patterns for the result graph.

Example 3.3.27 (Construct SPARQL Query). *The following SPARQL query returns an RDF graph that reverses the direction of *dnc:belongs_to* relationships and calls them *dnc:has_figure* relationships.*

```

1 PREFIX dnc: <http://www.dancedescriptions.net/>
2 CONSTRUCT {
3   ?y dnc:has_figure ?x
4 } WHERE {
5   ?x dnc:belongs_to ?y
6 }

```

Applied to the RDF graph from example 3.3.8, the query yields the following result:

```

dnc:waltz          dnc:has_figure  dnc:natTurn .
dnc:vienneseWaltz dnc:has_figure  dnc:natTurn .

```

There are, however, several useful features that are not supported by SPARQL (yet). The first are recursive queries, where a subgraph pattern is recursively applied to instances of the same property. A workaround for this limitation is to declare a property as transitive and to use an inference engine to make the transitive closure of this property explicit. This will, however, greatly increase the size of the RDF graph and thus increase the complexity of pattern matching in the graph. In chapter 9, we will present an alternate approach to this problem.

Other features that SPARQL currently lacks are aggregation functions and other second-order constructs like count, minimum/maximum, or sum, as they are known from other query languages like SQL. SPARQL also does not provide means for data manipulation, such as insert, update, or delete operations.

Several proposed extensions to the SPARQL language exist that address some of these shortcomings [AMS07, KJ07], but none are part of the official standard specification.

While it is possible to map large parts of SPARQL onto expressions in the relational algebra [PB09], there are several caveats that make this mapping less than straight-forward. One of these caveats is that NULL values in the relational algebra do not have the same semantics as unbound variables in SPARQL: in the former, NULL is *never* a valid join partner, while in the latter, unbound variables are *always* valid join partners. Additionally, there are several SPARQL filter conditions, such as regular expressions, that have no equivalent in the relational algebra. Further details can be found in [Cyg05, ECTOO09].

3.3.6 RDF Frameworks

There exist several software frameworks that support RDF or even OWL.

The Jena framework [Jen, CDD⁺04] was originally developed by Hewlett Packard, then became an independent Open Source project, and is currently an Apache project. It provides an API and a data model for RDF data. On top of the RDF data model, there is an OWL data model, but its implementation is still rather inefficient at the time of this writing. Jena also provides SPARQL support and an interface for reasoning engines. It even includes several simple reasoning engines. Jena offers persistent RDF and OWL storage, both file-based and using relational database systems with a triple-based schema.

The Sesame framework [Ses, BKvH02] is also an Open Source project, primarily developed by the Aduna software company. It has similar capabilities as Jena, but an early evaluation showed the handling of persistent data to be more efficient in Jena [SF09].

The OWL API [OWL], another Open Source project, provides an efficient data model for OWL data, as well as an interface for external reasoning engines. It does not, however, offer support for SPARQL or for persistent storage other than in RDF files.

There exist many graph databases, but few are both mature and RDF-capable. The well-known graph database Neo4j [Neo] for example does not support RDF, nor does it provide

operations that are often needed in an RDF context, such as subgraph pattern matching. AllegroGraph [Fra] supports RDF and can even be used in combination with Jena or Sesame, but it is a commercial product and only a limited version is available for non-commercial use. Bigdata [Sys] only supports reasoning for RDFS, not OWL, which for example prohibits the use of transitive properties or complex classes. The Virtuoso Universal Server [Ope] is another commercial product, which provides only limited OWL reasoning that excludes, among other things, complex classes. Additional information about using graph databases for RDF can be found in [AG05].

In this thesis, we will make use of the Jena framework, because at the time of writing it supports the most relevant features, combined with an adequate performance.

3.4 Rule Languages

There exist several rule languages, with different expressive power and with different intended application domains. Most rules in these languages consist of a head and a body, usually called premise and conclusion or antecedent and consequent, where the premise contains a number of conditions, and the conclusion holds if these conditions hold.

Datalog is a rule language that is based on a subset of the programming language Prolog [Llo87], with semantics that are based on first order predicate logic [AHV95]. It is often used in conjunction with databases, where either the database serves as a data source for the Datalog rules, or where Datalog techniques are used to optimise database systems. There exist multiple Datalog implementations, both free and commercial.

RuleML, the Rule Markup Language, is a generic rule specification language [BTW]. It can be interpreted directly, or used as a rule interchange format.

The Semantic Web Rule Language (SWRL) is a combination of OWL DL and the Datalog sub-language of RuleML [HPSB⁺04]. It combines an OWL knowledge base with rules. However, it can be used to model constructs like subsumption of role chains (also called role-value maps, where one role chain subsumes another), which makes the combination undecidable [SS89]. Nonetheless, there exist several implementations for SWRL.

The JBoss Drools rule language is an object-oriented extension of production rules, using forward chaining inference [Pro07, PVB09]. It is based on Java, and rules can be compiled to Java programs by the Drools Expert rule engine. Rule premises consist of a set of conditions that are matched against a fact base of Java objects. Rule conclusions consist of a set of Java commands that are executed on objects matching the premise. It is possible to modify the fact base from the conclusion of a rule. Rule matching is implemented along the lines of the RETE algorithm [Doo95]. Conflict resolution, i.e., when multiple objects match the premise of a rule, or when one object matches multiple rules, is done by rule priority and last-in-first-out (LIFO) ordering [BBP07].

JBoss Drools supports rule specialisation in the following manner: Regard two rules $R_1 = (P_1, C_1)$ and $R_2 = (P_2, C_2)$ with premises P_1 and P_2 , and conclusions C_1 and C_2 . If P_2 is entailed by P_1 , then objects that match P_1 will also match P_2 , resulting in the application of both C_1 and C_2 . This can be regarded as a specialisation relationship, where R_2 is a specialisation of R_1 .

Example 3.4.1 (JBoss Drools Rule Specialisation). *Let R_1 be a rule that matches all Java objects of type C_{sup} . Let R_2 be a rule that matches all Java objects of type C_{sub} , where C_{sub} is a sub-class of C_{sup} . Then R_2 can be seen as a specialisation of R_1 .*

JBoss Drools also allows access to external tools and resources from rule specifications. For example, this makes it possible to include data from a knowledge base in the premise of a rule, or to process textual data with external libraries in the conclusion of a rule.

3.5 Model Checking

The following section is in large parts based on [Eme90, HR04, Wei08, WJF09, SWF11].

Model checking is the process of verifying or falsifying whether or not a given specification holds against a given model. In this section, we will briefly describe two formalisms for specifications, and how they can be used for model checking.

3.5.1 CTL

CTL (*Computation Tree Logic*) is a discrete, branching-time temporal logic [Eme90]. This means that it regards the concept of “time” as a tree of distinct states, starting with a root state that represents the “present”. The truth-value of a formula in temporal logic depends on the state in which it is evaluated.

Definition 3.5.1 (Syntax of CTL). *For an atomic formula p in propositional logic and CTL formulae ϕ_1 and ϕ_2 , the following constructs are CTL formulae:*

$false$	(false),
$true$	(true),
p	(atomic formula),
$\neg\phi_1$	(negation),
$\phi_1 \wedge \phi_2$	(conjunction),
$\phi_1 \vee \phi_2$	(disjunction),
$\phi_1 \rightarrow \phi_2$	(implication),
$AX\phi_1$	(next state on all paths),
$EX\phi_1$	(next state on one path),
$A[\phi_1 U \phi_2]$	(ϕ_1 until ϕ_2 on all paths), and
$E[\phi_1 U \phi_2]$	(ϕ_1 until ϕ_2 on one path).

Remark 3.5.2. *For a CTL formula ϕ , we use the following abbreviations:*

$AF\phi$	$= A[true U \phi]$	(future state on all paths),
$EF\phi$	$= E[true U \phi]$	(future state on one path),
$AG\phi$	$= \neg EF\neg\phi$	(all states on all paths), and
$EG\phi$	$= \neg AF\neg\phi$	(all states on one path).

Example 3.5.3 (Syntax of CTL). *For atomic formulae $DefDS$, $ExaDS$, $DefBT$, and $ExaBT$, the following are syntactically valid CTL formulae:*

$$DefDS \rightarrow EFExaDS \quad (3.1)$$

$$DefBT \rightarrow EFExaBT \quad (3.2)$$

$$AG EF(ExaDS \vee ExaBT) \quad (3.3)$$

The propositions $DefDS$, $ExaDS$, $DefBT$, and $ExaBT$, respectively, indicate if a definition (Def) or an example (Exa) for data structures (DS) or for binary trees (BT) exists in a particular state.

Formula 3.1 states that if there is a definition of a data structure, then there must also be an example of a data structure in a future state. Formula 3.2 states the same for binary trees. Formula 3.3 states that an example for either a data structure or a binary tree must be reachable from every state.

CTL formulae are evaluated on transition systems called CTL *temporal models*.

Definition 3.5.4 (CTL Temporal Model). *Let S be a set of states, let $R \subseteq S \times S$ be a left-total binary relation that assigns at least one successor to each state, and let L be a labelling function that assigns a set of atomic formulae to each state. Then $M = (S, R, L)$ is a CTL temporal model.*

Remark 3.5.5. *The semantics of temporal logics requires that $\forall s \in S : \exists s' \in S : (s, s') \in R$, i.e., that R is left-total. To ensure this, the temporal model may be extended with reflexive edges in the transition relation R for states with no “natural” successor.*

Example 3.5.6 (CTL Temporal Model). *Let $S = \{s_1, s_2, s_3, s_4, s_5\}$ be a set of states, each one representing a single chapter in a document.*

Let $R = \{(s_1, s_2), (s_2, s_3), (s_2, s_4), (s_4, s_5), (s_3, s_5), (s_5, s_5)\}$ be a successor relation on S that represents the links between these chapters.

Let $L = \{(s_1, \emptyset), (s_2, \{DefDS\}), (s_3, \{ExaDS\}), (s_4, \{DefBT, ExaBT\}), (s_5, \emptyset)\}$ be a labelling function that assigns the atomic formulae from example 3.5.3 to the states, representing which chapter contains definitions and examples about which topic.

Then $M = (S, R, L)$ is a CTL temporal model of a document. This is illustrated in figure 3.2 (a).

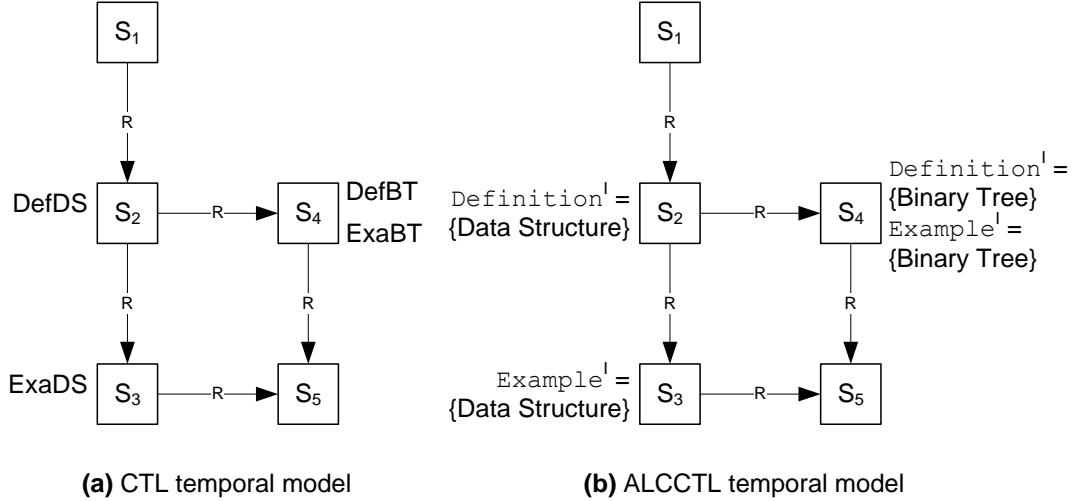


Figure 3.2: Temporal models

Definition 3.5.7 (CTL Path). *Let $M = (S, R, L)$ be a CTL temporal model. A CTL path π on M is an infinite sequence of states, where each state is in an R -relation with the next state in the sequence. Formally, $\pi = (s_0, s_1, s_2, \dots)$ with $(s_i, s_{i+1}) \in R$ for $i \in \mathbb{N}$. The set of CTL paths starting in a specific state $s \in S$ is denoted by π_s .*

Example 3.5.8 (CTL Path). *Let $M = (S, R, L)$ be the temporal model from example 3.5.6. Then $\pi_{s_1} = \{(s_1, s_2, s_3, s_5, s_5, \dots), (s_1, s_2, s_4, s_5, s_5, \dots)\}$ is the set of CTL paths starting in s_1 .*

Definition 3.5.9 (Semantics of CTL). *Let $M = (S, R, L)$ be a CTL temporal model, $s \in S$ a state, p an atomic formula, and ϕ_1 and ϕ_2 CTL formulae. Then whether M satisfies a formula ϕ in s , formally $M, s \models \phi$, is defined inductively as follows:*

$M, s \not\models \text{false}$	
$M, s \models \text{true}$	
$M, s \models p$	<i>iff</i> $p \in L(s)$
$M, s \models \neg\phi_1$	<i>iff</i> $M, s \not\models \phi_1$
$M, s \models \phi_1 \wedge \phi_2$	<i>iff</i> $M, s \models \phi_1$ and $M, s \models \phi_2$
$M, s \models \phi_1 \vee \phi_2$	<i>iff</i> $M, s \models \phi_1$ or $M, s \models \phi_2$
$M, s \models \phi_1 \rightarrow \phi_2$	<i>iff</i> $M, s \models \neg\phi_1 \vee \phi_2$
$M, s \models \text{AX}\phi_1$	<i>iff</i> $\forall (s, s_1, \dots) \in \pi_s : M, s_1 \models \phi_1$
$M, s \models \text{EX}\phi_1$	<i>iff</i> $\exists (s, s_1, \dots) \in \pi_s : M, s_1 \models \phi_1$
$M, s \models \text{A}[\phi_1 \text{ U } \phi_2]$	<i>iff</i> $\forall (s, s_1, \dots) \in \pi_s : \exists i \in \mathbb{N} :$ $(M, s_i \models \phi_2 \text{ and } \forall j \in \{0, \dots, i-1\} : M, s_j \models \phi_1)$
$M, s \models \text{E}[\phi_1 \text{ U } \phi_2]$	<i>iff</i> $\exists (s, s_1, \dots) \in \pi_s : \exists i \in \mathbb{N} :$ $(M, s_i \models \phi_2 \text{ and } \forall j \in \{0, \dots, i-1\} : M, s_j \models \phi_1)$

Example 3.5.10 (Semantics of CTL). *Let $M = (S, R, L)$ be the temporal model from example 3.5.6. Recall the formulae from example 3.5.3. Then in s_2 , M satisfies the first two formulae (3.1 and 3.2), but not the third (3.3).*

For formula 3.1, this can be seen easily, because

$$\begin{aligned}
M, s_2 \models \text{DefDS} \rightarrow \text{EFExaDS} &\Leftrightarrow M, s_2 \models \neg\text{DefDS} \vee \text{EFExaDS} \\
&\Leftrightarrow M, s_2 \models \neg\text{DefDS} \text{ or } M, s_2 \models \text{EFExaDS} \\
&\Leftrightarrow M, s_2 \not\models \text{DefDS} \text{ or } M, s_2 \models \text{E}[\text{true U ExaDS}]
\end{aligned}$$

$M, s_2 \not\models \text{DefDS}$ is obviously false for M . But $M, s_2 \models \text{E}[\text{true U ExaDS}]$ is true because $(s_2, s_3, \dots) \in \pi_{s_2}$ is a path that fulfills the requirement with $i = 3$.

Formula 3.2 can be verified in a similar manner. M does not satisfy formula 3.3 because it has to hold in every state (AG) but it does not hold in s_5 .

Specifications represented as CTL formulae can be verified against a CTL temporal model by model checking.

Definition 3.5.11 (CTL Model Checking Problem). *Let $M = (S, R, I)$ be a CTL temporal model such that S is finite. Let ϕ be a CTL formula. The CTL model checking problem is to decide for all $s \in S$ if $M, s \models \phi$.*

An alternate version of the CTL model checking problem is to restrict it to a (small) set of starting states $S_0 \subseteq S$ and to decide for all $s \in S_0$ if $M, s \models \phi$.

There exist sound and complete algorithms for the CTL model checking problem. The complexity of the problem has an upper bound of $O(|\phi| \cdot (|S| + |R|))$, where $|\phi|$ is the number of sub-expressions in ϕ .

Example 3.5.12 (CTL Model Checking Problem). *Let $M = (S, R, L)$ be the temporal model from example 3.5.6. Recall the formulae from example 3.5.3.*

It can be shown that

- ▶ $\forall s \in S : M, s \models \text{DefDS} \rightarrow \text{EFExaDS}$
- ▶ $\forall s \in S : M, s \models \text{DefBT} \rightarrow \text{EFExaBT}$
- ▶ $\forall s \in S : M, s \not\models \text{AG EF}(\text{ExaDS} \vee \text{ExaBT})$

3.5.2 \mathcal{ALCCTL}

The temporal description logic \mathcal{ALCCTL} was introduced in [Wei08]. It is a combination of the description logic \mathcal{ALC} and the temporal logic CTL.

Definition 3.5.13 (Syntax of \mathcal{ALCCTL}). *For an atomic concept A (cf. definition 3.2.3), an atomic role R (cf. definition 3.2.5), and \mathcal{ALCCTL} concepts C_1 and C_2 , the following constructs are \mathcal{ALCCTL} concepts:*

\perp	(bottom concept),
\top	(top concept),
A	(atomic concept),
$\neg C_1$	(complement),
$C_1 \sqcap C_2$	(intersection),
$C_1 \sqcup C_2$	(union),
$\forall R.C_1$	(universal quantification),
$\exists R.C_1$	(existential quantification),
AXC_1	(next state on all paths),
EXC_1	(next state on one path),
$A[C_1 \text{ U } C_2]$	(C_1 until C_2 on all paths), and
$E[C_1 \text{ U } C_2]$	(C_1 until C_2 on one path).

For \mathcal{ALCCTL} concepts C_1 and C_2 , and for \mathcal{ALCCTL} formulae ϕ_1 and ϕ_2 , the following constructs are \mathcal{ALCCTL} formulae:

<i>false</i>	(false),
<i>true</i>	(true),
$C_1 \sqsubseteq C_2$	(concept subsumption),
$C_1 \equiv C_2$	(concept equivalence),
$\neg \phi_1$	(negation),
$\phi_1 \wedge \phi_2$	(conjunction),
$\phi_1 \vee \phi_2$	(disjunction),
$AX\phi_1$	(next state on all paths),
$EX\phi_1$	(next state on one path),
$A[\phi_1 \text{ U } \phi_2]$	(ϕ_1 until ϕ_2 on all paths), and
$E[\phi_1 \text{ U } \phi_2]$	(ϕ_1 until ϕ_2 on one path).

Remark 3.5.14. For an \mathcal{ALCCTL} concept C and an \mathcal{ALCCTL} formula ϕ , we use the following abbreviations:

AFC	$= A[\top \text{ U } C]$	(future state on all paths),
EFC	$= E[\top \text{ U } C]$	(future state on one path),
AGC	$= \neg EF\neg C$	(all states on all paths),
EGC	$= \neg AF\neg C$	(all states on one path),
$AF\phi$	$= A[\top \text{ U } \phi]$	(future state on all paths),
$EF\phi$	$= E[\top \text{ U } \phi]$	(future state on one path),
$AG\phi$	$= \neg EF\neg\phi$	(all states on all paths), and
$EG\phi$	$= \neg AF\neg\phi$	(all states on one path).

Example 3.5.15 (Syntax of \mathcal{ALCCTL}). *Syntactically valid \mathcal{ALCCTL} formulae are for example*

$$\text{Definition} \sqsubseteq EF\text{Example} \quad (3.4)$$

$$AG\neg(EF\text{Example} \sqsubseteq \perp) \quad (3.5)$$

where *Definition* and *Example* are atomic concepts that represent all definitions and examples in a particular state.

Formula 3.4 states that if there is a definition, then there must also be an example with the same topic in a future state. Formula 3.5 states that an example must be reachable from every state.

\mathcal{ALCCTL} formulae are evaluated on transition systems called \mathcal{ALCCTL} temporal models.

Definition 3.5.16 (\mathcal{ALCCTL} Temporal Model). *Let S be a set of states, let $R \subseteq S \times S$ be a left-total binary relation that assigns at least one successor to each state, and let LI be the set of \mathcal{ALC} interpretations (cf. definition 3.2.28). Then I is a function $S \rightarrow LI : I(s) = (\Delta^I, ()^{I(s)})$ that assigns a description logics interpretation $(\Delta^I, ()^{I(s)})$ to each state. Then $M = (S, R, I)$ is an \mathcal{ALCCTL} temporal model.*

Example 3.5.17 (\mathcal{ALCCTL} Temporal Model). *Let $S = \{s_1, s_2, s_3, s_4, s_5\}$ be a set of states, each one representing a single chapter in a document.*

Let $R = \{(s_1, s_2), (s_2, s_3), (s_2, s_4), (s_4, s_5), (s_3, s_5), (s_5, s_5)\}$ be a successor relation on S that represents the links between these chapters.

Let $\Delta^I = \{\text{Data Structure, Binary Tree}\}$ be an interpretation domain.

Let $I = \{(s_1, (\Delta^I, ()^{I(s_1)})), (s_2, (\Delta^I, ()^{I(s_2)})), (s_3, (\Delta^I, ()^{I(s_3)})), (s_4, (\Delta^I, ()^{I(s_4)})), (s_5, (\Delta^I, ()^{I(s_5)}))\}$ be a function that assigns an \mathcal{ALC} model to each state.

Let $\text{Definition}^{I(s_2)} = \{\text{Data Structure}\}$, $\text{Example}^{I(s_3)} = \{\text{Data Structure}\}$, $\text{Definition}^{I(s_4)} = \{\text{Binary Tree}\}$, and $\text{Example}^{I(s_4)} = \{\text{Binary Tree}\}$.

Then $M = (S, R, I)$ is an \mathcal{ALCCTL} temporal model of a document. This is illustrated in figure 3.2 (b).

Definition 3.5.18 (\mathcal{ALCCTL} Path). *An \mathcal{ALCCTL} path on an \mathcal{ALCCTL} temporal model is defined analogue to a CTL path on a CTL temporal model.*

Example 3.5.19 (\mathcal{ALCCTL} Path). *Let $M = (S, R, I)$ be the temporal model from example 3.5.17. Then $\pi_{s_1} = \{(s_1, s_2, s_3, s_5, s_5, \dots), (s_1, s_2, s_4, s_5, s_5, \dots)\}$ is the set of \mathcal{ALCCTL} paths starting in s_1 .*

Definition 3.5.20 (Semantics of \mathcal{ALCCTL}). *Let $M = (S, R, I)$ be an \mathcal{ALCCTL} temporal model, $s \in S$ a state, A an atomic concept, and C_1 and C_2 \mathcal{ALCCTL} concepts. Then the semantics of \mathcal{ALCCTL} concepts C with respect to M and s , formally $(M, s)(C)$, are defined inductively as follows:*

$$\begin{aligned}
(M, s)(\perp) &= \emptyset \\
(M, s)(\top) &= \Delta^I \\
(M, s)(A) &= A^{I(s)} \\
(M, s)(\neg C_1) &= \Delta^I \setminus (M, s)(C_1) \\
(M, s)(C_1 \sqcap C_2) &= (M, s)(C_1) \cap (M, s)(C_2) \\
(M, s)(C_1 \sqcup C_2) &= (M, s)(C_1) \cup (M, s)(C_2) \\
(M, s)(\forall R.C_1) &= (M, s)(\neg \exists R.\neg C_1) \\
(M, s)(\exists R.C_1) &= \{a \in \Delta^I \mid \exists b \in \Delta^I : (a, b) \in R^{I(s)} \wedge b \in (M, s)(C_1)\} \\
(M, s)(\text{AX}C_1) &= \bigcap_{s' \in \{s' \in S \mid (s, s') \in R\}} (M, s')(C_1) \\
(M, s)(\text{EX}C_1) &= \bigcup_{s' \in \{s' \in S \mid (s, s') \in R\}} (M, s')(C_1) \\
(M, s)(\text{A}[C_1 \text{ U } C_2]) &= \bigcap_{(s, s_1, \dots) \in \pi_s} \{a \in \Delta^I \mid \exists i \in \mathbb{N} : a \in (M, s_i)(C_2) \wedge \\
&\quad \forall j \in \{0, \dots, i-1\} : a \in (M, s_j)(C_1)\} \\
(M, s)(\text{E}[C_1 \text{ U } C_2]) &= \bigcup_{(s, s_1, \dots) \in \pi_s} \{a \in \Delta^I \mid \exists i \in \mathbb{N} : a \in (M, s_i)(C_2) \wedge \\
&\quad \forall j \in \{0, \dots, i-1\} : a \in (M, s_j)(C_1)\}
\end{aligned}$$

If the model M is clear from the context, we will simply write $C^{I(s)}$ instead of $(M, s)(C)$.

Let $M = (S, R, I)$ be an \mathcal{ALCCTL} temporal model, $s \in S$ a state, C_1 and C_2 \mathcal{ALCCTL} concepts, and ϕ_1 and ϕ_2 \mathcal{ALCCTL} formulae. Then whether M satisfies a formula ϕ in s , formally $M, s \models \phi$, is defined inductively as follows:

$$\begin{aligned}
M, s &\not\models \text{false} \\
M, s &\models \text{true} \\
M, s &\models C_1 \sqsubseteq C_2 && \text{iff } C_1^{I(s)} \subseteq C_2^{I(s)} \\
M, s &\models C_1 \equiv C_2 && \text{iff } C_1^{I(s)} = C_2^{I(s)} \\
M, s &\models \neg\phi_1 && \text{iff } M, s \not\models \phi_1 \\
M, s &\models \phi_1 \wedge \phi_2 && \text{iff } M, s \models \phi_1 \text{ and } M, s \models \phi_2 \\
M, s &\models \phi_1 \vee \phi_2 && \text{iff } M, s \models \phi_1 \text{ or } M, s \models \phi_2 \\
M, s &\models \text{AX}\phi_1 && \text{iff } \forall (s, s_1, \dots) \in \pi_s : M, s_1 \models \phi_1 \\
M, s &\models \text{EX}\phi_1 && \text{iff } \exists (s, s_1, \dots) \in \pi_s : M, s_1 \models \phi_1 \\
M, s &\models \text{A}[\phi_1 \text{ U } \phi_2] && \text{iff } \forall (s, s_1, \dots) \in \pi_s : \exists i \in \mathbb{N} : \\
&&& (M, s_i \models \phi_2 \text{ and } \forall j \in \{0, \dots, i-1\} : M, s_j \models \phi_1) \\
M, s &\models \text{E}[\phi_1 \text{ U } \phi_2] && \text{iff } \exists (s, s_1, \dots) \in \pi_s : \exists i \in \mathbb{N} : \\
&&& (M, s_i \models \phi_2 \text{ and } \forall j \in \{0, \dots, i-1\} : M, s_j \models \phi_1)
\end{aligned}$$

Example 3.5.21 (Semantics of \mathcal{ALCCTL}). Let $M = (S, R, I)$ be the temporal model from example 3.5.17. Recall the formulae from example 3.5.15. Then in s_2 , M satisfies the first formula (3.4), but not the second (3.5).

For formula 3.4, this can be seen easily, because

$$\begin{aligned}
M, s_2 &\models \text{Definition} \sqsubseteq \text{EFExample} \Leftrightarrow \\
M, s_2 &\models \text{Definition}^{I(s_2)} \subseteq \text{EFExample}^{I(s_2)} \Leftrightarrow \\
M, s_2 &\models \{\text{Data Structure}\} \subseteq \text{E}[\text{T U Example}]^{I(s_2)}
\end{aligned}$$

This is true, because $\text{E}[\text{T U Example}]^{I(s_2)}$ evaluates to all **Example** interpretations in all states that follow s_2 , resulting in the set $\{\text{Data Structure}, \text{Binary Tree}\}$.

M does not satisfy formula 3.5 because it has to hold in every state (**AG**) but it does not hold in s_5 .

Specifications represented as \mathcal{ALCCTL} formulae can be verified against an \mathcal{ALCCTL} temporal model by model checking.

Definition 3.5.22 (\mathcal{ALCCTL} Model Checking Problem). Let $M = (S, R, I)$ be an \mathcal{ALCCTL} temporal model such that S and Δ^I are finite. Let ϕ be an \mathcal{ALCCTL} formula. The \mathcal{ALCCTL} model checking problem is to decide for all $s \in S$ if $M, s \models \phi$.

An alternate version of the \mathcal{ALCCTL} model checking problem is to restrict it to a (small) set of starting states $S_0 \subseteq S$ and to decide for all $s \in S_0$ if $M, s \models \phi$.

There exists a sound and complete algorithm for the \mathcal{ALCCTL} model checking problem. The complexity of the problem has an upper bound of $O(|\phi| \cdot (|S| + |R|) \cdot |\Delta^I|^2)$, where $|\phi|$ is the number of sub-expressions in ϕ .

Example 3.5.23 (\mathcal{ALCCTL} Model Checking Problem). Let $M = (S, R, I)$ be the temporal model from example 3.5.17. Recall the formulae from example 3.5.15.

It can be shown that

- ▶ $\forall s \in S : M, s \models \text{Definition} \sqsubseteq \text{EFExample}$
- ▶ $\forall s \in S : M, s \not\models \text{AG}\neg(\text{EFExample} \sqsubseteq \perp)$

Chapter 4

Modelling Digital Documents

In this chapter, we will examine and define some of the terminology relevant for this thesis (section 4.1), including the nature of documents. We will then formalise our understanding of documents and discuss options for modelling digital documents. In section 4.2, we first define a formal model for documents that includes semantic information. We then transfer this model to the domain of process descriptions.

We will attempt to give a clear notion of a document instead of relying on an implicit, vague, or exceedingly narrow definition. Our novel definition will be independent of any specific document type and format, and it will encompass both complex structural and semantic information.

4.1 Terminology

4.1.1 What is a Document?

In the domain of computer science, the term *document* is often used but rarely defined. Authors rely on the intuitive understanding of their readers, and sometimes guide that intuition by listing a number of technical properties that their understanding of a document requires [SFC98, Jel02, ESS05]. Outside of the computer science domain, however, the discussion about the nature of a document has a long tradition and is a controversial issue.

The common ground in this controversy is the symbolic nature of documents – they contain words, thoughts and ideas indicated by symbols. The persistence of these symbols on the document artefact has been used to store and to distribute information in a consistent, i.e., unchanged, way (cf. figure 4.1). Traditionally, a document looked alike for every reader¹ – an important property for disseminating information. However, with the advent of digital documents and especially hypertexts (cf. definition 4.1.1), this is no longer generally the case. On the one hand, the presentation of digital documents can change with the circumstances. For example, a web page might be rendered differently for viewers from different countries. On the other hand, a hypertext offers the reader the choice of many different “forking paths” (a notion introduced in [Bor48]), reading paths by which the document can be traversed (cf. definition 4.1.5). Different readers – or even a single reader – following different paths through a document will likely experience a very different presentation of the same hypertext [Joy87, Mou92, Esp97].

Definition 4.1.1 (Hypertext). *A hypertext $H = (P, p_0, l)$ consists of*

¹This does not dispute that different readers can *interpret* the content of the document in fundamentally different ways.



Figure 4.1: Left: Legal document with seal impression, Babylon, 414 BCE. Right: Seal and seal impression, Babylon, 5th/6th century BCE. Pergamon Museum, Berlin, Germany.

P a non-empty set of atomic pages,
 $p_0 \in P$ a starting page, and
 $l \subseteq P \times P$ a binary relation that models links, i.e., references from one page to another.

From the starting page p_0 , every page $p \in P \setminus \{p_0\}$ must be reachable through l , either directly or indirectly.

Definition 4.1.1 is a formalisation of the properties and characteristics of a hypertext as it is informally described or implied in various texts [Bus45, Nel80, Bol01].

Example 4.1.2 (Hypertext). Let

P = {“Introduction”, “Chapter 2”, “Chapter 3”, “Chapter 4”, “Conclusion”},
 p_0 = “Introduction”, and
 l = {(“Introduction”, “Chapter 2”), (“Chapter 2”, “Chapter 3”), (“Chapter 2”, “Chapter 4”), (“Chapter 3”, “Conclusion”), (“Chapter 4”, “Conclusion”).}

Then $H = (P, p_0, l)$ is a hypertext.

For reasons of clarity, each element $p \in P$ only represents a page in this example, instead of actually being one. In other words, the elements of P are symbolic representations of entire “pages” or text blocks. For example, “Introduction” represents an actual introduction of a document, including text fragments or images.

Remark 4.1.3. Note that, in keeping with the usual understanding of hypertexts in the literature, this definition ignores a possible sub-structuring of pages. It regards each page as an atomic entity, not as a complex text or even a hypertext in its own right.

Remark 4.1.4. Also note that this definition implies that there must be a page $p_i \in P$ for every page $p_j \in P \setminus \{p_0\}$ such that $(p_i, p_j) \in l$, i.e., every page that is not the starting page must have a predecessor. This excludes the possibility of a partitioning of the hypertext, where non-empty subsets of P are completely separated from the rest of the hypertext. In particular, it excludes the possibility of “orphaned”, i.e., unreachable, pages.

Definition 4.1.5 (Reading Path (cf. [Wei08]). *A reading path r on a hypertext $H = (P, p_0, l)$ is a non-empty, possibly infinite sequence of pages (p_0, p_1, p_2, \dots) , for which holds that $(p_i, p_{i+1}) \in l$, for $i \in \mathbb{N}$.*

Example 4.1.6 (Reading Path). *Let H be the hypertext from example 4.1.2. Then (“Introduction”, “Chapter 2”, “Chapter 3”, “Conclusion”) is a reading path on H .*

Remark 4.1.7. *Using definition 4.1.5, we can put the reachability condition from definition 4.1.1 more concisely:*

$$\forall p \in P \setminus \{p_0\} : \exists r = (p_0, \dots, p)$$

where r is a reading path.

The controversy about documents focusses on whether a document is *fixed* (unchanging) or *fluid* (changing). The notion of fixity or fluidity describes a state of being, not a technical property. It is not enforced through some technical mechanism, but it is an inherent aspect of a document [Bol01]. For example, a marble inscription as in figure 4.2 is literally set in stone and therefore fixed.



Figure 4.2: Fragment of a marble tablet with an Arabic line of text, Egypt, 10th/11th century CE. Pergamon Museum, Berlin, Germany.

Non-digital writing usually entails putting marks onto or into some material, which gives it a certain permanence. Based on this, Bolter and others argue that traditional documents are entirely fixed. Digital documents, in contrast, can not only be changed easily because of their lack of physical permanence, but their multitude of reading paths constantly change the way in which they are perceived, making them inherently fluid. Indeed, the reader is given some of the power – and the burden – that is traditionally reserved for the author: to determine the order in which a document is being read. [SW88, Bol01]

Bolter further argues that the technological basis for documents directly influences the nature of these documents: handwriting or print lead to fixed documents, which – because of their

permanence – in turn promote a rigid canon of documents for each domain that are the most relevant there and that define the standard for their domain. These definite works are created by a small number of authors who dominate the domain, such as Augustine of Hippo or Thomas Aquinas. The dynamic nature of digital documents, says Bolter, also leads to the abolition of fixed canons and – by diluting the separation between reader and author – leads to curbing the influence of dominant authors. [Bol01]

The premise that technology shapes society is rejected by the school of *social construction of technology*. Its adherents suppose the opposite effect, namely that society shapes technology [BHP87]. This directly contradicts Bolter’s thesis that the technological basis of documents promotes a specific way of working with them. It can also be argued that the continued existence of authors who dominate their field weakens Bolter’s arguments.

Levy challenges Bolter’s assertion that the introduction of digital documents signals the transition from fixed to fluid documents. Instead, he argues that all documents are both fixed and fluid: any document can be changed, independent of its technological basis; and any document can remain unchanged, even if it is in digital form. [Lev94]

“ (...) all documents, regardless of technology, are fixed and fluid – fixed at certain times and fluid at others. Indeed, they exist in perpetual tension between these two poles – fixing content for periods of time to serve particular human needs, and changing as necessary to remain in synch with the changing circumstances of the world.” [Lev94]

Thus, following Levy, it is not the property of fixity or fluidity that is changed by the advent of digital documents, but rather the *rhythm of change* between the two. He also objects to Bolter’s argument on the ground that the “need for fixity is a basic human concern”, which society will not be willing to do without. [Lev94]

However, arguably the crux of the controversy lies only partially in whether or not a document is fixed. It can also be seen as a fundamental discrepancy in the notion of a document as seen by Levy and Bolter, as well as their respective adherents. Levy poses that fixity or fluidity is not invariant: a document can be in a fluid state for some time and then become fixed [LM95]. For example, a document can be fluid during its creation and editing and become fixed after it has been published. This clearly indicates that for Levy, the notion of a document encompasses its entire life cycle, including the process of its creation. Bolter, on the other hand, is much more concerned with the “finished” document, for example once it is published in some form.

Concerning published physical documents, as shown in figure 4.3, Bolter and Levy actually agree that they are fixed. However, it is less the physical permanence of the document that determines its fixity, but rather the constraints imposed by the position in its life cycle: a document, once published, is usually very hard to retract. The established way to incorporate redactions, corrections, extensions or other changes into a document is to publish a new version or edition. This new version would constitute a new and distinct document in Bolter’s sense. It is unclear how Levy would regard it.

On digital documents however, both authors clearly disagree. One point of contention is that Bolter denies digital documents any kind of permanence, even temporary permanence in the sense of non-modifiability. This stance has obviously been overtaken by technological advances in cryptography and digital signatures, as well as digital media like compact discs that provide protection against modification. But it has also been proven wrong by societal developments. Many digital documents, while not technically secured against changes, are inherently fixed either by law or by convention. For example, the digital document stating the terms and conditions of using a specific piece of software (licence agreement) cannot be changed without prior no-



Figure 4.3: Collection of books. Pražský hrad, Prague, Czech Republic.

tice. The digital documents published by the World Wide Web Consortium (W3C) describing various versions of web standards are widely regarded as immutable: changes to the standard require a new version of the standard and thus a new document, leaving the previous documents unchanged.

In part, this disagreement may also be caused by the stages of the life cycle that are not always clearly distinct for digital documents. For instance, there is not always an explicitly “published” version – sometimes, the draft version of a document implicitly becomes the final document when no more changes are made. For other digital documents, especially social collaboration documents like Wikipedia articles, there never is a finished document, only the current (draft) version.

Another point of conflict is that Bolter and Levy regard different aspects of digital documents. While Levy’s focus stays on the document itself, Bolter includes how a document is perceived by its readers into his analysis.

The points of contention in this controversy show that a clearly defined notion of a document is not only instrumental but essential for any discourse about documents, and must also be the first step in describing document processing.

4.1.2 Our Notion of a Document

In this thesis, we will limit our focus on purely digital documents, and we exclude documents that are only meant for machine consumption, because they follow different rules. For now, we will also limit our discussion to fixed states of documents, i.e., to snapshots in the life cycle of documents.

We will first regard the notion of a document on the basis of various examples, before we will try to identify some common aspects of documents. From an analysis of these common aspects we will develop a property-based description of a document. This description must obviously be

limited as it can only cover types of documents that are currently known, i.e., it cannot take new and emerging types of documents into account.

We will also develop a more formal notion of a document, which will be harmonised with the property-based description.

Some common *types of digital documents* are simple text files, office documents (rich text documents, spreadsheets, presentations), electronic books (e-books), and web pages. It is also possible to regard images (bitmap and vector graphics), audio files (wave-based and note-based audio) and video files as documents.

Text files contain lists of words, separated by whitespace. They can cover any topic, but their lack of sophisticated features like text formatting and visual structuring usually restricts their use to simple note keeping or as a simple exchange format.

Rich text documents provide the ability to format text, and thus to visually structure a document. While linear in nature, they often allow the definition of cross-references within the same text, and of references to external resources. The same is true for presentation documents like Microsoft Powerpoint documents, yet presentations focus more on the “page” as the primary structural element. In rich text documents, the page arrangement can easily change with changes in text content or formatting. In presentations, the page structure remains outwardly constant, even if the pages’ content changes. Spreadsheets focus less on text flow than on structured data. While in other documents text directionality (e.g., left-to-right) is well-defined, this is not necessarily the case in spreadsheets. They may, for example, contain tables that can be read either line-by-line or column-by-column.

Office documents often contain letters, reports, records or notes in a professional (e.g., business) or private context. It is the nature of such documents that they often only cover a small number of related topics.

Electronic books often mimic some of the attributes of their non-digital archetype, with most current e-books being electronic adaptations of printed books. They usually have a primarily linear structure, sometimes broken by textual or even technical cross-references. They often provide a page-based visualisation, but the actual instances of a “page” may depend on the visualisation context, for example on the size of the visualisation surface. So in different contexts, e.g., on different reading devices, the partitioning of an electronic book into pages may change dynamically.

E-books may cover virtually any topic, ranging from fiction to non-fiction, but a single e-book usually does not cover a wide range of topics unless these topics are closely related. A brief survey of 100 randomly selected electronic books sold by various online retailers supports this observation. The vast majority of non-fiction books either cover a single topic, or a small number of related topics. Virtually all fiction books either cover a single narrative, or a small number of narratives that are somehow connected. Notable exceptions are artificial aggregations like collections of short stories and reference books like dictionaries or codes of law.

Web pages are similar to rich text documents in that they may contain formatted text and multimedia elements. When regarding collections of web sites instead of single sites, they exhibit a page-centric structure similar to presentations. However, other than in a presentation, a single page of a multi-page web document is not constrained in terms of size. For web pages, references (links) are usually far more important than for other document types. References to other pages of the same document and references to other sections on the same page can result in a very complex structure.

Like electronic books, web pages can cover any topic. Most pages, however, are dedicated to either a single topic, or to a number of topics belonging to a single domain. The latter is also often true for blogs, which are usually centred around a specific theme – a topic, a domain, or even a person. The most notable exceptions here are content aggregation sites that collect

information from various other pages, and news sites. But even these sites have “actuality” as a common theme.

Most of these document types have a single starting point, from which all parts of the document can be reached. For web pages, it is possible to have pages that are completely separated from the starting page. These separated pages, however, arguably form separate documents of their own, instead of belonging to the original document. Their “physical” proximity on the same server or under a similar URL is their only direct and obvious connection.

Images can contain technical drawings like blueprints, diagrams or maps, as well as photos, paintings, or abstract art. Most of these image documents are focussed on a single topic or theme, but are structured around multiple objects. Unlike text-based documents, images can lack a sense of progression, i.e., they are sometimes not meant to be “read” in any particular order but to be regarded as a whole. Image-based stories and sequential diagrams are notable exceptions here.

Audio documents usually contain either recorded speech, such as political arguments, audio books or dictations, or music, either in the form of recordings or as synthesised notes. While speech is inherently linear, music can have a more complex structure. In sheet music, references like “da capo” can create a complex order in which a musician is meant to follow the notes. Of course, any musical recording is an instantiation of such an order as *interpreted* by musicians, and is therefore linear again.

Videos can also cover speech (e.g., “talking head” video), but also entire stories or more complex narrative structures (see below).

On a technical level, documents can be broken into multiple files and these files can be written in different formats. In the case of multiple documents, each consisting of multiple files, this leads to some interesting questions: where does one document end, and another begin? Can such a separation be defined, and how? Must such a separation be a strict partitioning, or can there be overlap, i.e., can two documents both contain one and the same file?

Any hard separation between sets of files can be identified based on one or more of the following aspects:

- ▶ random chance,
- ▶ file format,
- ▶ file name,
- ▶ file content,
- ▶ file metadata, including author and time stamp data, or
- ▶ human choice, for example that of the author.

Randomness in documents is primarily in the purview of art and avant-garde literary forms [Que61], which we will not regard here.

Since file name and format are either chosen by the author, which would fall back on the aspect of human choice, or can otherwise be influenced by some arbitrary circumstances, they make poor criteria for separating documents. As our understanding of documents so far requires them to deal with a number of related topics, the file content seems like a good criterion. However, while the content can give a good indication of how closely two or more files are related, this is necessarily a gradual separation and is therefore in most cases still unsuitable for defining a partitioning. File metadata can help in this regard, by partitioning the file set by author or by creation date. However, a single author can obviously write multiple documents, and multiple authors can create a single document.

So all of these aspects can give indications about the beginning and end of documents across multiple files, but the final decision must be left to human choice – usually that of the author, or sometimes that of the reader. An author may also choose to include a single file in multiple documents, without upsetting the integrity of these documents².

Yet there are cases of documents where the author is not able or willing to define the borders of a document. For example, the Wikipedia project consists of scores of articles on a multitude of topics, grouped by a hierarchy of categories, written by a host of authors. In this context, what is a document? A single article? A group of articles on closely related topics? All articles belonging to a certain category? All articles taken together? Clearly, the examples presented at the beginning of this section allow for all of these interpretations. So which interpretation is correct?

The answer obviously depends on the context and on the requirements of the person using the document, in this example Wikipedia: someone only interested in a very specific topic will likely regard a single article as sufficient and will consider other articles to be too unrelated to be part of the same “document”. Someone interested in a broader topic will probably regard a group of closely related articles as her “document” entity, and someone with an even broader interest might consider all articles on a certain domain to be a single “document”. This means that each interpretation given above is correct in a particular context.

This conclusion can be transferred to hypertexts in general. A single page $p \in P$ in a hypertext $H = (P, p_0, l)$ can be regarded as a self-sufficient document. A partial hypertext H_r (cf. our definition 4.1.8) can also be considered a document. And finally, an entire hypertext H can be considered a document as well. Which of these choices is actually made is immaterial in the context of this thesis, as long as the resulting document is a coherent whole.

Definition 4.1.8 (Partial Hypertext). *A hypertext $H = (P, p_0, l)$ can be reduced to a partial hypertext $H_r = (P_r, p_{0r}, l_r)$, where $P_r \subseteq P$, $p_{0r} \in P_r$, and $l_r : P_r \times P_r \subseteq l$ is a reduction of l onto P_r :*

$$(p_i, p_j) \in l_r \text{ iff } (p_i, p_j) \in l \wedge p_i, p_j \in P_r$$

A partial hypertext must also be a hypertext, so the requirements from definition 4.1.1 also apply to H_r :

$$\forall p \in P_r \setminus \{p_{0r}\} : \exists r = (p_{0r}, \dots, p)$$

where r is a reading path.

Description 4.1.9 (Document). *To summarise our perception of a document up to this point, we list its relevant properties.*

1. *A document is in digital form,*
2. *it can be displayed in a human-readable way, i.e., it has a symbolic representation,*
3. *its content covers one or more related topics,*
4. *its content elements are in a (possibly non-linear) order,*
5. *it has a distinct starting point from which all content elements can be reached, and*

²This does, however, necessitate a consistent policy about document management, otherwise an update in one document could inadvertently lead to changes in other documents.

6. *its content elements can be (formatted) text, image, audio, or video (this list is non-exhaustive).*

In particular, a document can be seen as a hypertext, where the content elements are perceived as hypertext pages.

4.1.3 Interactivity in Documents

Interactivity is generally defined as the possibility for a user to influence or control a system [Haa02], resulting in a reciprocal action [Goe04]. This precipitates the need for a two-way communication: in addition to a communication channel from the system to the user that carries for example the visualisation of a document's contents, a further channel is required for carrying user input back to the system [Pag00].

Interaction requires some action by a user. It is, however, not necessarily the user who initiates the interaction. The role of initiator can also be filled by the system, making the user's action a re-action. In a fully interactive system, every action or reaction by one side (either the system or the user) can be followed by a new reaction from the other party.

As discussed earlier, hypertext documents can offer a reader multiple different reading paths. This already is a simple form of interactivity, where the system provides the user with a choice (namely, a set of options on where to continue reading), the user selects one of these options (action), and the system responds by visualising the appropriate document part (reaction).

A more powerful way of interactive document navigation exists if the application that provides the visualisation for the document also provides a search mechanism. The user can at any time enter one or more search terms (action), the system provides a list of document locations relevant for these terms (reaction), the user selects one location (reaction), and the system visualises the selected document part (reaction).

Another form of interactive documents are forms. In their simplest occurrence, forms provide a means for a user to enter specific information into specific fields. More complex forms – in conjunction with an application that supports them – provide type checking (e.g., for dates or numbers), check that required fields are filled in, and offer data-dependent follow-up input fields (e.g., which input fields are required depends on how other fields have been filled in).

Many forms of interaction can be simplified as a precondition on a successor relationship. For example, a successor relation with a specific search term as a precondition leads to the page with the search results for this term, or the data in a form serves as a precondition that leads to a certain successor page, with different data leading to a different page. This leads to a definition of *interactive hypertext* that encompasses many, but not all, forms of interactive documents.

Definition 4.1.10 (Interactive Hypertext). *An interactive hypertext $H_i = (P, p_0, C, l, i)$ is an extension of a hypertext $H = (P, p_0, l)$, where C is a set of preconditions and $i \subseteq l \times C$ is a relation that assigns to pairs of pages $(p_i, p_j) \in l$ a precondition $c \in C$.*

Preconditions can be formalised as formulae, such as `step > 3`, `term = 'binary tree'`, or `fieldSSN ≠ ''`. Preconditions can act as filters that only allow access to some pages when certain conditions, such as user authentication, are met.

Example 4.1.11 (Interactive Hypertext). *Let $H = (P, p_0, l)$ be the hypertext from example 4.1.2. Let $C = \{(\text{userlevel} = \text{'advanced'})\}$ and $i = \{(\text{"Chapter 2"}, \text{"Chapter 4"}, (\text{userlevel} = \text{'advanced'}))\}$. Then $H_i = (P, p_0, C, l, i)$ is an interactive extension of H that only allows advanced users to navigate to chapter 4.*

This extension of regular hypertext can in similar form be transferred to other document models that will be discussed below.

Many interactive documents are not text-based, as will be discussed in the next section.

4.1.4 Other Types of Documents

An important type of non-textual documents are videos (cf. section 2.1.2 for an overview of video formats). A video in its simplest form is a sequence of images, with an optional accompanying audio sequence. More complex digital video documents abandon this linear structure in favour of a multitude of such sequences that are interconnected by hyperlinks, forming either a tree or a graph structure [CMLE08]. Such non-linear videos are often called *hypervideos*. The definition of hypertext (definition 4.1.1) can be adapted in a natural way to hypervideos, where instead of atomic pages there are atomic video sequences [AP05]. The definition of reading paths (definition 4.1.5) can be adapted in a similar way.

The term hypervideo leads to the more general notion of *hypermedia* that abstracts from the form of the document content and that encompasses both hypertext and hypervideo [MS04]. In addition, we extend the notion of a reading path on hypertexts to *navigation paths* on hypermedia documents.

Definition 4.1.12 (Hypermedia). *A hypermedia document $H = (O, o_0, l)$ consists of*

O	$\in O$	$l \subseteq O \times O$	<i>a non-empty set of atomic objects,</i>
			<i>a starting object, and</i>
			<i>a binary relation that models links, i.e., references from one object to another.</i>

Atomic objects can be text, image, audio or video objects.

Every hypermedia document must satisfy the condition that

$$\forall o_x \in O \setminus \{o_0\} : \exists n = (o_0, \dots, o_x)$$

where n is a navigation path.

Definition 4.1.13 (Navigation Path). *A navigation path n on a hypermedia document $H = (O, o_0, l)$ is a non-empty, possibly infinite sequence of objects (o_0, o_1, o_2, \dots) , for which holds that $(o_i, o_{i+1}) \in l$, for $i \in \mathbb{N}$.*

Digital videos, particularly hypervideos, conform to our notion of a document as established in description 4.1.9. The same is true for hypermedia documents, as long as they can be visualised for humans. A single video usually covers only a small range of topics. The same is true for most other types of hypermedia documents known today, namely for hypermedia documents containing text, images, audio and/or video. If hypermedia consists of multiple objects, then there is a distinct starting objects o_0 and the different sequences are ordered – either linearly or as defined by the link structure.

When regarding interactive documents, especially modern hypermedia documents that contain Javascript (cf. section 2.1.2), the distinction between documents and full computer programs quickly becomes blurred. This begs the question if and under what circumstances programs might be considered to be documents as well. A program consists of a sequence of commands that were written to execute a specific function [Knu73]. A program is usually not executed directly, but first transformed (compiled) into a form that is closer to the physical or logical architecture of the machine where the program should run. Most programs provide some form of visualisation (for example, a graphical user interface or a command line interface) that also facilitates user interaction.

Comparing the properties of computer programs with description 4.1.9, we can conclude that programs conform to our notion of a document, provided that they are not primarily running in the background, but offer a visualisation for all of their important aspects. This visualisation can consist of text, images, audio and even video. Since programs are created for specific tasks,

they cover only a small number of aspects (“topics”). Note that this does not preclude programs from providing a large number of functions, but rather that these functions are usually related, like different functions in an image manipulation program or in a word processor.

A program can be seen as a set of states that each describe the current status of the program. Transitions between states are triggered in a specific order that is either set by the author or that is determined through interaction with a user or by some other external input. A program can run multiple execution paths simultaneously (concurrency), but if it can provide a visualisation of the primary tasks it can still be regarded as a document. Every program has a distinct starting point from which all valid program states can be reached.

While interactive documents, videos and programs are not in the focus of this work, the principles and techniques developed here can be transferred and adapted to them.

4.1.5 Layers of Abstraction on Documents

We will now define different ways of regarding a document that reflect different levels of abstraction. This allows us to talk about different aspects of a document more clearly by grouping similar aspects, and by separating dissimilar aspects.

The files and formats of a document serve to define the visual representation of that document. However, they need to be interpreted by some appropriate program, which produces (renders) the visualisation. This visualisation is thus *derived* from the technical specification, but it is not directly *contained* within.

From this follows directly that data for which no visualisation is defined, for example entries in a database or a binary file with no associated rendering program, cannot be regarded as a document according to our understanding. It is literally not (human) readable, until visualisation instructions – even very simple ones – have been defined.

Note that if a document does not contain the definitions for all major parts of its visualisation, then external rendering instructions must exist in some form, for example as a set of guidelines, as a reference implementation, or as an external stylesheet for HTML or XML documents. The *possibility* of defining such instructions is not sufficient. This is an important distinction between what we regard as documents, and what we do not. Since the possibility to “read”, i.e., to view and to interpret, a document has always been a major aspect of its nature, we believe this distinction to be both meaningful and justified.

Yet the visual representation of the technical document is not the end of the interpretation process. Based on different visual or content-related clues, for example the formatting of certain paragraphs, the reader creates a further abstraction of the document in her mind that focusses on the structure: the text is divided into coherent units, with a superstructure of sections and chapters. Sometimes, this interpretation is trivial, when a headline explicitly states that it marks the beginning of a new chapter, as in “Chapter 1: Introduction”. But it still is a new level of abstraction, because it reduces visual and textual clues to an interpretation that provides new semantics, namely an aggregation of textual units (into, e.g., chapters or sections) and a hierarchy of such aggregations. Ideally, the reader’s interpretation mirrors that of the author, but ultimately – as in any form of communication – the burden of interpretation lies with the recipient.

Under certain circumstances, this interpretation can also be made by a computer program (see section 5.1). While the computer program cannot completely dispense with the visual interpretation, it takes its clues directly from the technical instructions that the visualisation is based on. It does not need to analyse the rendered document representation, but it still needs to take the rendering instructions into account.

We therefore define three different layers of abstraction for documents:

- ▶ The *technical layer* regards the technical representation of a document. It offers a technical perspective on the document.
- ▶ The *visual layer* regards how a document is visually presented to and seen by a reader. It offers a visual perspective on the document, which is an abstraction of the technical perspective.
- ▶ The *structural layer* regards the *logical* structure of a document. It offers a structural perspective on the document, which is an abstraction of both the technical and the visual perspective.

Example 4.1.14 (Document). *Let an example document consist of five chapters, starting with an introduction and ending with a conclusion. Chapter 1 links to chapter 2, which in turn links to both chapters 3 and 4. Chapters 3 and 4 both link to chapter 5.*

Chapter 2 contains a definition of data structures, chapter 3 contains an example for data structures, and chapter 4 contains both the definition and an illustration for binary trees.

*Each chapter is displayed on a different page or screen, chapter headlines are formatted in **bold and italics**, paragraph headlines such as for definitions are formatted in italics, and links are formatted as underlined.*

The document is spread over three different files, the first one holding chapters 1 and 5, the second one holding chapters 2 and 3, and the third one holding chapter 4.

Figure 4.4 shows how the document from example 4.1.14 can be seen on each of the three layers. On the technical layer (left), the three source files are shown. The actual file format could be HTML, L^AT_EX, an office document format such as Microsoft Word, or an XML-based format. Since all of these formats are both verbose and complex, the files are presented here in an abstract file format for illustration purposes. It consists of structure indicators, rendering instructions, text fragments written in quotes, and include instructions.

Rendering instructions indicate how some element should be interpreted on the visual layer, i.e., they define a visualisation. For example, the first rendering instruction in figure 4.4 (top left) indicates that the following text fragment should be displayed in both **bold and italics**. In document file formats, rendering instructions can be commands like `\textbf{}` and `\small{}` (L^AT_EX), or `` and `<small>` (HTML). Some rendering instructions only give an abstract description of the intended visual effect, like the L^AT_EX emphasis (`\emph{}`) command that produces cursive output if the surrounding text is straight, and straight output if the surrounding text is cursive. Thus the precise interpretation of the rendering instructions is sometimes up to the rendering application.

Structure indicators give an indication of how some element should be interpreted on the structural layer. For example, the first structure indicator in figure 4.4 (top left) indicates that the following text fragment should be interpreted as a chapter, namely as chapter C1 as shown on the structural layer (right). Structure indicators in actual file formats are, for instance, `\section{}` commands in L^AT_EX, or headline (`<h1>`, `<h2>`, ...) commands in HTML. Usually, they are a combination of rendering instructions and additional structural semantics. For example, the L^AT_EX `\section{}` command defines how a section title should be visualised, and it defines a structural semantics that causes the section title to be included in the table of contents. References such as `\ref{}` in L^AT_EX or `` are also a combination of structure indicators and rendering instructions. In the visual representation of a document, these references are usually interpreted as either textual references or hyperlinks.

Text fragments are the verbatim content of a document. Their precise visualisation is influenced by the surrounding rendering instructions.

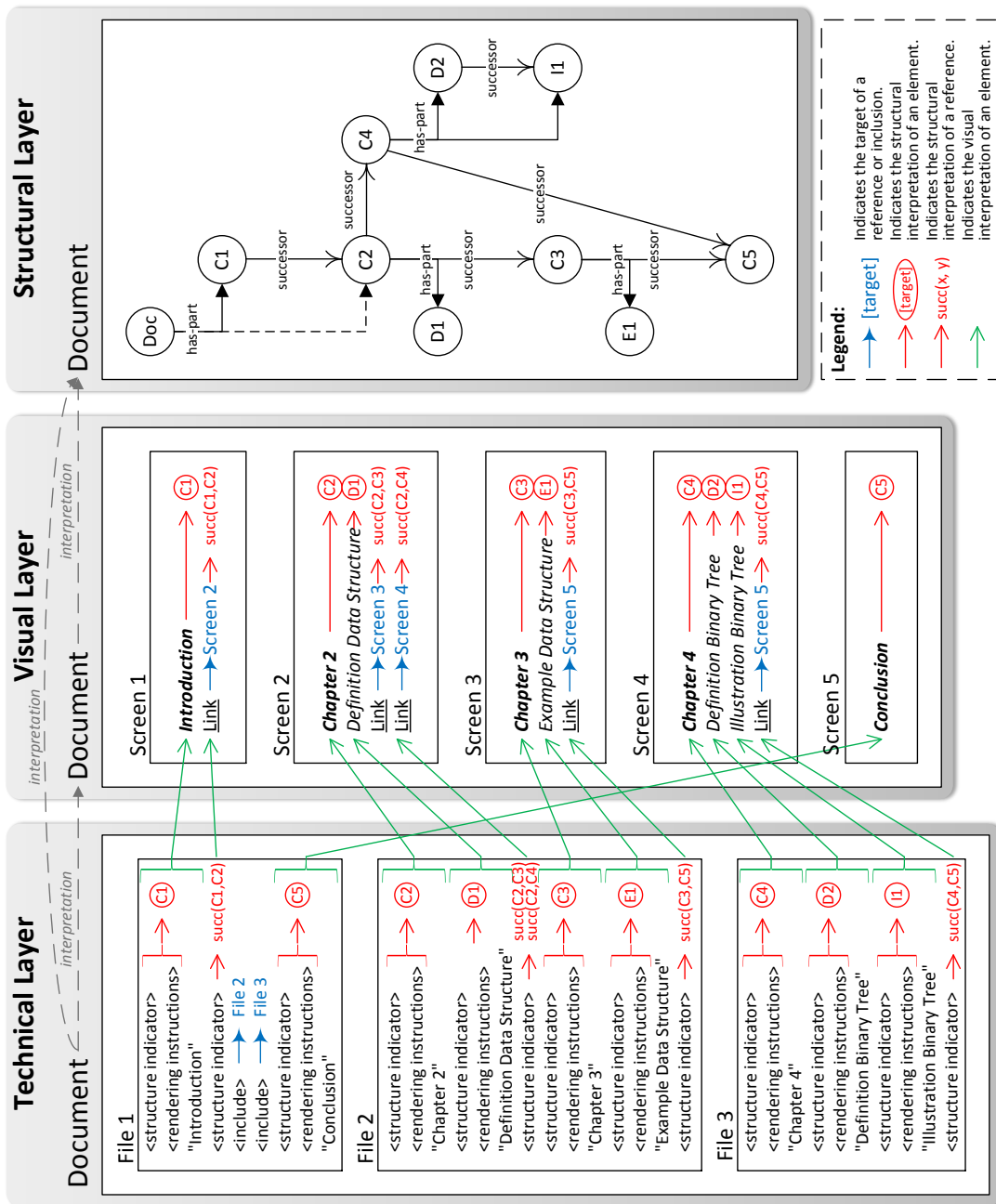


Figure 4.4: Document abstraction layers

Include instructions indicate that another file should be processed, and all commands within should be treated as if they occurred at the current file location. Include instructions have no bearing on the visual or structural interpretation of a document. In L^AT_EX, the `\input{}` command is an include instruction. In HTML, includes can be accomplished by either a scripting language such as *Server Side Includes* (SSI)³, or by using *iframes*. In most file formats, circular includes are forbidden, while including the same file multiple times is allowed. In the illustration, a blue arrow indicates the target of the include.

Definition 4.1.15 (Technical Layer). *The technical layer is an abstract perspective on a document that regards its technical composition:*

1. *the files it is contained in,*
2. *the formats it is written in, and*
3. *the commands used to specify its content and layout.*

As shown in figure 4.4, the file contents (technical layer) are interpreted by some appropriate mechanism as a visual representation (visual layer). Chapter and paragraph headings are rendered as described by the structure indicators and the rendering instructions. The green arrows point from the elements in the source file to the visual interpretation on the visual layer (centre). Other structure indicators represent references that are interpreted as hyperlinks and are rendered accordingly. Include instructions are resolved and the included files rendered as well. The targets of the include instructions are indicated with blue arrows in the figure.

Definition 4.1.16 (Visual Layer). *The visual layer is an abstract perspective on a document that regards its visual representation:*

- | | |
|---------------|--|
| what | <i>the text, images, or other media elements that are shown,</i> |
| how | <i>their formatting,</i> |
| where | <i>their layout and placement, and</i> |
| in what order | <i>the representation of the possible reading paths.</i> |

The structural layer can be seen on the right hand side. The five chapters are shown as part of the document (for reasons of clarity, only two of these part-of arrows are displayed), and the paragraphs (definitions, examples, and illustrations) are shown as part of the chapters. The order between the content elements is indicated as a successor relationship, drawn as open-headed arrows. On the technical and on the visual layer, an indication of their structural representation is shown in red for elements that are interpreted as structural elements. For example, the first two instructions in File 1 are interpreted as the structural element C1, as is the “Introduction” headline on the visual layer.

Details on how these interpretations can be defined will be discussed in section 5.1.

Definition 4.1.17 (Structural Layer). *The structural layer is an abstract perspective on a document that regards its logical structure, consisting of*

- | | |
|---------------------------------|--|
| <i>atomic content objects</i> | <i>e.g., short passages of text or images that have no internal structure,</i> |
| <i>low-level content units</i> | <i>i.e., aggregations of atomic content objects,</i> |
| <i>high-level content units</i> | <i>i.e., aggregations of atomic content objects and content units, and</i> |
| <i>successor relations</i> | <i>between different atomic content objects and content units.</i> |

³http://httpd.apache.org/docs/2.2/mod/mod_include.html

In section 4.2 we will define restrictions on the possible structures that will be regarded in this thesis.

We can now combine definitions 4.1.15, 4.1.16 and 4.1.17 to define a *three-layered document perspective*.

Definition 4.1.18 (Three-Layered Document Perspective). *The three-layered document perspective is a perspective on a document that defines three abstract layers on a document: the technical layer, the visual layer, and the structural layer.*

This perspective will serve as a point of reference when modelling different aspects of digital documents.

4.1.6 Formalised Notion of a Document

The perception gained by the discussion above allows us to further refine our notion of a document. As can be seen on the technical layer, a document contains unstructured text and other media objects interspersed with commands that have an impact on the interpretation (even if only the visual interpretation) of the text. Such a type of document is usually called *semi-structured*.

Definition 4.1.19 (Semi-Structured Data). *Semi-structured data is data that contains both unstructured elements and elements that serve to separate, group, or hierarchically organize the unstructured elements. The structure of semi-structured data is not defined by a fixed schema, but rather by a flexible schema in the form of a grammar (cf. [Fre09]).*

We will call the unstructured elements of a document *media objects*.

Definition 4.1.20 (Media Object). *A media object is a continuous range of text, an image, some other media element, or a combination of multiple elements that can be part of a digital document. A media object is considered an atomic entity within a document.*

Based on definitions 4.1.15 and 4.1.16, we will define a *base document model* that incorporates aspects of the technical specification of a document and of its rendering instructions. It does not, however, include any interpretation of these instructions; in particular, it does not include a visualisation. It abstracts from the document's source files – both from the file formats and from possible inclusion commands.

Definition 4.1.21 (Base Document Model). *Let*

$M = \{m_0, \dots, m_n\}$ *be a set of media objects, and* $T \subseteq M$ *the subset of* M *that contains all text fragments of* M ;

F *be a set of formatting options such as bold or italic;*

c *be a function that assigns to each* $t \in T$ *its text content;*

$s \subseteq M \times M$ *be a relation that assigns successors to media objects; and*

$f \subseteq T \times 2^F$ *be a relation that assigns formatting options to text fragments.*

Then $B = (M, F, c, s, f)$ *is a base document model.*

A base document model does not have to be complete, i.e., it does not have to incorporate all elements of the source document. Therefore, there can be various different base document models for any document.

Remark 4.1.22. *Note that, whenever a media object has more than one successor, we will assume that the base document model holds these successor relationships in the order in which there were defined in the original document.*

Remark 4.1.23. Also note that, while a technical representation of a document may contain commands to include other files or parts of files, a base document model is an abstract view of the entire technical representation, with all includes resolved.

The definition of a base document model does not require any specific directionality in the document's text, for example left-to-right or top-to-bottom, or even the restriction to a single directionality. Without loss of generality, we will assume a left-to-right directionality throughout this thesis unless indicated otherwise.

Example 4.1.24 (Base Document Model). Let the following XML code, split into three separate parts, implement the document sketched in example 4.1.14. The specification language is inspired by HTML and will not be discussed in detail.

```

1 <!-- index.xml -->
2 <document>
3   <chapter id="chapter1">
4     <div style="bold, italic">Introduction</div>
5     <a href="#chapter2" style="underlined"/>
6   </chapter>
7   <include href="c2c3.xml"/>
8   <include href="c4.xml"/>
9   <chapter id="chapter5">
10    <div style="bold, italic">Conclusion</div>
11  </chapter>
12 </document>

1 <!-- c2c3.xml -->
2 <document>
3   <chapter id="chapter2">
4     <div style="bold, italic">Chapter 2</div>
5     <div style="italic">Definition Data Structure</div>
6     <a href="#chapter3" style="underlined"/>
7     <a href="#chapter4" style="underlined"/>
8   </chapter>
9   <chapter id="chapter3">
10    <div style="bold, italic">Chapter 3</div>
11    <div style="italic">Example Data Structure</div>
12    <a href="#chapter5" style="underlined"/>
13  </chapter>
14 </document>

1 <!-- c4.xml -->
2 <document>
3   <chapter id="chapter4">
4     <div style="bold, italic">Chapter 4</div>
5     <div style="italic">Definition Binary Tree</div>
6     <div style="italic">Illustration Binary Tree</div>
7     <a href="#chapter5" style="underlined"/>
8   </chapter>
9 </document>

```

Then $B = (M, F, c, s, f)$ is a base document model for the XML implementation of the document, with

- $M = \{t_1, t_2, \dots, t_9, l_1, \dots, l_5\} = T,$
- $F = \{\text{bold, italic, underlined}\},$
- $c = \{(t_1, \text{"Introduction"}), (t_2, \text{"Chapter 2"}),$
 $(t_3, \text{"Definition Data Structure"}), \dots, (l_1, \text{"Link"}), \dots\}$
- $s = \{(t_1, l_1), (l_1, t_2), (t_2, t_3), (t_3, l_2), (l_2, t_7), (l_2, t_4), \dots\},$ and
- $f = \{(t_1, \{\text{bold, italic}\}), (l_1, \{\text{underlined}\}), \dots\}.$

This example is illustrated in figure 4.5.

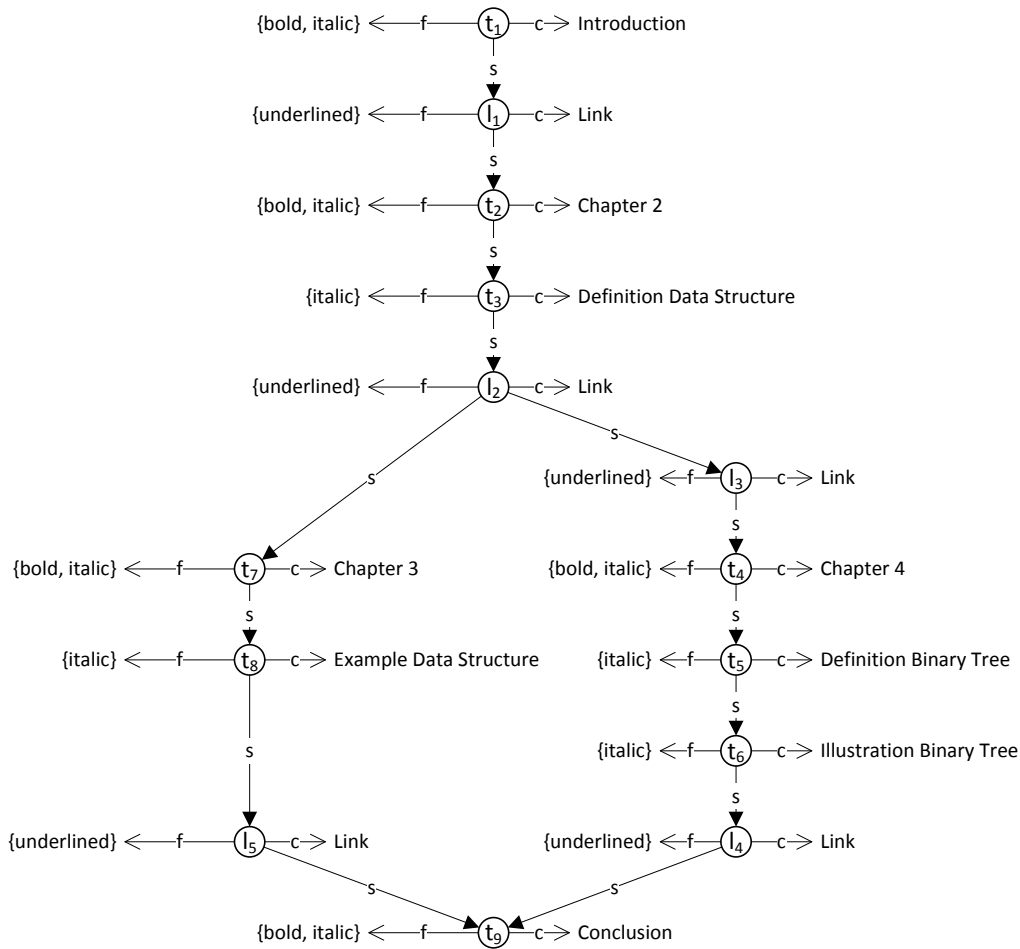


Figure 4.5: Illustration of the base document model from example 4.1.24

Remark 4.1.25. Note that example 4.1.24 contains text and formatting information for the references (the links in figure 4.4). A base document model may include any visual interpretation of these elements, such as underlined text for the references, whether the rendering is explicitly specified in the source document or only implicitly as a default rendering convention.

A base document model is an abstract view of a technical representation of a document. It can be used for any document representation that is based on a graph structure. Some simple document formats consist of a linear list of data representation commands, for example plain text files that consist of a sequence of strings, or text files with simple formatting commands to indicate bold or italic text.

Most document formats, however, are a bit more structured, often in the form of command-trees. XML-based formats are an obvious example, but other formats like L^AT_EX or Rich Text also use trees of opening and closing commands. Even binary formats such as EDIFACT⁴ use a tree-shaped hierarchy to structure their data representation commands. For these formats, a base document model may simply consist of a sequence of nodes representing the various command elements in document order, with their visualisation and formatting instructions and their textual content attached. The model may also contain nodes representing the closing tags for each command in the appropriate locations.

Cross-reference commands like hyperlinks expand the linear structure and lead to a more complex graph structure in the base document model.

Data that describes a document or that provides additional information about a document – *metadata* – can be associated with a base document model.

Definition 4.1.26 (Base Document Model with Associated Metadata). *Let $B = (M, F, c, s, f)$ be a base document model. Then $A_B = (L, V, d)$ is the metadata associated with B , where*

- L is a set of labels such as ‘author’ or ‘creation date’,*
- V is a set of literal metadata values such as “Johann Wolfgang von Goethe” or “1808”, and*
- $d \subseteq (M \cup \{B\}) \times L \times V$ is a ternary relation that assigns labels and literal metadata values to media objects or to the base document model itself.*

Example 4.1.27 (Base Document Model with Associated Metadata). *Let B be the base document model from example 4.1.24.*

$A_B = (L, V, d)$ is a set of metadata associated with B , with

- $L = \{\text{author, publisher}\}$,*
- $V = \{\text{“D. Knuth”, “Aaron M. Tenenbaum”, “University Press”}\}$, and*
- $d = \{(B, \text{author, “D. Knuth”}), (B, \text{publisher, “University Press”}), (t_6, \text{author, “Aaron M. Tenenbaum”})\}$.*

Real-world examples for sources of metadata in base document models are office documents such as Microsoft Word or PDF documents that contains author or keyword data.

In contrast to hypertext documents, the definition of a base document model does not require a document to have a single starting point, or even to be connected. For example,

$$B_{abc} = (\{a, b, c\}, \emptyset, \{(a, \text{“A”}), (b, \text{“B”}), (c, \text{“C”})\}, \emptyset, \emptyset)$$

is a valid base document model with three unconnected text fragments “A”, “B” and “C”. It is, however, hard to imagine a sensible visualisation for B_{abc} , and there is nothing to base an interpretation for a document structure on, so the structural layer remains empty. For the purposes of this thesis, documents with more structural coherence are required. We therefore define the notion of a *connected base document model*. To this end, we first transfer the notion of reading paths (cf. definition 4.1.5) from hypertexts to base document models.

⁴While not a “document” in the sense of this theses because it lacks visualisation instructions, EDIFACT is a standard format for electronic data in commercial environments.

Definition 4.1.28 (Reading Path on a Base Document Model). *A reading path r on a base document model $B = (M, F, c, s, f)$ is a non-empty, possibly infinite sequence of media objects (m_0, m_1, m_2, \dots) , for which holds that $(m_i, m_{i+1}) \in s$, for $i \in \mathbb{N}$.*

Definition 4.1.29 (Connected Base Document Model). *A base document model $B = (M, F, c, s, f)$ is connected iff*

$$\exists m_0 \in M : \forall m \in M \setminus \{m_0\} : \exists r = (m_0, \dots, m) \text{ (existence of a starting object)}$$

where r is a reading path. If B is connected, then $B' = (M, F, m_0, c, s, f)$, where $m_0 \in M$ is a starting object, is the connected base document model for B .

Example 4.1.30 (Connected Base Document Model). *Let $B = (M, F, c, s, f)$ be the base document model from examples 4.1.24 and 4.1.27.*

B is connected and $B' = (M, F, t_1, c, s, f)$ is the connected base document model for B .

Remark 4.1.31. *Note that a base document model that is connected can have multiple potential starting objects. For example, for the base document model*

$$B_{ab} = (\{a, b\}, \emptyset, \{(a, \text{“A”}), (b, \text{“B”})\}, \{(a, b), (b, a)\}, \emptyset)$$

both $B'_{ab} = (\dots, a, \dots)$ and $B''_{ab} = (\dots, b, \dots)$ are valid connected base document models.

Not surprisingly, hypertexts can be seen as connected base document models. Vice versa, for every connected base document model there exists exactly one hypertext.

Definition 4.1.32 (Hypertexts as Connected Base Document Models). *Let $H = (P, p_0, l)$ be a hypertext. Then $B'_H = (P, F, p_0, c_P, l, f)$ is a connected base document model for H , where c_P is a function assigning to each page $p \in P$ its textual content, F is a set of formatting options, and f is a relation assigning to pages $p \in P$ a subset of F .*

Definition 4.1.33 (Connected Base Document Models as Hypertexts). *Let $B' = (M, F, m_0, c, s, f)$ be a connected base document model. Then $H_{B'} = (M, m_0, s)$ is the hypertext for B' .*

Thus, connected base document models are an *extension* of hypertexts that allow for a finer sub-structuring of atomic hypertext pages and that explicitly include visual information like text formatting. A page-centric document model is insufficient for the purposes of this thesis, as we will explore in more detail in section 4.2.1.

Example 4.1.34 (Hypertexts as Connected Base Document Models). *Let*

$$H = (P, p_0, l)$$

with $P = \{p_0, p_1, p_2\}$ and $l = \{(p_0, p_1), (p_0, p_2), (p_1, p_2)\}$ be a hypertext. Then

$$B'_H = (P, \emptyset, p_0, c_P, l, \{(p_0, \emptyset), (p_1, \emptyset), (p_2, \emptyset)\})$$

and

$$B''_H = (P, \{\text{plain}\}, p_0, c_P, l, \{(p_0, \{\text{plain}\}), (p_1, \{\text{plain}\}), (p_2, \{\text{plain}\})\})$$

are both connected base document models for H , where c_P is a function assigning to each page $p \in P$ its textual content.

Example 4.1.35 (Connected Base Document Models as Hypertexts). *Let $B' = (M, F, t_1, c, s, f)$ be the connected base document model from example 4.1.30. Then $H_{B'} = (M, t_1, s)$ is the hypertext for B' , where every media object from B' is represented as a single page.*

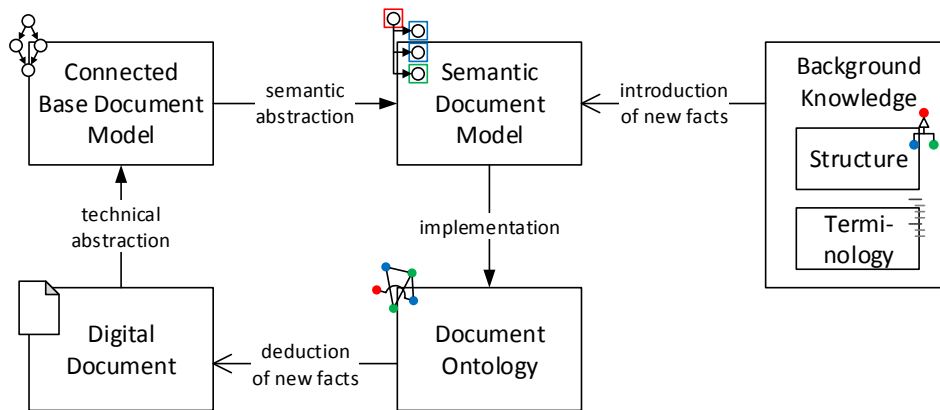


Figure 4.6: Document models

4.2 Modelling Semantic Data

Figure 4.6 shows how the document models that we will discuss in this section fit together. The connected base document model is a technical abstraction of the original document that ignores implementation specific details of the document format. This abstraction is usually very straightforward. The semantic document model is a semantic abstraction of the base document that explicitly models the structure and semantic relations. New domain-dependent background knowledge about a document’s structure and about the terminology can be added to the semantic model. It must be added to this model, because the base document model lacks the necessary structure and semantic “anchors”, i.e., entities that can be source or target of semantic relations. The semantic model can be implemented in description logics or in a compatible language, and its semantics are compatible with description logics. From the document structure, the semantic relations, the background knowledge, and their combination new facts about the document can be deduced. This deduction is generally not possible on the original document or on the base document model alone. Details on the technical and the semantic abstraction can be found in section 5.1, methods of obtaining background knowledge are described in chapter 6, and the semantic processing is discussed in chapter 9.

4.2.1 Modelling Documents

Having defined our notion of a document and having defined a low-level document model, the following section will describe a more abstract model for documents conforming to this understanding. This model will extend the formalisations of section 4.1 to include structural and semantic data that is not commonly contained explicitly in a document.

The structure of a document, i.e., the hierarchy of chapters, sub-chapters, paragraphs and so forth as evident on the structural layer (cf. definition 4.1.17), generally exists only implicitly in documents. Some document formats like \LaTeX allow for indications of some structuring, for example via the `\section{}` command. However, use of these structural indicators is not enforced; an author can easily disregard them in favour of simple formatting instructions like `\large{}` or `\textbf{}`, or even redefine them to mean something entirely different. Moreover, not all document formats provide such commands; many formats solely offer formatting commands. Other formats, such as office formats, offer templates that assign names to specific

formatting options. Through a consistent naming scheme, some semantics for these templates can be implied; however, as with \LaTeX , the semantic cannot be enforced.

Yet to be able to establish reliable semantic connections between different parts of a document, the structural relationship of these parts is sometimes pivotal, for example when aggregating the semantics of sub-structures at a higher structural level. Therefore knowledge about the structure needs to be represented in the document model. To this end, we will define a structural document model, before defining a semantic document model. How this knowledge about the structure can be obtained will be discussed in section 5.1.

For a structural view on a document as shown in figure 4.4, two primary types of relation on document fragments are required. The first type of relation are “horizontal” successor or reference relations that define the possible reading paths through the document. Such relationships are usually limited to document fragments on the same hierarchical level within the document structure. The second type are “vertical” has-part relations that define the hierarchical structure between the document fragments. A structural document model needs to have at least one successor relation and one has-part relation.

A document model focussed on the structure of a document needs to model the different document fragments on different hierarchical levels, for example a fragment that represents a section, and multiple fragments that represent paragraphs within that section. These fragments are a logical subdivision of a document, with the bottom fragments (leaves) in the has-part hierarchy representing media objects. While the actual media objects do not need to be part of the model, in order to keep information relevant for a semantic model, relevant terms or topics covered by the media objects must be represented in the structural model.

With this understanding, we define a structural document model.

Definition 4.2.1 (Structural Document Model). *Let*

- F = $\{f_0, \dots, f_n\}$ be a set of document fragments;
- $f_0 \in F$ be a starting fragment from F with respect to both p and s ;
- s be a relation $s \subseteq F \times F$ between document fragments that defines successors for fragments;
- p be a relation $p \subseteq F \times F$ between document fragments that defines a hierarchy on fragments;
- T be a set of terms; and
- c be a relation $c \subseteq F \times T$ that specifies which document fragments cover or deal with which terms.

The relation p must satisfy the following constraints, with $F_{>} = F \setminus \{f_0\}$:

$$\forall f_{n+1} \in F_{>} : \exists F_n = \{f_1, \dots, f_n\} \subseteq F_{>} : \{(f_0, f_1), \dots, (f_n, f_{n+1})\} \subseteq p \quad (4.1)$$

$$\forall f_{n+1} \in F_{>} : (\exists F_n = \{f_1, \dots, f_n\}, F_m = \{f'_1, \dots, f'_m\} \subseteq F_{>} : \{(f_0, f_1), \dots, (f_n, f_{n+1})\} \cup \{(f_0, f'_1), \dots, (f'_m, f_{n+1})\} \subseteq p) \Rightarrow F_n = F_m \quad (4.2)$$

Then $S = (F, f_0, s, p, T, c)$ is a structural document model.

Remark 4.2.2. Constraints 4.1 and 4.2 from definition 4.2.1 state that from the starting object f_0 , every fragment $f \in F$ must be reachable through p in exactly one unique sequence of applications of p , either directly or indirectly.

Proposition 4.2.3 (Constraints on p). *Constraints 4.1 and 4.2 from definition 4.2.1 are equivalent to stating that the transitive closure p^+ of p must be both right-total and left-unique:*

$$\forall f_j \in F_{>} : \exists f_i \in F : (f_i, f_j) \in p^+ \quad (4.3)$$

$$\forall f_k \in F_{>} : (\exists f_i, f_j \in F : (f_i, f_k) \in p^+ \wedge (f_j, f_k) \in p^+) \Rightarrow f_i = f_j \quad (4.4)$$

where $F_{>} = F \setminus \{f_0\}$.

Proof of Proposition 4.2.3.

1. constraints 4.1 and 4.2 \Rightarrow constraint 4.3 (proof by contradiction):

Assume $\exists f_j \in F_{>} : \nexists f_i \in F : (f_i, f_j) \in p^+$.

Then $\exists F_{j-1} = \{f_1, \dots, f_{j-1}\} \subseteq F_{>} : \{(f_0, f_1), \dots, (f_{j-1}, f_j)\} \subseteq p$
(constraint 4.1).

Let $f_i = f_{j-1}$. ⚡

2. constraints 4.1 and 4.2 \Rightarrow constraint 4.4 (proof by contradiction):

Assume $\exists f_k \in F_{>}, f_i, f_j \in F : (f_i, f_k) \in p^+ \wedge (f_j, f_k) \in p^+ \wedge f_i \neq f_j$.

Then $\exists F_i = \{f_1, \dots, f_i\} \subseteq F_{>} : \{(f_0, f_1), \dots, (f_i, f_k)\} \subseteq p$
(constraint 4.1)

and $\forall F_j = \{f'_1, \dots, f_j\} \subseteq F_{>} : \{(f_0, f'_1), \dots, (f_j, f_k)\} \subseteq p \Rightarrow F_j = F_i$
(constraint 4.2).

Let $F_j = F_i$. Then $F_j = F_i \Rightarrow f_j = f_i$. ⚡

3. constraints 4.1 and 4.2 \Leftarrow constraints 4.3 and 4.4 (proof by induction over n):

Hypothesis ($n = 0$): $\forall f_1 \in F_{>} : \exists F_0 = \emptyset : \{(f_0, f_1)\} \in p$, F_0 is unique.

Step ($n \rightarrow n + 1$):

Assume $\forall f_{n+1} \in F_{>} : \exists F_n = \{f_1, \dots, f_n\} \subseteq F_{>} :$

$\{(f_0, f_1), \dots, (f_n, f_{n+1})\} \in p$

and $\forall F_m = \{f'_1, \dots, f'_m\} \subseteq F_{>} :$

$\{(f_0, f'_1), \dots, (f'_m, f_{n+1})\} \in p \Rightarrow F_n = F_m$

Show $\forall f_{n+2} \in F_{>} : \exists F_{n+1} = \{f_1, \dots, f_{n+1}\} \subseteq F_{>} :$

$\{(f_0, f_1), \dots, (f_{n+1}, f_{n+2})\} \in p$

Let $F_{n+1} = F_n \cup \{f_{n+1}\}$, with $(f_{n+1}, f_{n+2}) \in p$.

Then F_n exists (assumption), f_{n+1} exists (constraint 4.3), and f_{n+1} is unique (constraint 4.4).

□

Ideally, the terms T used in a structural document model are grammatically normalised, i.e., nouns are all in nominative singular, adjectives are all in nominative singular for a fixed gender (if the language differentiates adjective forms for different genders), and verbs are all in infinitive form. As a technical alternative, we can abstract from the grammatical form of the terms by reducing them to their word stems, using well-established stemming algorithms [FBY92]. For the remainder of this thesis, we will assume that all terms are in some way normalised, either as a grammatical base form, or through some technical means. This allows us to abstract from different forms of a single word.

The successor relation s defines a graph structure, which is the common structure of hyper-media documents. However, the constraints on the has-part relation p have to be more strict: every fragment of a document can be regarded in its own local context, which among other things depends on its precise position on the structural hierarchy, i.e., on its partners regarding the has-part relations. This leads to a tree-structure for the has-part relation, which is also the commonly established form of document hierarchies. It is, for example, generally expected that a subsection has a unique parent section. An illustration of possible s - and p -relationships in different models can be seen in figure 4.7.

This restriction also implies that the has-part relation must not be transitive: regard a simple hierarchy of document fragments A , B and C , with $has-part(A, B)$ and $has-part(B, C)$. For a transitive $has-part$, this hierarchy would entail the additional relationship $has-part(A, C)$, which violates constraint 4.2.

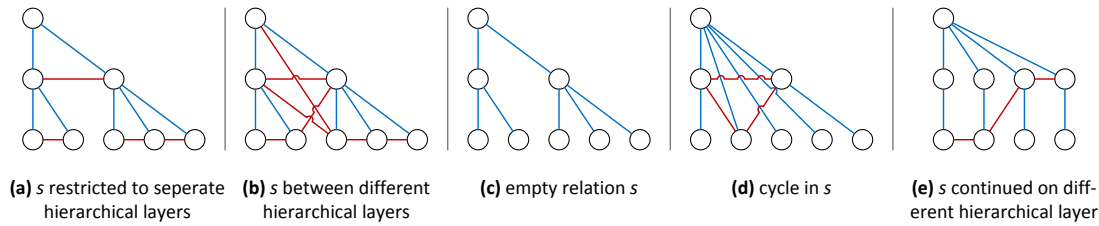


Figure 4.7: Illustration of the relations s (red) and p (blue) in various structural document models (directionality omitted for clarity)

Example 4.2.4 (Structural Document Model). *Let*

$$\begin{aligned}
 F &= \{f_0, f_1, f_2, f_3, f_4, f_5, f_{21}, f_{31}, f_{41}, f_{42}\}; \\
 s &= \{(f_1, f_2), (f_2, f_3), (f_2, f_4), (f_3, f_5), (f_4, f_5), (f_{41}, f_{42})\}; \\
 p &= \{(f_0, f_i) \mid \text{for } 1 \leq i \leq 5\} \cup \{(f_2, f_{21}), (f_3, f_{31}), (f_4, f_{41}), (f_4, f_{42})\}; \\
 T &= \{\text{Data Structure, Binary Tree}\}; \text{ and} \\
 c &= \{(f_{21}, \text{Data Structure}), (f_{31}, \text{Data Structure})\} \cup \\
 &\quad \{(f_{41}, \text{Binary Tree}), (f_{42}, \text{Binary Tree})\}.
 \end{aligned}$$

Then $S = (F, f_0, s, p, T, c)$ is a structural document model. S is based on the document illustrated in figure 4.4.

Note that the successor relation in a base document model represents a technical relationship, such as a natural successor by file order, a link, or an include command. The successor relation in a structural document model represents a logical relationship, such as a natural successor by document order, a link, or a cross reference.

We will now introduce a convenient notation for structural document models.

Definition 4.2.5 (Notation for Structural Document Model). *The basic syntax for structural document models is shown as an EBNF grammar:*

```

model      = fragment | "[" "]" ;
fragment   = id | id, "[", { sub-fragments }, { "|", sub-fragments }, "]" ;
sub-fragments = fragment, { ",", sub-fragments } ;

```

id represents a unique identifier for a fragment, such as 1 or ID0001. A structural document model is thus written as the identifier of its root fragment, followed by an ordered list of sub-fragments of the root in square brackets. Each sub-fragment is recursively followed by a list of

its sub-fragments, and so forth. \square represents an empty document model. If a fragment does not have sub-fragments, the empty list \square can be omitted. If the order of sub-fragments is not unique, then alternate orderings can be written after a $|$ symbol.

Example 4.2.6 (Notation for Structural Document Model). *Let $1, \dots, 4$ be fragment identifiers. Then*

- ▶ \square ,
- ▶ 1 ,
- ▶ $1\square$,
- ▶ $1[2]$,
- ▶ $1[2, 3]$, and
- ▶ $1[2[3, 4|4, 3]]$

are notations of structural document models.

The structural document model from example 4.2.4 can be written as $0[1, 2[21], 3[31], 5 | 1, 2[21], 4[41, 42], 5]$.

In addition to the notation of the basic structure of a document model, we define five operations on structural document models: $+$, $*$, $!$, $-$, and \pm .

Definition 4.2.7 (Operations on Structural Document Models). *Let S be the set of structural document models, with $s, s_1, s_2 \in S$, and let F be the set of document fragments, with $f, f_1, f_2 \in F$.*

$+$: $S \times S \rightarrow S$ is a **graft operator** that combines two structural document models. If the root fragment of the second model is contained (without its sub-fragments) within the first, then the root of the second model replaces its counterpart in the first model, and all its sub-fragments are transferred to the first model. If the root fragment already has sub-fragments in the first model, the sub-fragments from the first model are appended after them. If the root fragment of the second model is not contained within the first, then the operation simply returns the first model.

As a convenience, we will also allow $+$: $S \times F$ to be used as a shorthand: $+(s, f) := +(s, f\square)$.

$*$: $S \times F \times S \rightarrow S$ is an **auxiliary operator** that serves as another shorthand: $*(s_1, f, s_2) := +(s_1, f[s_2])$. It can be used in situations where $+$ cannot be applied because the root fragment of the second model s_2 is not contained within the first model s_1 . A fragment $f \in s_1$ is then selected and $*(s_1, f, s_2)$ applies the $+$ operator to the first model and to the second model extended by f , namely $f[s_2]$.

$!$: $S \times F \times F \rightarrow S$ is an **augment operator** that adds a reference relationship between two fragments if both fragments are part of the model. If the fragments are contained within the model, the model augmented with the new relationship is returned, otherwise the original model is returned.

$-$: $S \times F \rightarrow S$ is a **prune operator** that removes a fragment and all its sub-fragments and their relationships from a model if the fragment is part of the model. If the fragment is not contained within the model, the operation returns the original model. This operator is currently not used or required, but it becomes useful when changes to a document model are regarded, for example when tracking multiple versions of a document, where parts can not only be added but also removed.

\pm : $S \times S \times F \rightarrow S$ is a **move operator** that moves a fragment and all its sub-fragments from one point in a model to another. It is defined as $\pm(s_1, f_2[s_2], f_1) := +(-(s_1, f_2), f_1[f_2[s_2]])$, i.e., first the second model is removed from the first, and then added to it at a specified fragment.

For convenience, we will also allow the $+$ and $-$ operators to be used in infix notation.

Example 4.2.8 (Operations on Structural Document Models). *Let $1, \dots, 4$ be fragment identifiers. Then*

- ▶ $1 + 1[2] = 1[2]$,
- ▶ $1[2] + 1[3[4]] = 1[2, 3[4]]$,
- ▶ $1[2] + 1[3] = 1[2, 3]$,
- ▶ $1 + 1[2[3]] = 1[2[3]]$,
- ▶ $1[2, 4[3, 4]] + 4[5] = 1[2, 4[5][3, 4[5]]]$,
- ▶ $1 + 1 = 1$,
- ▶ $*(1, 1, 2) = 1 + 1[2] = 1[2]$,
- ▶ $*(1[2[3], 4], 2, 5) = 1[2[3], 4] + 2[5] = 1[2[3, 5], 4]$,
- ▶ $!(1[2, 3], 2, 3) = 1[2, 3]$ augmented with a reference relationship between 2 and 3,
- ▶ $1 - 1 = []$,
- ▶ $1[2] - 2 = 1$,
- ▶ $1[2] - 1 = []$,
- ▶ $1[2, 3] - 2 = 1[3]$,
- ▶ $1[2, 3] - 3 = 1[2]$,
- ▶ $1[2[3]] - 2 = 1$,
- ▶ $1[2, 4[3, 4]] - 4 = 1[2, 3]$, and
- ▶ $\pm(1[2[3], 4], 4[], 2) = (1[2[3], 4] - 4) + 2[4[]] = 1[2[3, 4]]$.

Proposition 4.2.9 (Properties of Operations on Structural Document Models). *The $+$ operator is associative but not commutative. Any structural document model s serves as its own neutral element w.r.t. $+$.*

Therefore, $(S, +)$ is a semigroup.

Proof Sketch of Proposition 4.2.9.

$+$ is not commutative because for example $1[2] + 2 = 1[2] \neq 2 = 2 + 1[2]$.

Let $s_1, s_2, s_3 \in S$ be three structural document models. To show associativity of $+$, namely that $(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$, there are three relevant cases to be regarded.

First, let the root fragment of s_2 not be contained in s_1 . Then $s_1 + s_2 = s_1$, regardless of what has been added to s_2 because the root fragment cannot change. Therefore $(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$ holds in this case.

Second, let the root fragment of s_3 not be contained in s_2 . Then $s_2 + s_3 = s_2$, regardless of whether or not s_2 has already been integrated into s_1 or not. Therefore $(s_1 + s_2) + s_3 = s_1 + (s_2 + s_3)$ holds in this case as well.

Finally, let the root fragment of s_2 be contained in s_1 and let the root fragment of s_3 be contained in s_2 . W.l.o.g., let $s_1 = 1[\dots, 2[\dots], \dots]$, $s_2 = 2[\dots, 3[\dots], \dots]$, and $s_3 = 3[\dots]$. Then $(s_1 + s_2) + s_3 = 1[\dots, 2[\dots, 3[\dots], \dots], \dots] = s_1 + (s_2 + s_3)$. \square

In order to extend structural document models to encompass semantics, we recall definition 3.2.23 (ontology), which we will use to represent knowledge.

As a bridge between the mathematical document models and ontologies based on description logics, we introduce the functions $name()$ and $assertions()$.

Definition 4.2.10 (Function $name()$). $name()$ returns a unique description logics atomic role name for a relation.

Example 4.2.11 (Function $name()$). For example, $name(s) = s$ for a relation s , or $name(hasPart) = \mathit{has-part}$ for a relation $hasPart$.

Definition 4.2.12 (Function $assertions()$). $assertions()$ returns a set of individual assertions that are compatible with a given relation: $assertions(s) := \{name(s)(d,r) \mid \forall d \in D : \forall r \in R : (d,r) \in s\}$ for a relation $s \subseteq D \times R$. The elements of D and R are implicitly used as description logic individuals.

Example 4.2.13 (Function $assertions()$). For example, $assertions(s) = \{s(x_1, x_2)\}$ for a relation $s = \{(x_1, x_2)\}$.

Definition 4.2.14 (Semantic Document Model). Let $S = (F, f_0, s, p, T, c)$ be a structural document model. Let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ and $\mathcal{O}_T = (C_T, R_T, T, X_T)$ be two ontologies. Then $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ is a semantic document model.

\mathcal{O}_S is a structural ontology that provides the terminology for modelling the structure of the document. It makes use of the document fragments F of S as individuals, and defines

- C_S a set of concepts that describe structural elements of the document, such as **Chapter** or **Definition**;
- R_S a set of roles that define relations between fragments, with $name(s) \in R_S$ and $name(p) \in R_S$, and with a special role **hasNarrower**; and
- X_S a set of axioms and assertions that define relationships between concepts, between roles, and between individuals and concepts or roles, with $assertions(s) \cup assertions(p) \subseteq X_S$.

The **hasNarrower** role is used to describe relationships between structural types, namely if one type is on a lower structural level than another. For example, a relationship **hasNarrower(Chapter, Paragraph)** indicates that chapters occur higher up in a document's structure than paragraphs. However, this modelling approach poses a conundrum for the semantics of the ontology, because semantics based on description logics do not allow for role-relationships between concepts.

Since the intended semantics are clear, we leave the formal semantics deliberately open at this point. Possible solutions include punning as introduced in OWL 2, which basically lifts the unique name assumption far enough to allow for the introduction of individuals as place holders for existing concepts, so that a role-relationship between concepts is "simulated" as a relationship between individuals of the same names as the concepts in question. Another possible solution is shown in section 5.3. Using concept equivalence or concept subsumption to model these relationships is not applicable because of the set semantics of concept subsumption: a paragraph can be narrower than a chapter (i.e., be on a lower structural level), but instances of **Paragraph** are usually not instances of **Chapter**.

\mathcal{O}_T is a terminological ontology that provides the terminology for modelling the content of the document. It makes use of the terms T of S as individuals, and defines

- C_T a set of concepts that classify terms, such as *Abbreviation* or *Person*;
- R_T a set of roles that define relations between terms, such as *synonym* or *broaderThan*; and
- X_T a set of axioms and assertions that define relationships between concepts, between roles, and between individuals and concepts or roles, such as *Person*(Alan Turing) or *synonym*(happy, joyful).

Both ontologies are domain dependent, but they do not depend on the same domain: \mathcal{O}_T depends on the domain of discourse, the content domain. \mathcal{O}_S , on the other hand, depends on the form of the document (such as “book”, “article”, or “e-learning document”), the presentation domain. For example, a book has a different structure and uses different structural elements than an e-learning document.

Example 4.2.15 (Semantic Document Model). Let S be the structural document model from example 4.2.4. Let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ be an ontology with

$$\begin{aligned}
C_S &= \{ \text{Document, Chapter, Paragraph, Definition, Example,} \\
&\quad \text{Illustration} \}, \\
R_S &= \{ \text{name}(s), \text{name}(p), \text{hasNarrower} \}, \text{ and} \\
X_S &= \{ \text{Definition} \sqsubseteq \text{Paragraph,} \\
&\quad \text{Example} \sqsubseteq \text{Paragraph,} \\
&\quad \text{Illustration} \sqsubseteq \text{Example} \} \cup \\
&\quad \{ \text{hasNarrower}(\text{Document, Chapter}), \\
&\quad \text{hasNarrower}(\text{Chapter, Paragraph}) \} \cup \\
&\quad \{ \text{Document}(f_0), \text{Definition}(f_{21}), \text{Example}(f_{31}), \\
&\quad \text{Definition}(f_{41}), \text{Paragraph}(f_{42}), \text{Illustration}(f_{42}) \} \cup \\
&\quad \{ \text{Chapter}(f_i) \mid \text{for } 1 \leq i \leq 5 \} \cup \\
&\quad \{ \text{Paragraph}(f_{i1}) \mid \text{for } 1 \leq i \leq 5 \} \cup \\
&\quad \text{assertions}(s) \cup \text{assertions}(p).
\end{aligned}$$

Let $\mathcal{O}_T = (C_T, R_T, T, X_T)$ be an ontology with

$$\begin{aligned}
C_T &= \{ \text{Term} \}, \\
R_T &= \{ \text{broaderThan} \}, \text{ and} \\
X_T &= \{ \text{Term}(\text{Data Structure}), \text{Term}(\text{Binary Tree}), \\
&\quad \text{broaderThan}(\text{Data Structure, Binary Tree}) \}.
\end{aligned}$$

Then $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ is a semantic document model.

The structure of D is illustrated in figure 4.8. The structural classification is shown in blue, and the terminology is shown in green.

We will now extend the notation for structural document models to semantic document models.

Definition 4.2.16 (Notation for Semantic Document Model). The basic syntax for semantic document models is shown as an EBNF grammar:

```

model      = fragment | "[" ]" ;
fragment   = id |
            id, "{", { annotations }, "}" |
            id, "[", { sub-fragments }, { "|", sub-fragments }, "]" |
            id, "{", { annotations }, "}",
            "[", { sub-fragments }, { "|", sub-fragments }, "]" ;
sub-fragments = fragment, { ",", sub-fragments } ;
annotations  = name, ":", "'", value, "'", { annotations } ;

```

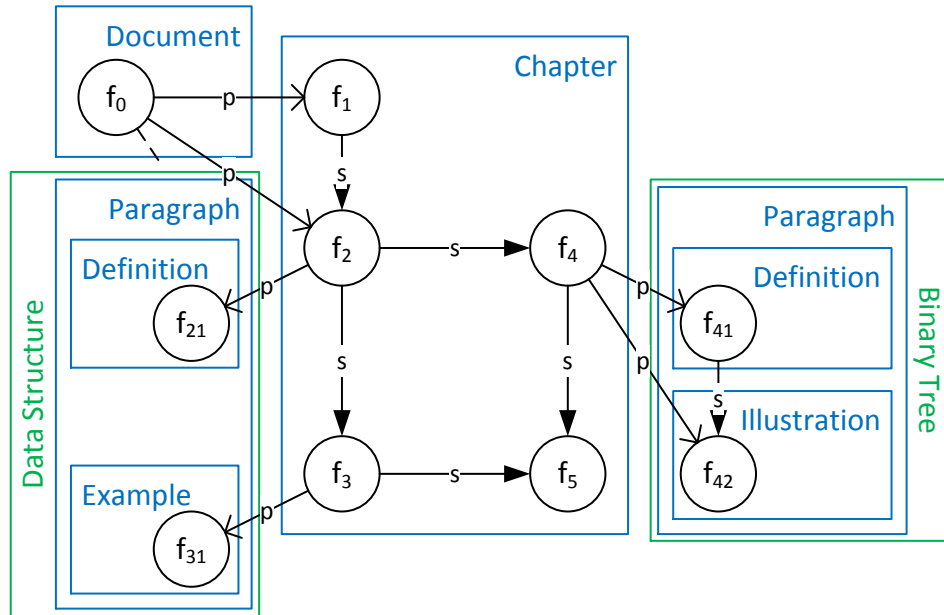


Figure 4.8: Partial illustration of a semantic document model

Fragments may now also contain annotations, a list of name-value pairs enclosed in “{ }”. Role assertions can be annotated by using the role name as the name, and the role value as the value. Concept assertions can be annotated by using the string “type” as the name and the concept name as the value.

Example 4.2.17 (Notation for Semantic Document Model). Let 1, 2, 3 be fragment identifiers. Then

- ▶ $1\{type : 'Document'\}$, and
- ▶ $1\{type : 'Document'\}[2\{type : 'Chapter', title : 'Trees'\}, 3]$.

are notations of semantic document models.

The operations defined on structural document models can be transferred directly to semantic document models. We will now define three additional operations for inserting, deleting, and changing annotations.

Definition 4.2.18 (Operations on Semantic Document Models). Let F be the set of document fragments, let N be the set of annotation names, and let V be the set of annotations values.

$+_a : F \times N \times V \rightarrow F$ is an operation that adds an annotation consisting of a name and a value to a fragment. If the annotation already exists in the fragment, the fragment remains unchanged.

$-_a : F \times N \times V \rightarrow F$ is an operation that removes an annotation consisting of a name and a value from a fragment. If the annotation does not exist in the fragment, the fragment remains unchanged.

$\#_a : F \times N \times V \times V \rightarrow F$ is an operations that changes the value of an annotation consisting of a name and a value in a fragment by replacing the existing value with a new one. If the annotation does not exist in the fragment, the fragment remains unchanged.

Example 4.2.19 (Operations on Semantic Document Models). Let 1 be a fragment identifier, let *type* and *title* be annotation names, and let ‘Document’, ‘Chapter’, ‘Trees’, and ‘Binary Trees’ be annotation values. Then

- ▶ $+_a(1, \text{type}, \text{‘Document’}) = 1\{\text{type} : \text{‘Document’}\},$
- ▶ $-_a(1\{\text{type} : \text{‘Chapter’}, \text{title} : \text{‘Trees’}\}, \text{title}, \text{‘Trees’}) = 1\{\text{type} : \text{‘Chapter’}\},$ and
- ▶ $\#_a(1\{\text{title} : \text{‘Trees’}\}, \text{title}, \text{‘Trees’}, \text{‘Binary Trees’}) = 1\{\text{title} : \text{‘Binary Trees’}\}.$

We will now define what it means that a structural ontology is *compatible* with a semantic document model.

Definition 4.2.20 (Compatibility of a Structural Ontology with a Semantic Document Model). Let $\mathcal{O}'_S = (C'_S, R'_S, I'_S, X'_S)$ be a structural ontology. Let $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be a semantic document model. Then \mathcal{O}'_S is *compatible* with D iff

1. $(\exists t_1, t_2 \in C'_S : \text{hasNarrower}(t_1, t_2) \in X'_S \vee (t_1 \equiv t_2) \in X'_S) \Rightarrow \exists f_1, f_2 \in F : t_1(f_1), t_2(f_2) \in X_S \wedge (f_2, f_1) \in p,$ i.e., no fragment is a sub-fragment of another fragment with a narrower or equivalent type (the document hierarchy respects the type hierarchy), and
2. the combined ontology $\mathcal{O}''_S = \mathcal{O}_S \cup \mathcal{O}'_S = (C_S \cup C'_S, R_S \cup R'_S, I_S \cup I'_S, X_S \cup X'_S)$ does not contain a logical contradiction, i.e., $\mathcal{O}''_S \not\models \perp.$

Let $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be a semantic document model and let $\mathcal{O}'_S = (C'_S, R'_S, I'_S, X'_S)$ be a structure ontology that is compatible with D . Then both $D' = (F, f_0, s, p, T, c, \mathcal{O}'_S, \mathcal{O}_T)$ and $D'' = (F, f_0, s, p, T, c, \mathcal{O}_S \cup \mathcal{O}'_S, \mathcal{O}_T)$ are semantic document models.

Example 4.2.21 (Compatibility of a Structure Ontology with a Semantic Document Model). The empty ontology is compatible with any semantic document model. The structure ontology from example 4.2.15 is compatible with the semantic document model from the same example.

Let \mathcal{O}'_S be an ontology that contains the axiom *Definition* $\sqsubseteq \perp$. Then \mathcal{O}'_S is not compatible with the semantic document model from example 4.2.15, because $\mathcal{O}_S \cup \mathcal{O}'_S$ contain a contradiction. Let \mathcal{O}''_S be an ontology with

$$\begin{aligned} C_S &= \{\text{Table}, \text{Cell}\}, \\ R_S &= \{\text{name}(s), \text{name}(p), \text{hasNarrower}\}, \text{ and} \\ X_S &= \{\text{hasNarrower}(\text{Table}, \text{Cell})\} \cup \\ &\quad \{\text{Table}(f_{21}), \text{Cell}(f_2)\} \cup \\ &\quad \text{assertions}(s) \cup \text{assertions}(p). \end{aligned}$$

Then \mathcal{O}''_S is not compatible with the semantic document model from example 4.2.15, because f_{21} is a sub-fragment of f_2 , but its type is broader than that of f_2 .

It is possible that new structural knowledge arises after a semantic document model has been created. In such a case it can be advantageous to combine the new knowledge with the existing knowledge. This is only useful and permissible if the new knowledge, in the form of a structure ontology, is compatible with the existing semantic document model. It can also be useful to check the reciprocal compatibility of the structure of two different semantic document models to see

if the models are structurally similar. If the structure ontology of the first model is compatible with the second model, and if the structure ontology of the second model is compatible with the first model, then it stands to reason that the models have a very similar structure.

We will now start to regard ways of formally representing and of implementing semantic document models.

Lemma 4.2.22 (Semantic Document Models in Description Logics). *For every semantic document model $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ there exists a document ontology \mathcal{O}_D .*

As a convenience, we extend the $name()$ function (cf. definition 4.2.10) from relations to objects, individuals, concepts and roles in an ontology \mathcal{O} : for every relation, object, individual, and atomic or symbolic concept or role name, $name()$ returns a new atomic role name, individual, or atomic or symbolic concept or role name, respectively, that is not contained in \mathcal{O} . For example, $name(\text{chapter2}) = \text{chapter2}'$ and $name(\text{hasPart}) = \text{hasPart}'$.

We also introduce the function $names()$ that applies $name()$ to assertions.

Definition 4.2.23 (Function $names()$). *$names()$ replaces every occurrence of individuals, and of atomic or symbolic concept or role names n with $name(n)$ in assertions and axioms.*

Example 4.2.24 (Function $names()$). *For example, $names(x) = \text{hasPart}'(\text{chapter2}', \text{paragraph2.1}')$ for an assertion $x = \text{hasPart}(\text{chapter2}, \text{paragraph2.1})$.*

With these preliminaries, we can now prove lemma 4.2.22.

Proof of Lemma 4.2.22. Let $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be a semantic document model, with $\mathcal{O}_S = (C_S, R_S, F, X_S)$ and $\mathcal{O}_T = (C_T, R_T, T, X_T)$.

Then an ontology $\mathcal{O}_D = (C_D, R_D, I_D, X_D)$ can be constructed as follows:

$$\begin{aligned} C_D &:= \{name(c_S) \mid c_S \in C_S\} \cup \{name(c_T) \mid c_T \in C_T\} \\ R_D &:= \{name(r_S) \mid r_S \in R_S \setminus \{\text{hasNarrower}\}\} \cup \\ &\quad \{name(r_T) \mid r_T \in R_T\} \cup \{name(c)\} \\ I_D &:= \{name(i_F) \mid i_F \in F\} \cup \{name(i_T) \mid i_T \in T\} \\ X_D &:= \{names(x_S) \mid x_S \in X_S\} \cup \{names(x_T) \mid x_T \in X_T\} \\ &\quad \cup \text{assertions}(c) \end{aligned}$$

Intuitively, we construct an ontology from individuals, types, and relationships in D by creating new symbols, concepts, and roles in \mathcal{O}_D . \square

Proposition 4.2.25 (Interpretation of \mathcal{O}_D). *For every semantic document model $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ and document ontology $\mathcal{O}_D = (C_D, R_D, I_D, X_D)$ constructed from D , there exists an interpretation $I(D) = (\Delta^I, ()^I)$, such that $I(D)$ is a model of \mathcal{O}_D in the sense of definition 3.2.31: $I(D) \models \mathcal{O}_D$. In other words, D can be seen as a model of \mathcal{O}_D .*

Proof of Proposition 4.2.25. The interpretation $I(D) = (\Delta^I, ()^I)$ can be defined in a natural way as follows:

$$\begin{aligned} \Delta^I &:= F \cup T \\ \forall c' \in C_D : c'^I &:= \begin{cases} \{f \in F \mid c'(f') \in X_D \wedge name(f) = f'\} \\ \quad \text{iff } c \in C_S \text{ and } name(c) = c' \\ \{t \in T \mid c'(t') \in X_D \wedge name(t) = t'\} \\ \quad \text{iff } c \in C_T \text{ and } name(c) = c' \\ \emptyset & \text{otherwise} \end{cases} \end{aligned}$$

$$\forall r' \in R_D : r'^I := \begin{cases} \{(f_1, f_2) \in F \times F \mid \\ r'(f_1, f_2) \in X_D \wedge \text{name}(f_1) = f_1' \wedge \text{name}(f_2) = f_2' \\ \text{iff } r \in R_S \text{ and } \text{name}(r) = r'\} \\ \\ \{(t_1, t_2) \in T \times T \mid \\ r'(t_1, t_2) \in X_D \wedge \text{name}(t_1) = t_1' \wedge \text{name}(t_2) = t_2' \\ \text{iff } r \in R_T \text{ and } \text{name}(r) = r'\} \\ \\ \{(f, t) \in F \times T \mid (f, t) \in c\} \\ \text{iff } \text{name}(c) = r' \\ \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$\forall i' \in I_D : i'^I := \begin{cases} f \in F & \text{iff } \text{name}(f) = i' \\ t \in T & \text{iff } \text{name}(t) = i' \\ \text{undefined} & \text{otherwise} \end{cases}$$

$I(D) \models \mathcal{O}_D$ follows directly from the semantics of $I(D)$ and from the construction of \mathcal{O}_D in Lemma 4.2.22. \square

Example 4.2.26 (Interpretation of \mathcal{O}_D). *Let $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be the semantic document model from example 4.2.15. $\mathcal{O}_D = (C_D, R_D, I_D, X_D)$ is a document ontology constructed from D with*

$$\begin{aligned} C_D &= \{\text{Document}', \text{Chapter}', \text{Paragraph}', \text{Definition}', \text{Example}', \\ &\quad \text{Illustration}', \text{Term}'\}, \\ R_D &= \{s', p', \text{broaderThan}', c\}, \\ I_D &= \{f'_0, f'_1, f'_2, f'_3, f'_4, f'_5, f'_{21}, f'_{31}, f'_{41}, f'_{42}\} \cup \\ &\quad \{\text{Data Structure}', \text{Binary Tree}'\}, \text{ and} \\ X_D &= \{\text{Definition}' \sqsubseteq \text{Paragraph}', \\ &\quad \text{Example}' \sqsubseteq \text{Paragraph}', \\ &\quad \text{Illustration}' \sqsubseteq \text{Example}'\} \cup \\ &\quad \{\text{Document}'(f'_0), \text{Definition}'(f'_{21}), \text{Example}'(f'_{31}), \\ &\quad \text{Definition}'(f'_{41}), \text{Paragraph}'(f'_{42}), \text{Illustration}'(f'_{42})\} \cup \\ &\quad \{\text{Chapter}'(f'_i) \mid \text{for } 1 \leq i \leq 5\} \cup \\ &\quad \{\text{Paragraph}'(f'_{i1}) \mid \text{for } 1 \leq i \leq 5\} \cup \\ &\quad \{\text{Term}'(\text{Data Structure}'), \text{Term}'(\text{Binary Tree}'), \\ &\quad \text{broaderThan}(\text{Data Structure}', \text{Binary Tree}')\} \cup \\ &\quad \{s'(f'_1, f'_2), s'(f'_2, f'_3), s'(f'_2, s'f'_4), s'(f'_3, f'_5), s'(f'_4, f'_5), \\ &\quad s'(f'_{41}, f'_{42})\} \cup \\ &\quad \{p'(f'_0, f'_i) \mid \text{for } 1 \leq i \leq 5\} \cup \\ &\quad \{p'(f'_2, f'_{21}), p'(f'_3, f'_{31}), p'(f'_4, f'_{41}), p'(f'_4, f'_{42})\} \cup \\ &\quad \{c(f'_{21}, \text{Data Structure}'), c(f'_{31}, \text{Data Structure}'), \\ &\quad c(f'_{41}, \text{Binary Tree}'), c(f'_{42}, \text{Binary Tree}')\}. \end{aligned}$$

Then $I(D) = (\Delta^I, ()^I)$ is a model of D , where

$$\begin{aligned}
\Delta^I &= \{f_0, f_1, f_2, f_3, f_4, f_5, f_{21}, f_{31}, f_{41}, f_{42}\} \cup \\
&\quad \{\text{Data Structure, Binary Tree}\}; \\
f_0^I &= f_0, f_1^I = f_1, f_2^I = f_2, f_3^I = f_3, f_4^I = f_4, f_5^I = f_5, \\
f_{21}^I &= f_{21}, f_{31}^I = f_{31}, f_{41}^I = f_{41}, f_{42}^I = f_{42}, \\
(\text{Data Structure}')^I &= \text{Data Structure}, \\
(\text{Binary Tree}')^I &= \text{Binary Tree}; \\
(\text{Document}')^I &= \{f_0\}, \\
(\text{Chapter}')^I &= \{f_1, f_2, f_3, f_4, f_5\}, \\
(\text{Paragraph}')^I &= \{f_{21}, f_{31}, f_{41}, f_{42}\}, \\
(\text{Definition}')^I &= \{f_{21}, f_{41}\}, \\
(\text{Example}')^I &= \{f_{31}\}, \\
(\text{Illustration}')^I &= \{f_{42}\}, \\
(\text{Term}')^I &= \{\text{Data Structure, Binary Tree}\}; \text{ and} \\
(s')^I &= \{(f_1, f_2), (f_2, f_3), (f_2, f_4), (f_3, f_5), (f_4, f_5), (f_{41}, f_{42})\}, \\
(p')^I &= \{(f_0, f_1), (f_0, f_2), (f_0, f_3), (f_0, f_4), (f_0, f_5), \\
&\quad (f_2, f_{21}), (f_3, f_{31}), (f_4, f_{41}), (f_4, f_{42})\}, \\
c^I &= \{(f_{21}, \text{Data Structure}), (f_{31}, \text{Data Structure}), \\
&\quad (f_{41}, \text{Binary Tree}), (f_{42}, \text{Binary Tree})\}.
\end{aligned}$$

A consequence of proposition 4.2.25 is that an ontology \mathcal{O}_D can be used as an abstract implementation of a semantic document model D , with a concrete implementation in a description logic compatible language such as OWL.

Note that the two ontologies contained in D (\mathcal{O}_S and \mathcal{O}_T) serve as part of the model of the document ontology \mathcal{O}_D , and are thus part of an abstraction of \mathcal{O}_D .

Another direct consequence of proposition 4.2.25 is that \mathcal{O}_D provides a formal semantics based on description logics for the semantic document model D .

In practical applications, the strict separation between model and instance can be relaxed, so that both \mathcal{O}_S and \mathcal{O}_T can serve as their own interpretation. Assuming that the sets of individuals, concepts and roles are pairwise disjoint between \mathcal{O}_S and \mathcal{O}_T , this allows us to use them directly in \mathcal{O}_D without any cumbersome renaming.

Recall figure 4.6 for an overview of how the different models and ontologies fit together.

Note that the semantic abstraction from the base document model to the semantic document model is not always possible: while a connected base document model is required to have a starting object, this root is not necessarily unique (cf. remark 4.1.31). However, multiple roots violate the tree structure of the semantic document model. Yet it is possible to split a base document model with multiple (potential) starting objects into multiple models, each with only one of the starting objects and duplicates of all other objects reachable from that root. This modelling approach also appears to be closer to the actual semantics of a document with multiple entry points than a combined model would be.

It is also possible that a document simply does not adhere to the structural conventions defined in definition 4.2.1. For example, a document might contain a chapter, which contains a paragraph that in turn contains the chapter itself (cyclic structure). Such documents cannot be analysed with the methods developed in this thesis.

Adequacy and Relation to Other Document Models

As already seen in definition 4.1.32, hypertexts can be represented as connected base document models. With this alone, many current document formats can be modelled on a technical level. Documents with hypertext characteristics like HTML, e-books, Microsoft Word, DocBook, DITA, and even PDF and L^AT_EX, can be represented directly. For documents with no apparent reading

order like some kinds of diagrams or spreadsheets that can be read in any order, a starting point has to be defined before they can be modelled. If the document has an inherent tree structure, its model can then be transformed into a semantic document model as described in section 5.1.

Other document models also start with hypertexts. For example, [Gar87, HS94, SFC98] model hypertexts as automata, i.e., as transition systems. This model is compatible with connected base document models.

Other models map the hypertext structure onto Petri Nets [SF89, vdA03, OVvdA⁺07]. A Petri Net N is a tuple (S, T, F) with $S = \{s_1, \dots, s_n\}$ a set of states, $T = \{t_1, \dots, t_m\}$ a set of transitions, and $F \subseteq (S \times T) \cup (T \times S)$ the control flow. [SF89] also include the document's content, which is mapped onto S , and its visualisation. Such a Petri Net can be represented as a base document model or as a structural document model, with two structure classes to represent states and transitions. Verifying control flow properties on the document model is not in the scope of this work, so the Petri Net semantics cannot be represented in either a base document model or a structural document model. It is, however, possible to generate suitable models from a semantic document model, as will be discussed in section 5.2.

[Jel02, NCEF02] represent documents in XML, in order to run XPath-based verification techniques on them. [ABF04] maps XML documents onto a term algebra, with elements and attributes as functions, and values as constants. These XML-based documents can be modelled as base document models like any other XML format, with the XML structure as a good starting point for more structured models.

The Text Encoding Initiative allows for document models that resemble instances of simplified semantic document models. They provide a large number of structural classes, but no semantic relations between them. Their goal is to represent documents, but not to process them further. [TEI07]

In [ESS05], document models based on description logics are introduced. While they model terminological relations, the document structure is disregarded entirely. As a semantic document model, this would result in a model with a single document fragment and no structural ontology, with all terms attached to the single document fragment.

[Wei08] partly addresses this problem by introducing a temporal description logic that allows the modelling of a simple hypertext-like structure for documents. We will revisit this formalism later to see how it can be combined with our approach for mutual benefit.

[PBB11] is mostly concerned with the visualisation aspects of documents, trying to model them in a way that a visualisation can be derived from the model that is either very close to the original document, or that is best suited for a specific context like an e-book reader. The document model consists of a tree structure of renderings down to small atomic entities. While this model could be represented as a semantic document model, with the renderings as atomic document fragments, there is little inherent value in doing so, because this model would not contain any semantic relations.

Other document models are based on a “bag of words” [BNJ03] or are token-based [Cow06]. Pure information retrieval models like the Standard Boolean Model [LF73] or the Vector Space Model [SWY75] indicate the existence or frequency, respectively, of terms in documents. Such information can be modelled as attached metadata in base document models.

An important limitation of semantic document models is that they allow for only a single dominant structuring of a document. If a document has two possible structures, either one of them must be chosen, or two document models must be created. Regard, as an example, a set of musical notes as shown in figure 4.9. Here, a piece of sheet music (a) is shown that can be structured in at least two sensible ways. It consists of eight notes in two bars (indicated by the vertical lines). The first six notes are played in low volume (piano), while the final two notes are played loudly (forte). These notes can either be structured by bar (b), with two bars of four

notes each as is briefly illustrated in example 4.2.27. Or it can be structured by volume, with six notes in the piano section and two notes in the forte section, as illustrated in example 4.2.28. It is, however, not possible to integrate both structurings into a single semantic document model.

This allows us to focus on a single dominant structure and keeps the model complexity manageable. Multiple structural hierarchies in a single model would require multiple sets F_i and multiple relations s_i and p_i . The same effect can be achieved more easily by defining multiple models and re-using one or both ontologies \mathcal{O}_S and \mathcal{O}_T , as shown below.

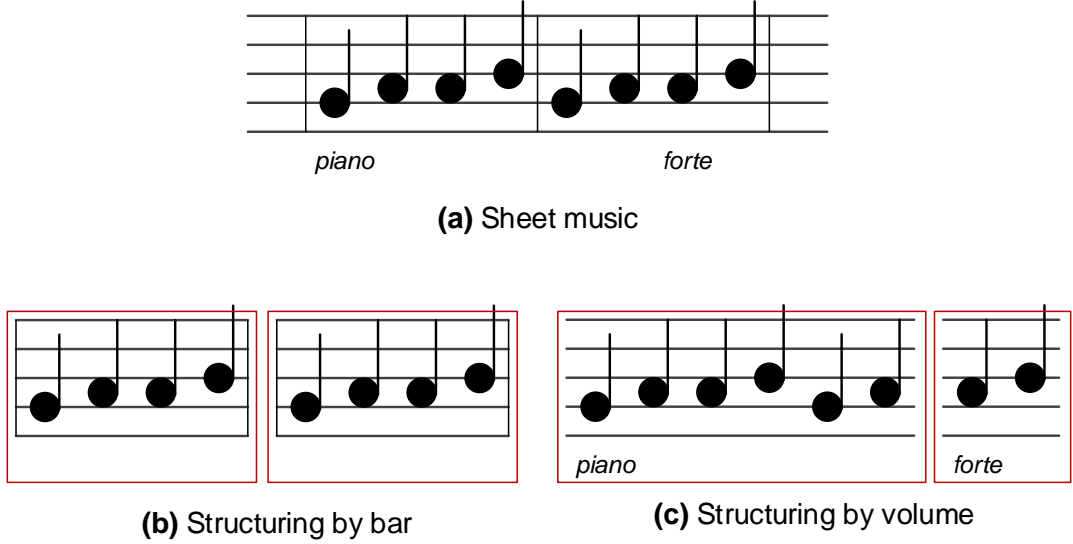


Figure 4.9: Structure of musical notes

Example 4.2.27 (Musical Notes by Bar). Let $D_B = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be a semantic document model with

$$\begin{aligned}
 F &= \{f_0, f_1, f_2, f_{11}, f_{12}, f_{13}, f_{14}, f_{21}, f_{22}, f_{23}, f_{24}\}; \\
 s &= \{(f_1, f_2), (f_{11}, f_{12}), (f_{12}, f_{13}), (f_{13}, f_{14}), (f_{21}, f_{22}), (f_{22}, f_{23}), (f_{23}, f_{24})\}; \\
 p &= \{(f_0, f_1), (f_0, f_2), (f_1, f_{11}), (f_1, f_{12}), (f_1, f_{13}), (f_1, f_{14}), \\
 &\quad (f_2, f_{21}), (f_2, f_{22}), (f_2, f_{23}), (f_2, f_{24})\}; \\
 T &= \{G, A, B, \text{piano}, \text{forte}\}; \text{ and} \\
 c &= \{(f_{11}, G), (f_{12}, A), (f_{13}, A), (f_{14}, B), (f_{21}, G), (f_{22}, A), (f_{23}, A), (f_{24}, B), \\
 &\quad (f_{11}, \text{piano}), (f_{12}, \text{piano}), (f_{13}, \text{piano}), (f_{14}, \text{piano}), \\
 &\quad (f_{21}, \text{piano}), (f_{22}, \text{piano}), (f_{23}, \text{forte}), (f_{24}, \text{forte})\}.
 \end{aligned}$$

Let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ be an ontology with

$$\begin{aligned}
 C_S &= \{\text{Document}, \text{Bar}, \text{Note}\}, \\
 R_S &= \{\text{name}(s), \text{name}(p), \text{hasNarrower}\}, \text{ and} \\
 X_S &= \{\text{hasNarrower}(\text{Document}, \text{Bar}), \text{hasNarrower}(\text{Bar}, \text{Note})\} \cup \\
 &\quad \{\text{Document}(f_0), \text{Bar}(f_1), \text{Bar}(f_2), \\
 &\quad \text{Note}(f_{11}), \text{Note}(f_{12}), \text{Note}(f_{13}), \text{Note}(f_{14}), \\
 &\quad \text{Note}(f_{21}), \text{Note}(f_{22}), \text{Note}(f_{23}), \text{Note}(f_{24})\} \cup \\
 &\quad \text{assertions}(s) \cup \text{assertions}(p).
 \end{aligned}$$

Let $\mathcal{O}_T = (C_T, R_T, T, X_T)$ be an ontology with

$$\begin{aligned}
C_T &= \{NoteName, Volume\}, \\
R_T &= \{lowerThan\}, \text{ and} \\
X_T &= \{NoteName(G), NoteName(A), NoteName(B), \\
&\quad lowerThan(G, A), lowerThan(A, B), \\
&\quad Volume(piano), Volume(forte)\}.
\end{aligned}$$

D_B is a model after figure 4.9 (a), modelling the structure by bar:

$$0[1[11, 12, 13, 14], 2[21, 22, 23, 24]]$$

Example 4.2.28 (Musical Notes by Volume). Let $D_V = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be a semantic document model with \mathcal{O}_T from example 4.2.27, and

$$\begin{aligned}
F &= \{f_0, f_1, f_2, f_{11}, f_{12}, f_{13}, f_{14}, f_{15}, f_{16}, f_{21}, f_{22}\}; \\
s &= \{(f_1, f_2), (f_{11}, f_{12}), (f_{12}, f_{13}), (f_{13}, f_{14}), (f_{14}, f_{15}), (f_{15}, f_{16}), (f_{21}, f_{22})\}; \\
p &= \{(f_0, f_1), (f_0, f_2), (f_1, f_{11}), (f_1, f_{12}), (f_1, f_{13}), (f_1, f_{14}), \\
&\quad (f_1, f_{15}), (f_1, f_{16}), (f_2, f_{21}), (f_2, f_{22})\}; \\
T &= \{G, A, B, piano, forte\}; \text{ and} \\
c &= \{(f_{11}, G), (f_{12}, A), (f_{13}, A), (f_{14}, B), (f_{21}, G), (f_{22}, A), (f_{23}, A), (f_{24}, B), \\
&\quad (f_1, piano), (f_2, forte)\}.
\end{aligned}$$

Let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ be an ontology with

$$\begin{aligned}
C_S &= \{Document, Section, Note\}, \\
R_S &= \{name(s), name(p), hasNarrower\}, \text{ and} \\
X_S &= \{hasNarrower(Document, Section), hasNarrower(Section, Note)\} \cup \\
&\quad \{Document(f_0), Section(f_1), Section(f_2), \\
&\quad Note(f_{11}), Note(f_{12}), Note(f_{13}), Note(f_{14}), \\
&\quad Note(f_{21}), Note(f_{22}), Note(f_{23}), Note(f_{24})\} \cup \\
&\quad assertions(s) \cup assertions(p).
\end{aligned}$$

D_V is a model after figure 4.9 (b), modelling the structure by volume:

$$0[1[11, 12, 13, 14, 15, 16], 2[21, 22]]$$

Outlook

One other thing that we cannot grasp with a semantic document model are different layers of truth, fact, supposition, and fiction. Imagine, for example, a novel. It will likely start with a page listing in terse language all pertinent information about copyright, date and place of publication, previous editions, and other things. This information can be regarded as fact (for now). Then might follow a preface by the author, making assertions about the progression of her work and offering thanks to her supporters. These statements can be regarded as supposition. After that begins the main event, the actual novel that consists of some fictitious account. And within this account, a character might relate a tale, or might recount the contents of a (fictitious) book. This could be called “second order fiction”, as it is fiction within fiction. We do not even touch upon mixed forms like allegories, parodies, or simple references to factual occurrences from within fictional context. Nor do we differentiate between “realistic” fiction, nonsensical fiction, and everything in between. We also ignore whether or not we can actually decide (and agree with others) if something is fact or fiction, including the entire realm of religious faith.

The reason why we cannot properly represent these layers of reality in semantic document models is simply because the underlying logic is not adequate for the task. In formal logics, an assertion is usually either true or false, even if we do not know which. Probabilistic logics extend this with a probabilistic or statistical component, stating that under given preconditions, an assertion has a certain probability of being true, or that in general a certain percentage of

assertions is true. However, the truth value itself is still binary – only the degree of uncertainty which of the two values is the correct one is stated explicitly.

Fuzzy logics take a different approach: here, the actual degree of truth is modelled. For an assertion like “Sherlock Holmes is a detective” to have a fuzzy logic value of 0.5 means that it is only half right, i.e., Sherlock Holmes is only half a detective and half something else. Yet this is clearly also not sufficient for modelling layers of reality, because Mr. Holmes is a *fictional* detective, not *half* a detective.

While it is possible to regard different layers of reality in separate models, it is hard to allow them to interact: to have explicit or implicit references from a fictional context to reality or vice versa. Modal logics can be used to model multiple states of knowledge, for example an agent who has a certain model of the state of the world can also know the (different) models as “known” by other agents. This leads to statements like “She knows that he knows that she knows [something]”. However, these logics are based on properties that are generally not realistic in a real-world setting, such as knowing that one does not know a specific thing. [HR04] More work is needed here to fully understand and deal with these complexities.

4.2.2 Modelling Processes

While at first glance, processes and documents have little in common, a closer examination yields remarkable similarities as will be shown below.

We will use the term *process* to mean an ordered, but not necessarily totally ordered, collection of process steps. Each process step can be either a process itself (namely, a sub-process), or an atomic action such as a service call. A definition of various types of process steps can be found in [All10].

Figure 4.10 shows an example process for processing an application for vacation. It starts with filling in the application form. Then, if the applicant requires a substitute to fill in for them, a suitable substitute is chosen who must be available during the relevant time. With the substitute’s signature, the application is now checked by the direct superior of the applicant. If the superior rejects the application, the applicant is informed and the process ends. Otherwise, the superior adds their signature to the application, and it moves on to the human resource department. If they reject the application, both the superior and the applicant are informed and the process ends. If they approve, the application is stamped, the vacation time is entered into a database, the applicant is informed, and the process ends. The checking by the human resource department is modelled as a sub-process, which is not specified further in this example.

Both documents and processes are structured in that they consist of an ordering of elements: the reading paths for documents, the control flow in processes. This order can also be hierarchical: sub-chapters in documents, or sub-processes in processes. In addition, both deal with data: the document content, topics and relevant terms for documents, the data flow in processes. And both deal with data coherence, like sections that are semantically related to each other for documents, or consistent data usage in processes. We will therefore attempt to model (and later: process) processes in a similar way to documents. This shows that our approach for documents can be transferred to other domains as well.

As with documents, we will first define a low-level process model, the *connected base process model*.

Definition 4.2.29 (Connected Base Process Model). *Let*

- $S = \{s_\alpha, \dots, s_\omega\}$ be a set of process steps,
- $s_\alpha \in S$ be the starting step,
- $s_\omega \in S$ be the final step, and
- $s \subseteq S \times S$ be a relation that assigns successors to process steps.

Then $B = (S, s_\alpha, s_\omega, s)$ is a base process model.

A base process model $B = (S, s_\alpha, s_\omega, s)$ is connected, i.e., is a connected base process model, iff

$$\forall s_x \in S \setminus \{s_\alpha\} : \exists c = (s_\alpha, \dots, s_x) \text{ (all steps are reachable from } s_\alpha), \text{ and}$$

$$\forall s_x \in S \setminus \{s_\omega\} : \exists c = (s_x, \dots, s_\omega) \text{ (all steps can reach } s_\omega),$$

where c is a control path in B (see below).

Note that definition 4.2.29 introduces a final step s_ω that was not used in definition 4.1.29.

Definition 4.2.30 (Control Path). A control path c on a connected base process model $B = (S, s_\alpha, s_\omega, s)$ is a non-empty, possibly infinite sequence of process steps (s_0, s_1, s_2, \dots) , for which holds that $s(s_i, s_{i+1})$, for $i \in \mathbb{N}$.

A finite control path $c_f = (s_0, \dots, s_n)$ is called closed iff $s_0 = s_\alpha$ and $s_n = s_\omega$.

The set of all closed control paths on a connected base process model B is called the control flow of B .

Example 4.2.31 (Connected Base Process Model). Let

$$\begin{aligned} S &= \{s_\alpha, s_{\text{fill in application}}, s_{\text{requires substitute?}}, s_{\text{choose substitute}}, s_{\text{substitute available?}}, \\ &\quad s_{\text{check by direct superior}}, s_{\text{direct superior approved?}}, s_{\text{check by HR department}}, \\ &\quad s_{\text{HR approved?}}, s_{\text{inform direct superior of rejection}}, s_{\text{inform applicant of rejection}}, \\ &\quad s_{\text{enter data into database}}, s_{\text{inform applicant of approval}}, s_\omega\}, \text{ and} \\ s &= \{(s_\alpha, s_{\text{fill in application}}), (s_{\text{fill in application}}, s_{\text{requires substitute?}}), \\ &\quad (s_{\text{requires substitute?}}, s_{\text{choose substitute}}), \\ &\quad (s_{\text{requires substitute?}}, s_{\text{check by direct superior}}), \dots\}. \end{aligned}$$

Then $B = (S, s_\alpha, s_\omega, s)$ is a connected base process model.

Example 4.2.32 (Control Path). Let $B = (S, s_\alpha, s_\omega, s)$ be the connected base process model from example 4.2.31. Then

$$\begin{aligned} c_1 &= (s_{\text{requires substitute?}}, s_{\text{choose substitute}}, s_{\text{substitute available?}}), \\ c_2 &= (s_{\text{choose substitute}}, s_{\text{substitute available?}}, s_{\text{choose substitute}}, \dots), \text{ and} \\ c_3 &= (s_\alpha, s_{\text{fill in application}}, s_{\text{requires substitute?}}, s_{\text{check by direct superior}}, \\ &\quad s_{\text{direct superior approved?}}, s_{\text{inform applicant of rejection}}, s_\omega) \end{aligned}$$

are control paths on B . c_2 is infinite, and c_3 is closed.

We will now define more abstract process models, namely the *structural process model* and the *semantic process model*.

Definition 4.2.33 (Structural Process Model). Let

$$\begin{aligned} F &= \{f_0, \dots, f_n\} \text{ be a set of process fragments, i.e., atomic and} \\ &\quad \text{non-atomic process steps;} \\ f_0 &\in F \text{ be a starting fragment from } F \text{ with respect to both } p \text{ and } s; \\ s &\in R \text{ be a relation } s \subseteq F \times F \text{ between process fragments that defines} \\ &\quad \text{successors for fragments;} \\ p &\in R \text{ be a relation } p \subseteq F \times F \text{ between process fragments that defines} \\ &\quad \text{a hierarchy on fragments;} \\ D &\text{ be a set of data objects; and} \\ c &\text{ be a relation } c \subseteq F \times D \text{ that specifies which process fragments deal} \\ &\quad \text{with which data objects.} \end{aligned}$$

From the starting object f_0 , every fragment $f \in F$ must be reachable through p in exactly one unique sequence of applications of p , either directly or indirectly.

Then $S = (F, f_0, s, p, D, c)$ is a structural process model.

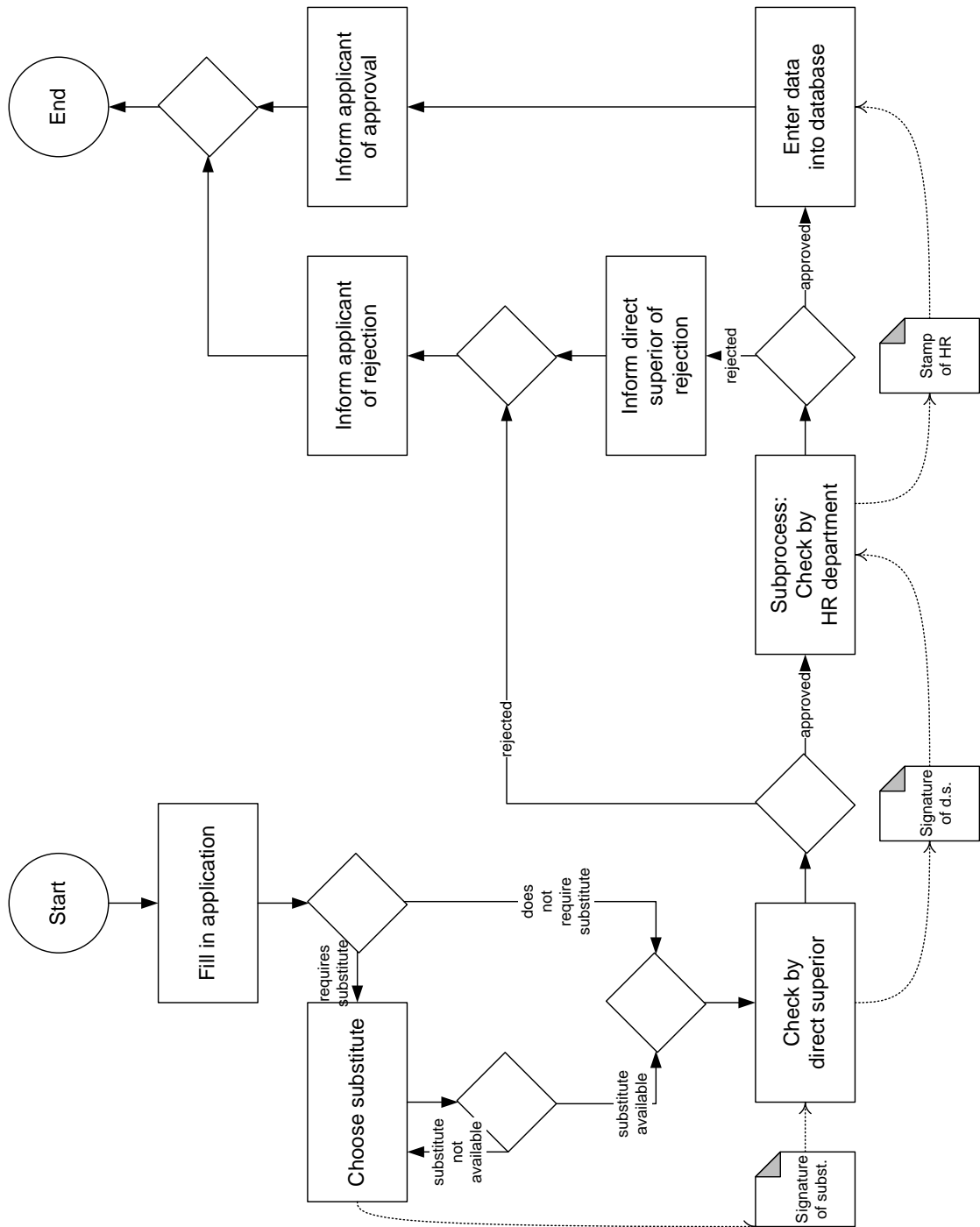


Figure 4.10: Example process: application for leave

Example 4.2.34 (Structural Process Model). *Let*

$$\begin{aligned}
F &= \{f_0, f_{start}, f_{fill \text{ in application}}, f_{requires \text{ substitute?}}, f_{choose \text{ substitute}}, \\
&\quad f_{substitute \text{ available?}}, f_{check \text{ by direct superior}}, f_{direct \text{ superior approved?}}, \\
&\quad f_{check \text{ by HR department}}, f_{HR \text{ approved?}}, f_{inform \text{ direct superior of rejection}}, \\
&\quad f_{inform \text{ applicant of rejection}}, f_{enter \text{ data into database}}, \\
&\quad f_{inform \text{ applicant of approval}}, f_{end}\}; \\
s &= \{(f_{start}, f_{fill \text{ in application}}), (f_{fill \text{ in application}}, f_{requires \text{ substitute?}}), \dots\}; \\
p &= \{(f_0, f_i) \mid \forall f_i \in F \setminus \{f_0\}\}; \\
T &= \{\text{Signature of substitute, Signature of direct superior, Stamp of HR}\}; \\
&\text{and} \\
c &= \{(f_{choose \text{ substitute}}, \text{Signature of substitute}), \\
&\quad (f_{check \text{ by direct superior}}, \text{Signature of substitute}), \\
&\quad (f_{check \text{ by direct superior}}, \text{Signature of direct superior}), \\
&\quad (f_{check \text{ by HR department}}, \text{Signature of direct superior}), \\
&\quad (f_{check \text{ by HR department}}, \text{Stamp of HR}), \\
&\quad (f_{enter \text{ data into database}}, \text{Stamp of HR})\}.
\end{aligned}$$

Then $S = (F, f_0, s, p, T, c)$ is a structural process model based on the process shown in figure 4.10.

Definition 4.2.35 (Semantic Process Model). *Let* $S = (F, f_0, s, p, D, c)$ be a structural process model. Let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ and $\mathcal{O}_D = (C_D, R_D, D, X_D)$ be two ontologies. Then $P = (F, f_0, s, p, D, c, \mathcal{O}_S, \mathcal{O}_D)$ is a semantic process model.

\mathcal{O}_S is a structural ontology that provides the terminology for modelling the structure of the process. It makes use of the process fragments F of P as individuals, and defines

- C_S a set of concepts that describe structural elements of the process, such as *Subprocess* or *ServiceCall*;
- R_S a set of roles that define relations between fragments, with $\text{name}(s) \in R_S$ and $\text{name}(p) \in R_S$ and with a special role *hasNarrower* that defines relationships between structural concepts; and
- X_S a set of axioms and assertions that define relationships between concepts, between roles, and between individuals and concepts or roles, with $\text{assertions}(s) \cup \text{assertions}(p) \subseteq X_S$.

\mathcal{O}_D is a data ontology that provides information about the data that is used in the process. It makes use of the data objects D of P as individuals, and defines

- C_D a set of concepts that classify data objects, such as *ManufacturedObject* or *PermissionForm*;
- R_D a set of roles that define relations between data objects, such as *dependsOn* or *broaderThan*; and
- X_D a set of axioms and assertions that define relationships between concepts, between roles, and between individuals and concepts or roles, such as *ManufacturedObject(Nail)*.

Example 4.2.36 (Semantic Process Model). *Let* S be the structural process model from example 4.2.34. Let $\mathcal{O}_S = (C_S, R_S, F \cup T, X_S)$ be an ontology with

$$\begin{aligned}
C_S &= \{Process, Step, Gateway, Subprocess\}, \\
R_S &= \{produces, consumes, hasNarrower, name(s), name(p)\}, \text{ and} \\
X_S &= \{Gateway \sqsubseteq Step, \\
&\quad Subprocess \sqsubseteq Step, \\
&\quad Subprocess \sqsubseteq Process\} \cup \\
&\quad \{Process(f_0), Step(f_{start}), Step(f_{fill \text{ in application}}), \\
&\quad Gateway(f_{requires \text{ substitute?}}), \dots \\
&\quad Subprocess(f_{check \text{ by HR department}, \dots})\} \cup \\
&\quad \{produces(f_{choose \text{ substitute}}, \text{Signature of substitute}), \\
&\quad consumes(f_{check \text{ by direct superior}}, \text{Signature of substitute}), \\
&\quad produces(f_{check \text{ by direct superior}}, \text{Signature of direct superior}), \\
&\quad consumes(f_{check \text{ by HR department}}, \text{Signature of direct superior}), \\
&\quad produces(f_{check \text{ by HR department}}, \text{Stamp of HR}), \\
&\quad consumes(f_{enter \text{ data into database}}, \text{Stamp of HR})\} \cup \\
&\quad assertions(s) \cup assertions(p).
\end{aligned}$$

Let $\mathcal{O}_T = (C_T, R_T, T, X_T)$ be an ontology with

$$\begin{aligned}
C_T &= \{Signature\}, \\
R_T &= \emptyset, \text{ and} \\
X_T &= \{Signature(\text{Signature of substitute}), \\
&\quad Signature(\text{Signature of direct superior}), \\
&\quad Signature(\text{Stamp of HR})\}.
\end{aligned}$$

Then $P = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ is a semantic process model.

Note how \mathcal{O}_S also makes use of T , which represents the data objects. This allows us to also model the data flow, i.e., where which data objects are produced and where they are required or consumed.

Lemma 4.2.37 (Semantic Process Models in Description Logics). *For every semantic process model $P = (F, f_0, s, p, D, c, \mathcal{O}_S, \mathcal{O}_D)$ there exists a process ontology \mathcal{O}_P .*

Proof of Lemma 4.2.37. The proof is analogue to the proof of lemma 4.2.22. \square

Proposition 4.2.38 (Interpretation of \mathcal{O}_P). *For every semantic process model $P = (F, f_0, s, p, D, c, \mathcal{O}_S, \mathcal{O}_D)$ and process ontology $\mathcal{O}_P = (C_P, R_P, I_P, X_P)$ constructed from P , there exists an interpretation $I(P) = (\Delta^I, ()^I)$, such that $I(P)$ is a model of \mathcal{O}_P in the sense of definition 3.2.31: $I(P) \models \mathcal{O}_P$. In other words, P can be seen as a model of \mathcal{O}_P .*

Proof of Proposition 4.2.38. The proof is analogue to the proof of proposition 4.2.25. \square

Conclusion

In this chapter, we have explored the nature of documents. We have seen how they are perceived both in the computer science community and beyond, and how they can be characterised. We have then formalised these perceptions, creating metamodels for documents on different layers of abstraction that culminate in a semantic document model, which includes complex structural information as well as content-related semantic data and relationships. We have given a semantics and implementation of semantic models. Finally, we have shown how this model can be transferred to a new domain.

Chapter 5

Processing Digital Documents

Chapter 5 starts with a discussion on how to extract document models from source documents (section 5.1). Afterwards, various inference tasks on document models will be regarded in section 5.2. In section 5.3 we will discuss how to deal with multiple layers of metadata in knowledge modelling.

5.1 Extracting Data Models

In this section, we will outline how to extract a semantic document model (cf. definition 4.2.14 on page 82) from a digital document. We will provide a mapping from a document, represented as a base document model (cf. definition 4.1.21 on page 4.1.21), onto a semantic document model, with a technical representation in description logics (cf. proposition 4.2.25 on page 86). In section 4.1, we have seen that a document can be represented as a connected base document model. Implementation details can be found in chapter 8.

Recall that a base document model B is a tuple (M, F, c, s, f) , where M is a set of media objects (with a subset T of text fragments), F is a set of formatting options, c is a function that assigns to every text fragment its actual content, s is a successor relation on M , and f is a relation that assigns formatting options to text fragments. A base document model is connected, if there is a starting object from which every other object can be reached.

Also recall that a semantic document model D is a tuple $(F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$, where F is a set of document fragments, f_0 is a starting fragment, s is a successor relation on F , p is a has-part relation on F , T is a set of terms, c is a relation that assigns terms to fragments, $\mathcal{O}_S = (C_S, R_S, F, X_S)$ is a structural ontology that provides the terminology for modelling the structure of the document, and $\mathcal{O}_T = (C_T, R_T, T, X_T)$ is a terminological ontology that provides the terminology for modelling the content of the document.

For the remainder of this chapter, we will equate a semantic document model D with its instantiation \mathcal{O}_D as derived by lemma 4.2.22. In particular, we will not differentiate between the description logic representation of D and the interpretation of this representation. As shown in proposition 4.2.25, D can always be regarded as a model of \mathcal{O}_D . Therefore, this simplification does not detract from the validity of the methods described here, but it will significantly enhance readability. As a further simplification, we will use the `hasNarrower` role on C_S as if it were part of any \mathcal{O}_D .

For example, for the role `p` and two document fragments f_0 and f_1 from D , we write a role assertion as `p(f0, f1)` instead of `name(p)(name(f0), name(f1))`. Vice versa, we write $(f_0, f_1) \in p$ instead of $(name(f_0)^I, name(f_1)^I) \in name(p)^I$ as the semantics of this role assertion.

For the mapping of a base document model onto a semantic document model, we will use *background knowledge*. Part of this knowledge are the structural and terminological ontologies \mathcal{O}_S and \mathcal{O}_T that belong to the semantic model. Other background knowledge is needed for identifying relevant content and properties of the source document, and how to transfer them to the target model.

This knowledge includes domain and language dependent keywords, such as “definition” or “example”, and what they imply, such as document fragments that are definitions or examples. It also includes document specific knowledge about formatting options, such as “italic” or “underlined”, and what they imply, such as document fragments that are paragraph headlines or hyperlinks. For example, a line of text starting with the keyword “definition” and formatted in “italics” could be the title of a paragraph that contains a definition.

The term “keyword” is not strictly accurate, because a single keyword can consist of multiple words, such as “for example”, or might even encompass place holders, such as “figure (*number*)”.

Different keywords can be semantically related to each other. In particular, different keywords can imply the same structure or semantics in the document model. These keywords are *equivalent* in the sense that they have the same meaning *with regard to the document structure*. This does not necessarily imply that they are synonyms. For example, the keywords “figure” and “illustration” can be equivalent, as can be “definition” and the abbreviation “def.”.

Another, albeit more rare form of semantic relation between keywords is specialisation. One keyword may imply a broader set of semantics than another. For example, the keyword “type signature” may imply the description of the input *or* the output of a method in a programming language, while the more specialised keywords “input” and “output” only imply one of them.

Given the properties of keywords discussed above, we will represent keywords and their relationships as a taxonomy (cf. definition 3.2.25) that contains the role `hasNarrower` to model specialisation, its inverse `hasBroader` \doteq `hasNarrower`⁻ to model generalisation, their reflexive versions `hasNarrowerSelf` and `hasBroaderSelf`, and the role `hasEquivalent` to model keyword equivalence. `hasEquivalent` is reflexive and symmetric. We will call a taxonomy of this form and purpose a *keyword taxonomy*.

Example 5.1.1 (Keyword Taxonomy). *Let*

$$\begin{aligned} C_K &= \emptyset, \\ R_K &= \{\text{hasNarrower}, \text{hasBroader}, \text{hasNarrowerSelf}, \text{hasBroaderSelf}, \\ &\quad \text{hasEquivalent}\}, \\ I_K &= \{\text{“chapter”}, \text{“introduction”}, \text{“conclusion”}, \text{“definition”}, \text{“example”}, \\ &\quad \text{“illustration”}\}, \text{ and} \\ X_K &= \{\text{hasNarrower}(\text{“chapter”}, \text{“introduction”}), \\ &\quad \text{hasNarrower}(\text{“chapter”}, \text{“conclusion”})\}. \end{aligned}$$

Then $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ *is a keyword taxonomy.*

The background knowledge also contains information about formatting options and their intended meaning. Formatting options can be grouped together to form *styles*. For example, the options “bold” and “italic” can be combined to form a style. Other styles can include font size or colour, indentation, or spacing. Giving a style a name allows for easier referencing. A collection of *named styles* is often called a *template*.

Definition 5.1.2 (Named Style). *A named style, or sometimes simply style, is a named set of formatting options.*

Let F *be a set of formatting options, and* n *be a literal name. Then* $s = (n, F)$ *is a named style.*

For reasons of simplicity, named styles will often only be referred to by their name, so that n *can be substituted for* s *whenever convenient. No ambiguity arises from doing so.*

Example 5.1.3 (Named Style). *Let*

$$\begin{aligned} F &= \{\text{“italic”}, \text{“bold”}\} \text{ and} \\ n &= \text{bold+italic.} \end{aligned}$$

Then $s = (n, F)$ is a named style, referred to by bold+italic.

Similar to keywords, styles can be semantically related to each other. Two or more styles can be equivalent, i.e., they have the same semantic implications. This does not necessarily mean that their formatting is similar. For example, blue coloured text and underlined text may both signify a reference. One style can also be a specialisation of another. For example, the style ‘italic’ may be used to represent headlines, while the style ‘bold+italic’ is used to represent more specific fist-level headlines. Often, these semantics are indicated by the styles’ names, e.g., ‘Headline’ or ‘Headline 1’.

We will model styles in a taxonomy similar to the keyword taxonomy, which we call *style taxonomy*.

Example 5.1.4 (Style Taxonomy). *Let*

$$\begin{aligned} C_S &= \{\text{Headline}, \text{Reference}\}, \\ R_S &= \{\text{hasNarrower}, \text{hasBroader}, \text{hasNarrowerSelf}, \text{hasBroaderSelf}, \\ &\quad \text{hasEquivalent}\}, \\ I_S &= \{\text{italic}, \text{bold+italic}, \text{underlined}\}, \text{ and} \\ X_S &= \{\text{Headline} \sqcap \text{Reference} \sqsubseteq \perp, \\ &\quad \text{Headline}(\text{italic}), \text{Headline}(\text{bold+italic}), \text{Reference}(\text{underlined}), \\ &\quad \text{hasNarrower}(\text{italic}, \text{bold+italic})\}, \end{aligned}$$

where ‘italic’ is a name that refers to text in italics, ‘bold+italic’ is a name that refers to text in bold and italics, and ‘underlined’ is a name that refers to underlined text.

Then $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$ is a style taxonomy.

Note that instead of the role assertion $\text{hasNarrower}(\text{italic}, \text{bold+italic})$, the same thing could be modelled by introducing a concept Headline_1 , an assertion $\text{Headline}_1(\text{bold+italic})$, and an axiom $\text{Headline}_1 \sqsubseteq \text{Headline}$. While this uses the well-defined semantics of concept specialisation instead of relying on the interpretation of the hasNarrower role, experience has shown that using the role is preferable. This is because of the greater flexibility afforded by the role instantiation on individuals as opposed to rigid concept specialisation. It is, for example, possible to introduce new roles that define other relationships between individuals. However, if used consistently, it is also possible to combine both modelling approaches.

Some document formats, such as office formats or HTML with CSS, already allow for the definition of named styles, which can be used directly in the formatting taxonomy.

Mappings are another form of background knowledge that consists of a named set of two-tuples (see below for additional details). The first component of each tuple, called the **source**, specifies the source for some data, while the second component, called **target**, specifies where that data belongs. A mapping may, for example, contain an entry that specifies that the text content of a specific media object from a base document model should be represented as the role-value of the **title** role, annotated to a specific fragment in a semantic document model.

Definition 5.1.5 (Semantic Document Model Induced by a Connected Base Document Model and Background Knowledge). *Let $B = (M, F, c, s, f)$ be a connected base document model, let $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ be a keyword taxonomy, let $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$ be a style taxonomy, let $\mathcal{O}_S = (C_S, R_S, F, X_S)$ be a structure ontology, and let $\mathcal{O}_T = (C_T, R_T, T, X_T)$ be a terminological ontology. Let the set of concepts C_K from the keyword taxonomy contain at*

least the concepts *DataKeyword*, *ReferenceKeyword*, and *FragmentKeyword* (or some equivalent concepts). Let the set of concepts C_S from the style taxonomy contain at least the concepts *DataStyle*, *ReferenceStyle*, and *FragmentStyle* (or some equivalent concepts). Let \mathcal{M} be a set of mappings that contains at least four mappings called “*DataMapping*”, “*ReferenceMapping*”, “*KeywordFragmentMapping*”, and “*StyleFragmentMapping*”, respectively (or some equivalent mappings). Let \mathcal{K} be a set of background knowledge containing \mathcal{O}_S , \mathcal{O}_T , \mathcal{X}_K , \mathcal{X}_S , and \mathcal{M} .

Then $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ is the semantic document model induced by B and \mathcal{K} , written as $B, \mathcal{K} \Rightarrow D$, iff

1. for every media object in B that induces the existence of a document fragment of some particular types, a corresponding fragment exists in D and this fragment has the appropriate types;
2. for every media object in B that induces a particular data point for a document fragment, there is an appropriate data annotation at the corresponding fragment in D ;
3. for every media object in B that induces a reference from the current document fragment to another, there is a reference between the two appropriate fragments in D ;
4. for every two media objects in B that both induce the existence of a document fragment each, that are in a (possibly indirect) successor relationship s , and that have no other media object that induces the existence of another fragment between them:
iff no type of the fragment induced by the first object is narrower than or is equal to any type of the fragment induced by the second object, then the second fragment is a sub-fragment of the first; and
5. for every two media objects in B that are in a direct successor relationship s : the closest fragments induced by them (see below) are either the same, or they are in a successor relationship, or they are in a has-part relationship.

The first three points simply define the existence of all required fragments, data points, and relationships. The fourth point defines the order of these fragments w.r.t. the has-part relation. The fifth point stipulates that successor relationships in the base document model are not lost in the semantic document model.

A fragment f induced by a media object m is written as $m \Rightarrow f$. The closest fragment induced by m , $m \in M$, is the fragment $f \in F : \exists m_1 \in M : m_1 \Rightarrow f \wedge (m_1 = m \vee ((m_1, m) \in s^+ \wedge \nexists m_2 \in M : m_1 \neq m_2 \neq m \wedge (m_1, m_2), (m_2, m) \in s^+ \wedge \exists f_2 \in F : m_2 \Rightarrow f_2))$. If neither m nor any of its predecessors (if they exist) induces a fragment, then the closest fragment induced by m is defined to be the root fragment of D , f_0 . The closest fragment f induced by m is written as $m \Rightarrow_0 f$. Formally, $B, \mathcal{K} \Rightarrow D \Leftrightarrow$

1. $\forall m \in M : f(m) \in \mathbf{FragmentStyle} \vee c(m) \in \mathbf{FragmentKeyword}$
 \Rightarrow
 $\exists f \in F : \forall (s, t) \in (\text{“KeywordFragmentMapping”} \cup \text{“StyleFragmentMapping”}) : (s \text{ applies to } m \Rightarrow t(f) \in X_S),$
2. $\forall m \in M : f(m) \in \mathbf{DataStyle} \vee c(m) \in \mathbf{DataKeyword}$
 \Rightarrow
 $\exists f \in F : (m \Rightarrow_0 f) \wedge \forall (s, t) \in \text{“DataMapping”} : s \text{ applied to } m \text{ results in } d \Rightarrow t(f, d) \in X_T,$

3. $\forall m \in M : f(m) \in \mathbf{ReferenceStyle} \vee c(m) \in \mathbf{ReferenceKeyword}$
 \Rightarrow
 $\exists f \in F : (m \Rightarrow_0 f) \wedge \forall (s, t) \in \text{“ReferenceMapping”} :$
(s applied to m results in $m_T \in M$)
 \Rightarrow
 $\exists f_T \in F : (m_T \Rightarrow_0 f_T) \wedge t(f, f_T) \in X_S,$
4. $\forall m_1, m_2 \in M, f_1, f_2 \in F :$
 $(m_1 \neq m_2) \wedge (m_1, m_2) \in s^+ \wedge (m_1 \Rightarrow f_1) \wedge (m_2 \Rightarrow f_2) \wedge$
 $\nexists m_3 \in M, f_3 \in F : (m_1, m_3), (m_3, m_2) \in s^+ \wedge m_3 \Rightarrow f_3$
 \Rightarrow
 $\nexists t_1, t_2 \in C_S : t_1(f_1), t_2(f_2) \in X_S \wedge$
 $(\mathbf{hasNarrower}^+(t_2, t_1) \in X_S \vee (t_1 \equiv t_2) \in X_S)$
 \Leftrightarrow
 $(f_1, f_2) \in p, \text{ and}$
5. $\forall m_1, m_2 \in M, f_1, f_2 \in F : (m_1, m_2) \in s \wedge m_1 \Rightarrow_0 f_1 \wedge m_2 \Rightarrow_0 f_2$
 \Rightarrow
 $(f_1 = f_2) \oplus ((f_1, f_2) \in s^+ \oplus (f_1, f_2) \in p).$

\oplus denotes the XOR operator.

Remark 5.1.6. Note that this definition expects a semantic document model to have the same basic order of elements as the underlying base document model. While this is the case for most document types, there are exceptions like the Microsoft Visio format. We will discuss options for dealing with such a situation in chapter 9 and in section 10.2.

Example 5.1.7 (Semantic Document Model Induced by a Connected Base Document Model and Background Knowledge). Let B be the connected base document model from example 4.1.24. Let \mathcal{X}_S be the style taxonomy from example 5.1.4. Let D be the semantic document model from example 4.2.15. Let \mathcal{K} be a set of background knowledge containing \mathcal{X}_S and the ontologies from D .

Then D is the semantic document model induced by B and \mathcal{K} .

A set of *transformation rules* can be used for mapping a connected base document model onto a semantic model. Using an appropriate set of transformation rules, it is possible to obtain the semantic document model induced by a base document model and a set of background knowledge.

Definition 5.1.8 (Transformation Rule). A *transformation rule* from a base document model $B = (M_B, F_B, c_B, s_B, f_B)$ onto a semantic document model $D = (F_D, f_0, s_D, p_D, T_D, c_D, \mathcal{O}_S, \mathcal{O}_T)$ consists of a *premise* P and a *conclusion* C , written as

$$P \hookrightarrow C.$$

The base document model B may also have associated metadata $A_B = (L_A, V_A, d_A)$ (cf. definition 4.1.26) that can be used in the transformation rule.

A transformation rule may also make use of a set of background knowledge \mathcal{K} , which may for example contain a keyword taxonomy $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ and a style taxonomy $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$.

Definition 5.1.9 (Premise of a Transformation Rule). *The premise of a transformation rule consists of one or more logically connected conditions, using the usual logical operators such as ‘ \neg ’, ‘ \wedge ’, or ‘ \vee ’ with their usual semantics. Each condition tests for a relationship between pairs of entities from the following base sets and elements, for all models I_D of D :*

- C the set of constants,
- V the set of variables,
- f : $M_B \rightarrow I_S$, a function returning the formatting style of a media object $m \in M_B$ from the base document model,
- c : $M_B \rightarrow \text{text}$, a function returning the text content of a media object $m \in M_B$ as a list of literals,
- md : $M_B \times L_A \rightarrow 2^{V_D}$, a function returning for pairs of media objects and labels the set of associated metadata values as a list of literals, and
- md_B : $L_A \rightarrow 2^{V_D}$, a function returning for a label the set of metadata values associated with the base document model as a list of literals.

Additionally, all individuals, concepts, roles, axioms and assertions from the background knowledge or from the ontologies associated with the semantic document model may be used. In particular, the function hn serves as an abbreviation for the interpretation of the `hasNarrower` role, and hn^+ serves as an abbreviation for the transitive closure of hn . Analogously, there exist abbreviations for the roles `hasBroader`, `hasNarrowerSelf`, `hasBroaderSelf`, and `hasEquivalent`: hb , hb^+ , hns , hns^+ , \dots

On these entities, several relations can be used. Based on their respective value ranges, we divide these relations into five groups, with $r_> \in \{<, \leq, \geq, >\}$, $r_ = \{=\}$, $r_ \in \{\in\}$, $r_ \subseteq \in \{\subseteq\}$, and $r_ \subseteq_1 \in \{\subseteq_1\}$, where

- $r_>$ is restricted to constants and variables of numerical type,
- $r_ =$ is restricted to pairs of equal type,
- $r_ \in$ is restricted to pairs of elements and sets or lists of the same type,
- $r_ \subseteq$ is restricted to pairs of sets or lists of the same type, and
- $r_ \subseteq_1$ is restricted to pairs of power sets and sets or lists of the same type.

All relations are written in infix notation for simplicity, e.g., with m a media object from the base document model, we write

$$f(m) \text{ ‘} \in \text{’ Reference.}$$

The relations ‘ $<$ ’, ‘ \leq ’, ‘ \geq ’, ‘ $>$ ’, ‘ $=$ ’, ‘ \in ’ and ‘ \subseteq ’ can be interpreted with their usual semantics. In addition, ‘ \in ’ can be used to test for membership in a list and to test for a substring relationship, and ‘ \subseteq ’ can be used to test if all elements of one set or list are contained in another set or list. ‘ \subseteq_1 ’ can be used as an infix operator for testing if the intersection of two sets or list is not empty, i.e., if at least one element of one set is a contained in another set:

$$S \text{ ‘} \subseteq_1 \text{’ } T \Leftrightarrow S \cap T \neq \emptyset.$$

The usual operations on Boolean, numerical, literal, list and set values are also allowed, such as negation, addition, concatenation, and insertion.

Intuitively, a premise allows for the specification of properties that objects from a base document model must match. These specifications may make use of variables, constants, the contents of the base document model, and the available knowledge bases. Several functions and relations are defined that allow for aggregation and comparison of values.

Example 5.1.10 (Premise of a Transformation Rule). *Let*

$$f(m) \text{ ‘} \in \text{’ } \textit{Headline}$$

$$hns^+(\text{"chapter"}) \text{ '}\subseteq_1\text{' } c(m)$$

be two conditions. Then

$$(f(m) \text{ '}\in\text{' } \mathbf{Headline}) \wedge (hns^+(\text{"chapter"}) \text{ '}\subseteq_1\text{' } c(m))$$

is the premise of a transformation rule.

Definition 5.1.11 (Conclusion of a Transformation Rule). *The conclusion of a transformation rule consists of a list of operations. Each operation can either*

1. assign to or modify the value of a variable,
2. create a new document fragment in the semantic document model,
3. define a relationship s or p between two document fragments,
4. add a new term to the semantic document model,
5. define a relationship c between a document fragment and a term,
6. add a new concept or role assertion to the structural ontology \mathcal{O}_S ,
7. add a new concept or role assertion to the terminological ontology \mathcal{O}_T , or
8. evaluate a condition or some other Boolean expression to a truth value and
 - (a) call a list of operations based on whether the value is true or false (*if-then-else*), or
 - (b) repeatedly call a list of operations as long as the value is true (*while-do*). This includes equivalent concepts like *do-while* and *for* loops.

Example 5.1.12 (Conclusion of a Transformation Rule). *Let*

```
DocumentFragment  $\mathcal{E}_L.f$  = new DocumentFragment() and
assert: Chapter( $\mathcal{E}_L.f$ )
```

be two operations specified in pseudo-code. Then

```
(
  DocumentFragment  $\mathcal{E}_L.f$  = new DocumentFragment(),
  assert: Chapter( $\mathcal{E}_L.f$ )
)
```

is the conclusion of a transformation rule.

As example 5.1.12 shows, conclusions may make use of variables. The values held by such variables are determined by their *environment*.

Definition 5.1.13 (Environment). *The environment \mathcal{E} of a variable maps the name of the variable onto the variable's value. Each environment may contain a finite number of variables. No two environments may contain the same variable.*

A variable v contained in an environment \mathcal{E} will be written as $\mathcal{E}.v$.

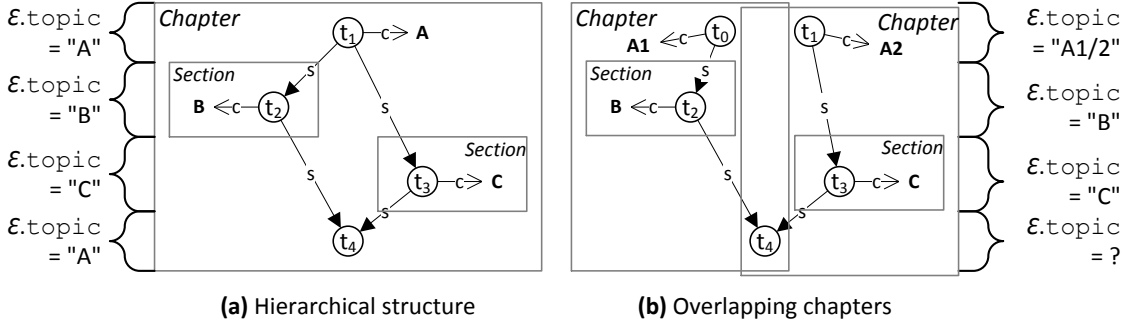


Figure 5.1: Values for a variable `topic` in an environment \mathcal{E} at different positions in a document

For use in transformation rules, we define two environments that satisfy different requirements. Some variables change their value based on the location in the document of the current media object. If the situation allows for it, they may even revert back to earlier values, where “earlier” refers to the order in which an algorithm processes the media objects.

For example, consider a variable `topic` that holds the topic of the current document position. As illustrated in figure 5.1 (a), it is assigned a value for a media object t_1 that indicates a chapter. For other media objects t_2, t_3 that indicate sections, the variable is assigned new values, namely the topic of the respective sections. Finally, when a media object t_4 indicates the end of the sections, the variable reverts to its earlier value.

For variables that change their value depending on the position in the document structure, i.e., chapter or section, it is always possible to uniquely determine to which value it should revert, provided that the structure is strictly hierarchical. This is possible because in a strict hierarchy, there is always at most a single parent element. The variable can revert to the value it held at this parent element. Figure 5.1 (b) shows how a non-hierarchical structure with overlapping chapters makes it impossible to uniquely identify a value for the `topic` variable in t_4 , at least without resorting to some arbiter function that picks one of the possible values “A1” or “A2”.

Other variables change their value irrespectively of the location in the document, such as the list of parsed files, or which media objects have already been processed by an algorithm. These variables also do not revert to former values.

Definition 5.1.14 (Local Environment). A *local environment* $\mathcal{E}_L(m)$ for a media object m contains variables whose value depends on the current position in the document, represented by m . A variable v contained in the local environment will be written as $\mathcal{E}_L.v$.

Example 5.1.15 (Local Environment). In figure 5.1 (a), for all media objects that belong directly to the chapter, the value of $\mathcal{E}_L.topic$ is “A”. Conversely, for all media objects that belong directly to one of the sections, the value is either “B” or “C”.

Definition 5.1.16 (Global Environment). A *global environment* \mathcal{E}_G contains variables whose value is independent of the current position in the document. A variable v contained in the global environment will be written as $\mathcal{E}_G.v$.

Example 5.1.17 (Global Environment). In figure 5.1 (a), if an algorithm processes t_4 for the first time, as the successor of t_2 , the value of $\mathcal{E}_G.processed$ is $\{t_1, t_2\}$. If an algorithm processes t_4 for the second time, as the successor of t_3 , the value of $\mathcal{E}_G.processed$ is $\{t_1, t_2, t_3, t_4\}$.

Example 5.1.18 (Transformation Rule). *For the base document model from example 4.1.24 and the semantic document model from example 4.2.15, the following listing contains a transformation rule in pseudo code that creates a new document fragment for each chapter and lists the relevant terminology for the fragment.*

```

1 (
2   f(m) '∈' Headline ∧
3   hns+("chapter") '⊆1' c(m)
4 )
5 ⇔
6 (
7   DocumentFragment  $\mathcal{E}_L.f = \text{new DocumentFragment}()$ ,
8   assert: Chapter( $\mathcal{E}_L.f$ ),
9   for each (term '∈' c(m))
10    if (term '∈'  $T_D$ )
11      assert: c( $\mathcal{E}_L.f$ , term)
12 )

```

Lines 2 and 3 contain the premise of the rule, stating that the conclusion should only be applied to media objects that have a formatting style that is part of the **Headline** concept (line 2) and that contain a keyword or one of its specialisations which indicate a headline (line 3). In particular, line 3 checks if one of the sets of chapter-keywords is contained in the text of the media object (different word forms are disregarded in this example). $\text{hns}^+(\text{"chapter"})$ contains the keyword "chapter" and all its specialisations, namely {"chapter", "introduction", "conclusion"}.

In line 7, a new document fragment is created and assigned to a variable f . Then this fragment is asserted to belong to the concept **Chapter** in line 8.

Finally, lines 9 through 11 check for each term in the text content of m if this term is a relevant domain term from the terminological ontology \mathcal{O}_T . If the term is a domain term, then it is asserted to be in a c -relationship with the new fragment. Recall that the role c models which document fragments deal with which terms (cf. definition 4.2.1).

Definition 5.1.19 (Application of a Transformation Rule). *A transformation rule $t = P \leftrightarrow C$ can be applied to a media object $m \in M_B$ from a connected base document model $B = (M_B, F_B, c_B, s_B, f_B)$ and to a semantic document model D to return a new semantic document model D' , written as*

$$m, D \xrightarrow{t} D'.$$

It returns an altered version D' of D that is usually used to replace D in further processing steps.

A condition $e_1 \ r \ e_2$ of the premise is fulfilled for m , iff the entities e_1 and e_2 are in an r -relationship.

The premise P is fulfilled for m , iff the evaluation of the Boolean combination of its conditions yields true.

The operations of the conclusion C are applied to D , resulting in an altered version D' , if the premise P is fulfilled for m . If the premise is not fulfilled, then $D = D'$.

Example 5.1.20 (Application of a Transformation Rule). *Let $B = (M_B, F_B, c_B, s_B, f_B)$ be the connected base document model from example 4.1.30 (with the tuple-entries renamed for clarity). Let $D = (F_D, f_0, s_D, p_D, T_D, c_D, \mathcal{O}_S, \mathcal{O}_T)$ be the semantic document model from example 4.2.15 (again with the tuple-entries renamed for clarity), with $F_D = \{f_0\}$ and $s_D = p_D = c_D = \emptyset$. Let t be the transformation rule from example 5.1.18.*

Then the application $t_1, D \xrightarrow{t} D'$ of t to the media object $t_1 \in M_B$ from B and to D results in a new semantic document model $D' = (F_D \cup \{f_1\}, f_0, s_D, p_D, T_D, c_D, \mathcal{O}_S, \mathcal{O}_T)$, with a newly created document fragment f_1 .

The media object t_1 has a formatting style {bold, italic} that corresponds to the named style ‘bold+italic’. Since ‘bold+italic’ \in *Headline*, the first operand of the conjunction that forms t ’s premise yields true.

The function hns^+ applied to the keyword “chapter”) returns the set $N = \{\text{“chapter”}, \text{“introduction”}, \text{“conclusion”}\}$. The function $c(t_1)$ returns the list $L = (\text{“introduction”})$. Since one element of N , namely “introduction”, is equivalent to an element in L , the second operand of t ’s premise also yields true: $N \text{ ‘}\subseteq_1\text{’ } L$.

With the premise of t fulfilled for t_1 , the conclusion is applied to D , creating a new document fragment. Since the text of t_1 does not contain any known domain terms, no additional data is added for the new fragment.

Definition 5.1.21 (Mapping of a Base Document Model onto a Semantic Document Model). A connected base document model $B = (M_B, F_B, c_B, s_B, f_B)$ is mapped onto a semantic document model $D = (F_D, f_0, s_D, p_D, T_D, c_D, \mathcal{O}_S, \mathcal{O}_T)$ using a set of background knowledge \mathcal{K} and a finite list of transformation rules $T = (t_1, t_2, \dots, t_n)$ by applying every $t \in T$ to every $m \in M_B$ and D_0 , each time replacing D_0 with the resulting D . $D_0 = (\{f_0\}, f_0, \emptyset, \emptyset, T_D, \emptyset, \mathcal{O}_S, \mathcal{O}_T)$ is an empty semantic document model with the same ontologies as D . Both the transformation rules and the media objects must be selected in a well-defined order. This mapping is written as

$$B, D_0 \xrightarrow{T, \mathcal{K}} D.$$

\mathcal{K} may, for example, contain a keyword taxonomy $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ and a style taxonomy $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$.

Example 5.1.22 (Mapping of Base Document Model onto Semantic Document Model). Let B be the connected base document model and D be the semantic document model from example 5.1.20. Let \mathcal{X}_K be the keyword taxonomy from example 5.1.1, and let \mathcal{X}_S be the style taxonomy from example 5.1.4. Let $T = \{t\}$, with t the transformation rule from example 5.1.18. Then $B, D_0 \xrightarrow{T, \{\mathcal{X}_K, \mathcal{X}_S\}} D$ results in $D = (F_D, f_0, s_D, p_D, T_D, c_D, \mathcal{O}_S, \mathcal{O}_T)$, with $F_D = \{f_0, f_1, f_2, f_3, f_4, f_5, f_{21}, f_{31}, f_{41}, f_{42}\}$.

More complex real-world examples will be discussed in chapter 10.

Application

For actual, real-world documents that are represented as base document models, sets of extraction rules can be defined. However, when specifying rules, the manner of conflict resolution must be defined beforehand. In particular, it must be clear in which order rules are processed that all match the same media object m . It must also be well-defined in which order media objects are checked for any matching rules.

In this section, we will assume that media objects are checked in the order defined by the successor relation s . If a media object has more than one successor, then these successors are processed in their correct order: the successor path started by the first successor is followed until all media objects on this path have been checked. Then, backtracking to the media object with multiple successors, the next successor is picked, and so forth, until all media objects in the base document model have been checked. A successor path ends if it re-visits a media object that is already on the path or that has already been processed, to prevent infinite loops.

The rules make use of a stack in the global environment that holds the current document fragment and all its parent fragments (see below). Whenever backtracking to a forking media object occurs, this stack is reset to the state it was in before the first path was chosen. If a media

object occurs on multiple reading paths, it is also processed multiple times. Since asserting facts that are already known does not change the fact base, and since creating new fragments for the same media objects leads to the creating of the same fragment over and over again, processing media objects multiple times has no ill side effects.

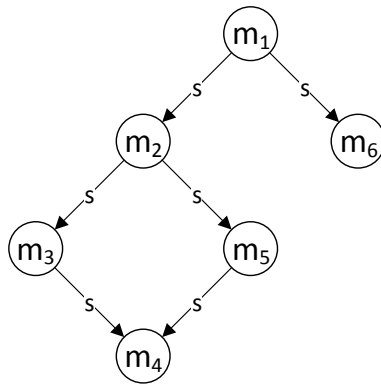


Figure 5.2: Simple base document model to illustrate the processing order of media objects

This management of the control flow is illustrated in figure 5.2. It shows a base document model of six media objects that are processed in the order $m_1, m_2, m_3, m_4, m_5, m_6$. If each of these objects induces a fragment of the same type, e.g., “chapter”, then this model induces the semantic document model $0[1, 2, 3, 4, 5, 6]$. If, however, m_3 induces a fragment of type “section” and both m_4 and m_5 induce fragments of type “paragraph” (with the intuitive hierarchy that a section is **narrower** than a chapter and that a paragraph is **narrower** than a section), then the model induces the semantic document model $0[1, 2[3[4], 5], 6]$. In particular, the paragraph induced by m_4 is a sub-fragment of the section induced by m_3 , and the paragraph induced by m_5 is a sub-fragment of the chapter induced by m_2 .

This not only shows that backtracking to a branching media object is necessary, as seen when backtracking from m_4 to m_2 to process m_5 and when backtracking from m_5 to m_1 to process m_6 . It also shows that resetting the stack when backtracking is necessary (see below). Had the stack not been reset after backtracking from m_4 to m_2 , then it would have held the paragraph fragment induced by m_4 and the section fragment induced by m_3 . When processing m_5 next, the paragraph fragment would have been removed from the stack, but the section fragment would have been used as the parent fragment for the new paragraph induced by m_5 , resulting in the semantic document model $0[1, 2[3[4, 5]], 6]$. If the stack is properly reset, it holds the chapter fragment induced by m_2 , and the new paragraph is inserted in its correct place.

We will also assume that rules will be processed in reverse specification order, i.e., the rule that was specified last will be matched first against a media object. This behaviour is compatible with that of common rule engines like JBoss Drools.

For the initialisation, we will assume that an almost empty semantic document model exists that only contains a single root fragment. A stack data structure is used in some of the rules in the sequel, which initially contains only this root fragment. The root fragment has a type for which there is no broader type in the background knowledge. This type is usually called “Document” or something similar, and there is no role assertion $\text{hasNarrower}(t, \text{“Document”})$ for any type t .

First, we will regard a set of rules defined for a specific document. Then, we will attempt to derive more generic rules from them. A thorough discussion of a concrete implementation of several sets of rules can be found in section 9.1.

Example 5.1.23 (Transformation Rules for Specific Document). *The following seven rules T can be used to extract a semantic document model from a document represented as a base document model.*

The first rule, starting in line 1, matches media objects whose formatting is contained in the style concept `TitleStyle` from \mathcal{X}_S . It then creates a role assertion that puts the current document fragment from the semantic document model, taken from the stack in the global context \mathcal{E}_G , in a `title`-relationship with the text content of the media object. For a more detailed discussion on how the document model is created, and why a stack is used to manage the order of fragments, we refer the reader to section 9.1.

The second rule (line 5ff.) is similar to the first, except that it processes terms, not titles.

*The third rule that starts in line 9 creates **reference** assertions between the current fragment and another fragment pointed to by a reference in the source document. The `$\mathcal{E}_G.getFragment()$` function backtracks from the current media objects until a preceding media object is found that induces a fragment, i.e., the function finds the closest fragment induced by the media object.*

The final four rules, starting in line 13, each match a particular kind of fragment type. The first one, for example, matches fragments that are identified by their formatting or keywords as chapters. The second clause of the rule’s premise checks if one of the terms in a media object is contained in the reflexive and transitive closure of the `hasNarrower`-role on “chapter”. A new document fragment is created in the conclusion of the rule, which is then asserted to be a `Chapter` instance and finally inserted into the document model in its proper place. This place is determined by the relative “narrowness” of the fragment’s type versus the type of the fragments on the stack. While the intersection of the reflexive and transitive closure of the `hasNarrower`-role on the types of the new fragments ($hns^+(\mathcal{E}_L.f.types)$) with the types of the fragment on top of the stack ($\mathcal{E}_G.stack.top.types$) is not empty, the top stack element is removed. Since the stack is initialised with a fragment of the broadest type, this process stops at the root fragment at the latest. After the new fragment has been put into a `p`-relationship with the top stack fragment, it is itself pushed onto the stack.

The next three rules have a similar behaviour for definitions, examples and illustrations that are recognised by a combination of their formatting style and keywords.

```

1 ( f(m) '∈' TitleStyle )
2   ⇨
3 ( assert: title( $\mathcal{E}_G.stack.top$ , c(m)) )
4
5 ( f(m) '∈' TermStyle )
6   ⇨
7 ( assert: term( $\mathcal{E}_G.stack.top$ , c(m)) )
8
9 ( f(m) '∈' ReferenceStyle )
10  ⇨
11 ( assert: reference( $\mathcal{E}_G.stack.top$ ,  $\mathcal{E}_G.getFragment(c(m))$ ) )
12
13 ( (f(m) '∈' ChapterStyle) ∨ (c(m) '⊆1' hns+("chapter")) )
14   ⇨
15 (
16   DocumentFragment  $\mathcal{E}_L.f$  = new DocumentFragment(m),
17   assert: Chapter( $\mathcal{E}_L.f$ ),
18   while (hns+( $\mathcal{E}_L.f.types$ ) '⊆1'  $\mathcal{E}_G.stack.top.types$ )
19      $\mathcal{E}_G.stack.pop()$ ,
20   assert: hasPart( $\mathcal{E}_G.stack.top$ ,  $\mathcal{E}_L.f$ ),

```

```

21    $\mathcal{E}_G.stack.push(\mathcal{E}_L.f)$ 
22 )
23
24 ( ( $f(m) \in \text{ParagraphStyle}$ )  $\vee$  ( $c(m) \subseteq_1 \text{hns}^+(\text{"definition"})$ ) )
25  $\leftrightarrow$ 
26 (
27    $\text{DocumentFragment } \mathcal{E}_L.f = \text{new DocumentFragment}(m),$ 
28    $\text{assert: Paragraph}(\mathcal{E}_L.f),$ 
29    $\text{assert: Definition}(\mathcal{E}_L.f),$ 
30    $\text{while } (\text{hns}^+(\mathcal{E}_L.f.types) \subseteq_1 \mathcal{E}_G.stack.top.types)$ 
31      $\mathcal{E}_G.stack.pop(),$ 
32    $\text{assert: hasPart}(\mathcal{E}_G.stack.top, \mathcal{E}_L.f),$ 
33    $\mathcal{E}_G.stack.push(\mathcal{E}_L.f)$ 
34 )
35
36 ( ( $f(m) \in \text{ParagraphStyle}$ )  $\vee$  ( $c(m) \subseteq_1 \text{hns}^+(\text{"example"})$ ) )
37  $\leftrightarrow$ 
38 (
39    $\text{DocumentFragment } \mathcal{E}_L.f = \text{new DocumentFragment}(m),$ 
40    $\text{assert: Paragraph}(\mathcal{E}_L.f),$ 
41    $\text{assert: Example}(\mathcal{E}_L.f),$ 
42    $\text{while } (\text{hns}^+(\mathcal{E}_L.f.types) \subseteq_1 \mathcal{E}_G.stack.top.types)$ 
43      $\mathcal{E}_G.stack.pop(),$ 
44    $\text{assert: hasPart}(\mathcal{E}_G.stack.top, \mathcal{E}_L.f),$ 
45    $\mathcal{E}_G.stack.push(\mathcal{E}_L.f)$ 
46 )
47
48 ( ( $f(m) \in \text{ParagraphStyle}$ )  $\vee$  ( $c(m) \subseteq_1 \text{hns}^+(\text{"illustration"})$ ) )
49  $\leftrightarrow$ 
50 (
51    $\text{DocumentFragment } \mathcal{E}_L.f = \text{new DocumentFragment}(m),$ 
52    $\text{assert: Paragraph}(\mathcal{E}_L.f),$ 
53    $\text{assert: Illustration}(\mathcal{E}_L.f),$ 
54    $\text{while } (\text{hns}^+(\mathcal{E}_L.f.types) \subseteq_1 \mathcal{E}_G.stack.top.types)$ 
55      $\mathcal{E}_G.stack.pop(),$ 
56    $\text{assert: hasPart}(\mathcal{E}_G.stack.top, \mathcal{E}_L.f),$ 
57    $\mathcal{E}_G.stack.push(\mathcal{E}_L.f)$ 
58 )

```

Let $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ be the keyword taxonomy from example 5.1.1.

Let $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$ be a style taxonomy, with

$$\begin{aligned}
C_S &= \{\text{TitleStyle}, \text{TermStyle}, \text{ReferenceStyle}, \\
&\quad \text{ChapterStyle}, \text{ParagraphStyle}\}, \\
R_S &= \{\text{hasNarrower}, \text{hasBroader}, \text{hasNarrowerSelf}, \text{hasBroaderSelf}, \\
&\quad \text{hasEquivalent}\}, \\
I_S &= \{\text{italic}, \text{bold+italic}, \text{underlined}\}, \text{ and} \\
X_S &= \{\text{TitleStyle}(\text{bold+italic}), \text{TermStyle}(\text{italic}), \\
&\quad \text{ReferenceStyle}(\text{underlined}), \\
&\quad \text{ChapterStyle}(\text{bold+italic}), \text{ParagraphStyle}(\text{italic})\}.
\end{aligned}$$

Then the base document model B from example 4.1.24 is mapped onto the semantic document model D from example 4.2.15 using the rules T and the taxonomies \mathcal{X}_K and \mathcal{X}_S

$$B, D \xrightarrow{T, \{\mathcal{X}_K, \mathcal{X}_S\}} D'.$$

If we push more information from the rules into the background knowledge to separate it from the extraction logic, we can specify more generic rules.

These rules use a special syntax for referring to arbitrary individual, concept, or role names: $\{name\}$ refers to the individual, concept, or role indicated by the place holder $name$. This allows for parameterised assertions. For example,

```
1 String name = "Chapter",
2 assert: {name}(fragment)
```

is equivalent to

```
1 assert: Chapter(fragment)
```

Description 5.1.24 (Transformation Rules for Generic Documents). *The following transformation rules T_G can be applied to generic base document models. The specifics of what media objects to map onto what document fragments are moved into the background knowledge.*

The first rule in line 1ff. covers the first two rules from example 5.1.23. It matches media objects of a specific style or containing specific keywords, and annotates their content to the current document fragment. The role used for this annotation is determined by the target specified in the “DataMapping” background knowledge ($map.target$), and what part of the content is used is determined by the source specified in the mapping ($map.source$). The $eval()$ function evaluates this source specification in the context of the current media object. For example, for XML-based file formats, the source can be specified in XPath, and the $eval()$ function evaluates the XPath expression against the current XML DOM node.

Line 8ff. contains a rule that processes references, similar to the one from example 5.1.23, but using background knowledge in the same manner as the previous rule. The $\mathcal{E}_G.getFragment()$ function remains the same as above.

The third rule (line 16ff.) covers the final four rules from example 5.1.23. It determines the types of a new document fragment dynamically from the background knowledge by checking if a mapping can be applied to a media object (line 21), and then asserting the type indicated by this mapping for the fragment (line 22). The fragment is then inserted into the document model as before.

```
1 ( (f(m) '∈' DataStyle) ∨ (c(m) '⊆1' hns+(DataKeyword)) )
2   ↪
3 (
4   for each (map '∈' DataMapping)
5     assert: {map.target}( $\mathcal{E}_G.stack.top$ , eval(map.source, m))
6 )
7
8 ( (f(m) '∈' ReferenceStyle) ∨ (c(m) '⊆1' hns+(ReferenceKeyword)) )
9   ↪
10 (
11   for each (map '∈' ReferenceMapping)
12     assert: {map.target}( $\mathcal{E}_G.stack.top$ ,
13        $\mathcal{E}_G.getFragment(eval(map.source, m))$ )
14 )
15
16 ( (f(m) '∈' FragmentStyle) ∨ (c(m) '⊆1' hns+(FragmentKeyword)) )
17   ↪
18 (
19   DocumentFragment  $\mathcal{E}_L.f$  = new DocumentFragment(m),
20   for each (map '∈' StyleFragmentMapping ∪ KeywordFragmentMapping)
21     if (hns+{map.source} '=' f(m) ∨ hns+{map.source} '⊆1' c(m))
22     assert: {map.target}( $\mathcal{E}_L.f$ ),
```



```

23   while (hbs+( $\mathcal{E}_G.stack.top.types$ ) ' $\subseteq_1$ '  $\mathcal{E}_L.f.types$ )
24      $\mathcal{E}_G.stack.pop()$ ,
25   assert: hasPart( $\mathcal{E}_G.stack.top$ ,  $\mathcal{E}_L.f$ ),
26    $\mathcal{E}_G.stack.push(\mathcal{E}_L.f)$ 
27 )

```

We refer the reader to section 9.1 for a detailed account of a concrete implementation of these rules, as well as a walkthrough application of the rules to an example document.

This approach is based on the assumption that there exists a well-defined order between structural types, i.e., there exists a relation *hasNarrower* on types that is

1. total,
2. transitive,
3. irreflexive,
4. asymmetric,
5. antisymmetric, and
6. a strict total ordering relation.

Totality is required so that the order of any two types can be determined, i.e., so that it can be decided for any pair of fragments if one is a sub-fragment of the other. Transitivity is required so that the order remains consistent even if a level is skipped, i.e., a chapter should always be broader than a paragraph, whether or not there is a section fragment in between. Irreflexivity is required to ensure that the relation remains acyclic.

Asymmetry follows from transitivity and irreflexivity, and antisymmetry follows from asymmetry. Thus, *hasNarrower* is a strict and total ordering relation.

In practise, however, none of these requirements *must* be met if it can be otherwise ensured that this does not lead to any of the problems mentioned. For instance, if it is “known” that specific types will never be in conflict then they do not have to be in an ordering relationship, thus violating the totality requirement. As an example, table cells, which must always be part of a table, will usually not come into conflict with non-table elements like sections.

If the source base document model is implemented in a language that has both opening and closing tags like XML, the requirements on the *hasNarrower* relation can also be relaxed, provided that an appropriate rule that matches the closing tags is added to the rule set. In this case, responsibility for keeping track of the structural hierarchy rests not solely with the *hasNarrower* relation, but also with this new rule. Imagine, for example, two successive media objects that each induce a new fragment. Yet there is no information about the relative order of the two fragments’ types, i.e., it cannot be determined if the second fragment should be a sub-fragment of the first or not using only the *hasNarrower* relation. However, the new rule for closing tags can help determine if the opening tag of the second media object is located *within* the tags of the first media object, indicating that the second fragment should be a sub-fragment of the first. If the opening tag of the second media object only occurs after the closing tag of the first media object, then the second fragment is not a sub-fragment of the first.

Special cases, for example if an order relationship between two particular types changes in specific contexts, must be handled by defining new rules for these cases.

The knowledge about a document’s structural hierarchy is also part of a formal semantic document model in the form of the **hasNarrower** role.

Example 5.1.25 (Transformation Rules for Generic Documents). *The transformation rules T_G from description 5.1.24 can also be applied to the base document model from example 4.1.24. Let \mathcal{M} be a set containing the following mappings, i.e., named sets of tuples:*

- ▶ “DataMapping”:
 $\{(if\ f(m) \in \{\text{bold+italic}\}\ \text{then}\ c(m),\ \text{title}),\ (if\ f(m) \in \{\text{italic}\}\ \text{then}\ c(m),\ \text{term})\}$,
- ▶ “ReferenceMapping”: $\{(c(m),\ \text{reference})\}$, and
- ▶ “StyleFragmentMapping”: $\{(\text{“Chapter”},\ \text{Chapter}),\ (\text{“Definition”},\ \text{Paragraph}),\ (\text{“Definition”},\ \text{Definition}),\ (\text{“Example”},\ \text{Paragraph}),\ (\text{“Example”},\ \text{Example}),\ (\text{“Illustration”},\ \text{Paragraph}),\ (\text{“Illustration”},\ \text{Illustration})\}$
- ▶ “KeywordFragmentMapping”: $\{(\text{“chapter”},\ \text{Chapter}),\ (\text{“definition”},\ \text{Definition}),\ (\text{“example”},\ \text{Example}),\ (\text{“illustration”},\ \text{Illustration})\}$

Let $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ be a keyword taxonomy, with

$$\begin{aligned} C_K &= \{\text{DataKeyword}, \text{ReferenceKeyword}, \text{FragmentKeyword}\}, \\ R_K &= \{\text{hasNarrower}, \text{hasBroader}, \text{hasNarrowerSelf}, \text{hasBroaderSelf}, \\ &\quad \text{hasEquivalent}\}, \\ I_K &= \{\text{“chapter”}, \text{“introduction”}, \text{“conclusion”}, \text{“definition”}, \text{“example”}, \\ &\quad \text{“illustration”}\}, \text{ and} \\ X_K &= \{\text{FragmentKeyword}(\text{“chapter”}), \text{FragmentKeyword}(\text{“introduction”}), \\ &\quad \text{FragmentKeyword}(\text{“conclusion”}), \text{FragmentKeyword}(\text{“definition”}), \\ &\quad \text{FragmentKeyword}(\text{“example”}), \text{FragmentKeyword}(\text{“illustration”}), \\ &\quad \text{hasNarrower}(\text{“chapter”}, \text{“introduction”}), \\ &\quad \text{hasNarrower}(\text{“chapter”}, \text{“conclusion”})\}. \end{aligned}$$

Let $\mathcal{X}_S = (C_S, R_S, I_S, X_S)$ be a style taxonomy, with

$$\begin{aligned} C_S &= \{\text{DataStyle}, \text{ReferenceStyle}, \text{FragmentStyle}\}, \\ R_S &= \{\text{hasNarrower}, \text{hasBroader}, \text{hasNarrowerSelf}, \text{hasBroaderSelf}, \\ &\quad \text{hasEquivalent}\}, \\ I_S &= \{\text{italic}, \text{bold+italic}, \text{underlined}\}, \text{ and} \\ X_S &= \{\text{DataStyle}(\text{bold+italic}), \text{DataStyle}(\text{italic}), \\ &\quad \text{ReferenceStyle}(\text{underlined}), \\ &\quad \text{FragmentStyle}(\text{bold+italic}), \text{FragmentStyle}(\text{italic})\}. \end{aligned}$$

Let D_0 be an empty semantic document model with the ontologies shown in example 4.2.15.

Then the base document model B from example 4.1.24 is mapped onto the semantic document model D from example 4.2.15 using the rules T_G , the taxonomies \mathcal{X}_K and \mathcal{X}_S , and the set of mappings \mathcal{M}

$$B, D_0 \xrightarrow{T_G, \{\mathcal{X}_K, \mathcal{X}_S, \mathcal{M}\}} D.$$

Adequacy and Other Approaches

We have shown a way to obtain complex semantic document models from a basic representation that is close to the technical representation of documents. This approach is powerful because it can incorporate domain knowledge, and it is flexible because it can be applied with little adaptation to most document formats. The approach is also well-structured in the sense that it separates the extraction logic from the background knowledge. This makes concrete implementations easier to understand, to maintain, and to transfer to other document formats or domains.

In chapter 12, we will show that the approach is indeed transferable with very few changes to the transformation rules.

We will now examine some properties of the rule-based approach.

A transition system (S, \rightarrow) is *confluent* iff $\forall s_1 \in S : (\forall s_a, s_b \in S : (s_1 \rightarrow^+ s_a) \wedge (s_1 \rightarrow^+ s_b)) \Rightarrow (\exists s_2 \in S : (s_a \rightarrow^+ s_2) \wedge (s_b \rightarrow^+ s_2))$, where \rightarrow^+ is the transitive closure of \rightarrow .

A rule system applied to a fact base can be interpreted as a transition system by interpreting the state of the fact base before and after the application of a rule as states in the transition system, and by interpreting the application of a rule as the transition between the two before and after states.

Proposition 5.1.26 (Confluence of the Rule System). *The rule system defined in description 5.1.24, with the conflict resolution techniques described above, is confluent.*

This can be easily seen because the conflict resolution techniques are defined in such a way (and for the sole purpose) that they ensure that in any state at most one rule can be applied. If more than one candidate rule exists for an object from the fact base, or if more than one object from the fact base has at least one candidate rule, then the order of application for these rules is well-defined. And since each rule application leads to a new state, it holds that $\forall s_1 \in S : (\forall s_a, s_b \in S : (s_1 \rightarrow^+ s_a) \wedge (s_1 \rightarrow^+ s_b)) \Rightarrow (s_a = s_b)$, i.e., the transition relation is right-unique.

Proposition 5.1.27 (Soundness and Completeness of the Transformation Rules). *Let B be a connected base document model. Let T_G be the set of transformation rules defined in description 5.1.24. Let \mathcal{K} be a set of appropriate background knowledge. Let D_0 be an empty semantic document model with appropriate ontologies \mathcal{O}_S and \mathcal{O}_T . Let D be semantic document obtained by applying T_G and \mathcal{K} to B and D_0 :*

$$B, D_0 \xrightarrow{T_G, \mathcal{K}} D.$$

Then D is the semantic document model induced by B and \mathcal{K} according to definition 5.1.5.

In other words, applying T_G in the order and using the conflict resolution techniques outlined above, is a sound and complete algorithm for obtaining the semantic document model induced by a connected base document model and a set of background knowledge.

Lemma 5.1.28 ($\mathcal{E}_G.\text{stack.top}$ is the Closest Fragment Induced by the Predecessor of a Media Object). *Whenever a transformation rule matches a media object m and is applied to it, the fragment on top of the stack ($\mathcal{E}_G.\text{stack.top}$) is the closest fragment induced by the media object processed before m . If m is the first media object to be processed, $\mathcal{E}_G.\text{stack.top}$ holds the root fragment of the semantic document model.*

Proof of Lemma 5.1.28. Proof by induction over the sequence of processed media objects. Let (m_0, \dots, m_n) be the sequence of media objects as applied to the transformation rules.

Base: If the processed media object is the first to be processed, then by definition the stack holds only the root fragment of the semantic document model.

Step: To show: $m_i \Rightarrow_0 \mathcal{E}_G.\text{stack.top}$.

If exists $m_j, j < i : m_j \Rightarrow f_j \wedge \nexists m_k, j < k < i : m_k \Rightarrow f_k$, for $f_j, f_k \in F$, then f_j was created by the third rule and pushed on top of the stack as its final operation. No other rule pushes anything on top of the stack, so the premise holds.

If no such m_j exists, then nothing has been pushed onto the stack by any rule, so the premise holds because of the induction base case. \square

Proof of Proposition 5.1.27. By definition of the algorithm, point 1 of definition 5.1.5 is satisfied: the existence of fragments with appropriate type information follows directly from the definition of the third rule in description 5.1.24.

Similarly, the existence of appropriate data annotations (point 2) follows directly from the definition of the algorithm. The fragment on top of the stack $\mathcal{E}_G.\text{stack.top}$ is the closest fragment induced by the current media object according to lemma 5.1.28.

The existence of appropriate reference relationships (point 3) also follows directly from the definition of the algorithm and the definition of the $\mathcal{E}_G.\text{getFragment}()$ function.

Let $m_i, m_j \in M, i < j : (m_i \neq m_j) \wedge m_i \rightrightarrows f_i \wedge m_j \rightrightarrows f_j \wedge \nexists m_k \in M, i < k < j : m_k \rightrightarrows f_k$, for $f_k, f_j, f_k \in F$ and $i, j, k \in \mathbb{N}$. If no such m_i, m_j exist, then at most one fragment is induced in the semantic document model, which therefore consists of only one or two fragments (including the root fragment). In this case, point 4 is trivially satisfied.

If they exist, however, then m_i has been processed by the third rule before m_j (because of the processing order as defined above and because of the rule's premise). Therefore, f_i is currently on top of the $\mathcal{E}_G.\text{stack}$.

If none of the types of f_i is narrower than or equal to any of the types of f_j , i.e., if none are in a direct or indirect **hasNarrower** or equivalence relationship, then the stack remains unchanged. The algorithm then puts f_i and f_j in a p relationship, thus satisfying point 4.

If, on the other hand, one of the types of f_i is indeed equal to or narrower than one of the types of f_j , then f_j is removed from the stack, thus preventing a p relationship between f_i and f_j , and thereby also satisfying point 4.

Let $m_i, m_{i+1} \in M : m_i \rightrightarrows_0 f_i \wedge m_{i+1} \rightrightarrows_0 f_{i+1}$, for $f_i, f_{i+1} \in F$ and $i \in \mathbb{N}$.

If m_i, m_{i+1} exist, then m_i has been processed before m_{i+1} (see above), and one of these cases is true:

1. $\exists f'_i, f'_{i+1} \in F : m_i \rightrightarrows f'_i \wedge m_{i+1} \rightrightarrows f'_{i+1}$ (both media objects directly induce a fragment, where $f'_i = f_i$ and $f'_{i+1} = f_{i+1}$),
2. $\exists f'_i \in F : m_i \rightrightarrows f'_i \wedge \nexists f'_{i+1} \in F : m_{i+1} \rightrightarrows f'_{i+1}$ (only the first media object directly induces a fragment, where $f'_i = f_i$),
3. $\nexists f'_i \in F : m_i \rightrightarrows f'_i \wedge \exists f'_{i+1} \in F : m_{i+1} \rightrightarrows f'_{i+1}$ (only the second media object directly induces a fragment, where $f'_{i+1} = f_{i+1}$), or
4. $\nexists f'_i, f'_{i+1} \in F : m_i \rightrightarrows f'_i \vee m_{i+1} \rightrightarrows f'_{i+1}$ (neither media object directly induces a fragment).

In the first case, f'_{i+1} is either inserted as a sub-fragment of f'_i : $(f'_i, f'_{i+1}) \in p$, which satisfies point 5. Or f'_{i+1} is inserted as a sub-fragment of a parent fragment of f'_i , which puts them in a (possibly indirect) successor relationship, also satisfying point 5.

In the second case, the closest fragment induced by m_{i+1} is the fragment f'_i induced by m_i , thus $f_i = f'_i = f_{i+1}$, which satisfies point 5.

In the third case, f'_{i+1} is either inserted as a sub-fragment of f_i : $(f_i, f'_{i+1}) \in p$, which satisfies point 5. Or f'_{i+1} is inserted as a sub-fragment of a parent fragment of f_i , which puts them in a (possibly indirect) successor relationship, also satisfying point 5.

Finally, in the fourth case, the closest fragment induced by both m_i and m_{i+1} is the same by definition, thus $f_i = f_{i+1}$, which satisfies point 5. \square

In principle, transformation rules can, given appropriate background knowledge, obtain semantic document models from any base document model. There are, however, some exceptions and restrictions for the rules given in description 5.1.24. Some special cases cannot be encoded in the background knowledge and require the creation of new rules or the adaptation of existing rules. This includes fragment or data indicators that span multiple media objects, for example a keyword occurring in one media object and a specific formatting option occurring in the next.

One caveat are indicators that can indicate two different things, with no way to differentiate between the two. For example, in a document where definitions and examples are both formatted in the same way and are not indicated by keywords, it is not possible to accurately classify the appropriate fragments in the semantic document model. Further analysis of the content of the paragraph in question may lead to a better classification. But unless this analysis is done beforehand and the results are included in the background knowledge, it is not used in the given rule set.

An alternative to transformation rules are graph selectors. Such a graph selector, for example an XPath expression, identifies and retrieves a media object of the base document model for further processing. However, using background knowledge in a language like XPath is technically cumbersome and inefficient: it requires that all background knowledge is represented in XML. Since XML data has a tree structure, where ontologies have a more complex graph structure, the background knowledge has to be represented in a more complex manner than necessary, in turn leading to complex XPath expressions. Additionally, there are no inference services for XML that would allow for automatically expanding transitivity or symmetry.

Imagine, for example, a keyword taxonomy containing the keywords “chapter”, “introduction”, and “intro”, with `hasNarrower`(“chapter”, “introduction”), `hasNarrower`(“chapter”, “intro”), and `hasEquivalent`(“introduction”, “intro”). In XML, this has to be represented in a way similar to the one shown in listing 5.1. Note how the equivalence between “introduction” and “intro” is represented twice, because there is no formalism for specifying the symmetry of the `<hasEquivalent>` tag. While it is possible to use an XML representation of an OWL ontology and to use the inference services for OWL, the prohibitively complex OWL XML syntax makes this approach infeasible in general.

```

1 <knowledge>
2   <keyword term="chapter">
3     <hasNarrower ref="introduction"/>
4     <hasNarrower ref="intro"/>
5   </keyword>
6   <keyword term="introduction">
7     <hasEquivalent ref="intro"/>
8   </keyword>
9   <keyword term="intro">
10    <hasEquivalent ref="introduction"/>
11  </keyword>
12 </knowledge>

```

Listing 5.1: Background Knowledge represented in XML

In order to obtain a similar effect to line 3 in example 5.1.18, i.e. selecting XML elements that contain one of the chapter keywords, a combination of two XPath expressions is required. This is shown in the simplified XSL program fragment in listing 5.2. The program iterates over every element of the XML document (line 1), saving the current element in a variable `$node` (line 2). Then it iterates over every relevant chapter keyword in the ‘keywords.xml’ file (line 3), namely the keyword ‘chapter’ itself (line 4) and every keyword that is narrower (line 6). After saving the current keyword in a variable `$keyword` (line 8), the program checks if the node saved above contains the keyword (line 9).

```

1 <xsl:for-each select="//*">
2   <xsl:variable name="node" select="."/ >
3   <xsl:for-each select="
4     document('keywords.xml')
5       //keyword[@term='chapter']/@term |
6     document('keywords.xml')

```

```

7           //keyword[@term='chapter']/hasNarrower/@ref">
8     <xsl:variable name="keyword" select="."/>
9     <xsl:if test="contains($node, $keyword)">
10      ...
11    </xsl:if>
12  </xsl:for-each>
13 </xsl:for-each>

```

Listing 5.2: Selecting XML elements for keywords with external background knowledge

The background knowledge could also be integrated directly into the graph selectors, i.e., into the XPath expressions. This removes the separation between domain knowledge and the extraction logic, so every time either the document format or the domain knowledge changes the expressions need to be adapted – usually at high expense, because they are far more complex than either transformation rules or ontologies with background knowledge. This is illustrated in listing 5.3. Line 3 contains the background knowledge and checks if the node contains one of the keywords.

In this short example, the XSL program version without external background knowledge is not only shorter, but appears easier to understand as well. This effect is quickly offset, however, if the amount of background knowledge becomes sufficiently large and complex.

```

1 <xsl:for-each select="//*">
2   <xsl:variable name="node" select="."/>
3   <xsl:if test="contains($node, 'chapter') or
4     contains($node, 'intro') or
5     contains($node, 'introduction'">
6     ...
7   </xsl:if>
8 </xsl:for-each>

```

Listing 5.3: Selecting XML elements for keywords with internal background knowledge

It is possible to define constants for each keyword, so that simple changes to keywords do not necessitate extensive updates in the XSL program. However, when new keywords are added, old keywords are removed, or relationships between existing keywords are added or removed, considerable effort is still required.

In general, graph selectors on their own are not sufficient for extracting semantic document models: often, several selectors need to be combined, and the resulting XML nodes need to be processed to obtain a semantic document model. Languages like XSL or XQuery can be used for this purpose. We will evaluate the use of XQuery as an alternative to rule languages in section 11.3.

Another alternative is to write the entire transformation in a programming language such as Java. While this is a valid approach, it is difficult to write such a program as clearly structured as a set of transformation rules. It takes much discipline to keep such a program as readable as the equivalent rules. This increases the cost of either the program creation or of the maintenance afterwards, in particular if the program is later adapted to a new domain.

Conclusion

In this section, we have defined a mapping from connected base document models onto semantic document models. Transformation rules, using a formalism based on description logics, were shown to be well suited for the task and alternatives were discussed. The employment of background knowledge was shown to be very beneficial. We have also provided a sound and complete algorithm for obtaining a semantic document model from a connected base document model.

5.2 Inference on Document Models

Having, finally, extracted a semantic document model from a digital document is certainly a worthy accomplishment. It is, however, not an end in itself: there is something that we want to do, that we want to have, or that we want to know that we need this document model for. To this end, we need to process the semantic document model further. Several utilisations for document models and the processing steps that lead to the desired results will be discussed in this section.

One important processing step is to extend and improve the document model itself. Clearly, this is not an end in itself, but rather serves to make the model better suited for whatever the final goal may be.

As we have already seen, a semantic document model D can be represented as an ontology \mathcal{O}_D . Based on this ontology, we can now use inference services for description logics to infer new facts and relations. Generalisations and equivalences can be used to derive new concept and role assertions, and role properties like transitivity, symmetry, or reflexivity can be used to derive new role assertions. In short, from the ontology \mathcal{O}_D a new ontology $\mathcal{O}_{D'}$ is inferred, which represents an extended semantic document model D' .

Example 5.2.1 (Extending Semantic Document Models). *Let $D = (F, f_0, s, p, T, c, \mathcal{O}_S, \mathcal{O}_T)$ be the semantic document model from example 4.2.15. Let D be represented by the ontology \mathcal{O}_D . Then, using inference rules, the following assertions (among others) can be inferred from \mathcal{O}_D :*

- ▶ $c(f_{42}, \text{Datastructure})$
from $\text{broaderThan}(\text{Datastructure}, \text{Binary Tree})$ and
 $c(f_{41}, \text{Binary Tree})$,
- ▶ $\text{Example}(f_{42})$
from $\text{Illustration}(f_{42})$ and $\text{Illustration} \sqsubseteq \text{Example}$.

These assertions are integrated into a new document ontology $\mathcal{O}_{D'}$ that represents an improved semantic document model D' .

The additional information greatly increases the coherence of the document model. Not only does it make explicit that there is an example for every definition in the document (even if the second example is in the form of an illustration), but it also makes the semantic relation between the two different topics within the document explicit. Now, there is a fitting example for every definition on every reading path through the document. While this fact was obvious to a human reader from the beginning, it has only now become possible for a computer program to recognise it as well.

Another important processing step is the transformation of a semantic document model into another type of model. This model can then for example be used for verification purposes. In chapter 10 we will show how a semantic document model is transformed into a temporal model used for model checking.

For the transformation into another type of model, usually little or no background knowledge is necessary. Most relevant background knowledge has already been integrated into the document model in previous steps. For this reason, and since there is only a single source “format” (namely a semantic document model), the arguments against using graph selectors do not apply here. While it is possible to use transformation rules similar to those described in section 5.1, we will now discuss the use of selectors as a viable alternative.

Definition 5.2.2 (Graph Selector). *For a path expression p , graph selectors s_1 and s_2 , and a condition c , the generic syntax of a graph selector is inductively defined as follows:*

- p (path selector),
- $s_1[c]$ (condition),
- $(s_1|s_2)$ (disjunction), and
- $(s_1\&s_2)$ (conjunction).

Path expressions describe a path in a tree or in a directed graph. Conditions are Boolean-valued expressions like (in-)equality assertions between path expressions and/or constant values, and Boolean combinations of other conditions.

Common graph selector languages are XPath (cf. section 3.1.2) and SPARQL (cf. section 3.3.5). In chapter 9, we will show how SPARQL can be used to transform semantic document models represented in OWL into other models.

Example 5.2.3 (Graph Selector). *The following expressions are XPath selectors:*

1. `knowledge/keyword`
2. `//*[term='intro']`
3. `//*[term='intro'] | #[term='introduction']`
4. `keyword[term=//hasNarrower/@ref]`

The following expressions are SPARQL selectors:

5. `SELECT ?s WHERE { ?s rdf:type vdk:Example }`
6. `SELECT ?s WHERE {
 ?x rdf:type vdk:Example .
 ?x vdk:hasTopic ?s }`
7. `SELECT ?s WHERE {
 { ?s rdf:type vdk:Definition } UNION
 { ?s rdf:type vdk:Example } }`

Definition 5.2.4 (Evaluating Graph Selectors against Models). *Let M be a model with a directed graph structure. W.l.o.g., we assume $M = (N, E)$, with a set of nodes N and a set of ordered pairs of nodes E .*

The evaluation context is a set $C_N \subseteq N$ that is used in the evaluation. All components of a selector are evaluated sequentially, and the evaluation context is updated with the result of the evaluation after every step. When all components of a graph selector have been evaluated, the result of the evaluation is the current evaluation context.

The initial evaluation context is $C_N = N$ for graph selectors like SPARQL. For selectors on tree structures like XPath, the initial evaluation context only contains the root node.

The evaluation of a graph selector against a model, written $s(M)$, is inductively defined as follows:

- $p(M)$ is the set of nodes from N that are reachable from C_N using the path described by the path expression p ,
- $s[c](M)$ is the set of nodes resulting from the evaluation of the graph selector s that satisfy the condition c ,
- $(s_1|s_2)(M)$ is the union of the nodes resulting from the evaluation of the graph selectors s_1 and s_2 , and
- $(s_1\&s_2)(M)$ is the intersection of the nodes resulting from the evaluation of the graph selectors s_1 and s_2 .

Example 5.2.5 (Evaluating Graph Selectors against Models). *Evaluated against the XML document from listing 5.1, the four XPath selectors from example 5.2.3 return the following values:*

1. all **keyword** elements within the **knowledge** element, namely the elements from lines 2, 6, and 9,
2. all elements that have a **term** attribute with the value ‘intro’, namely the element from line 9,
3. all elements that have a **term** attribute with either the value ‘intro’ or the value ‘introduction’, namely the elements from lines 6 and 9, and
4. all **keyword** elements that have a **term** attribute with a value that is the **ref** attribute of some **hasNarrower** element, namely the elements from lines 6 and 9.

Evaluated against an OWL implementation of the semantic document model from example 5.2.1, the three SPARQL selectors from example 5.2.3 return the following values:

5. all elements of type **Example**, namely f_{31} and f_{42} ,
6. the topics of all examples, namely “Datastructure” and “Binary Tree”, and
7. all elements of type **Definition** or **Example**, namely f_{21} , f_{41} , f_{31} , and f_{42} .

In general, a graph selector is used to obtain specific elements from a model. These elements are then put into new relationships, and combined with other elements that were previously selected. The resulting graph of elements from the existing model forms the new model. How exactly the selected elements are put into relationships must be specified in another language, for example XQuery or Java.

Yet another different processing step is the extraction of individual data points. Such data can be useful for answering questions about a document, or it can be used as background knowledge, for example in the form of statistical information about certain types of documents that helps designing more efficient transformation rules.

Single data points can be obtained from a semantic document model by means of selectors, as discussed before. Only now, these data elements are the desired result, not an intermediate result to be assembled into a larger structure.

Conclusion

In this section, we have discussed how a single document model can serve as a source for different tasks. We have seen how models for document verification can be obtained from many different sources through a single semantic metamodel, and that these models are of higher quality than verification models extracted directly from the source documents. We have also discussed different methods for obtaining these models.

5.3 Modelling Towers of Meta

After making extensive use of domain knowledge in the form of ontologies, we will now regard some questions about how such domain knowledge can be modelled. Specifically, we will examine how to treat knowledge that spans several metalayers.

Definition 5.3.1 (Layer). A *metalayer* in a knowledge base is a set of classes. A knowledge base may contain a sequence of several metalayers.

The *data layer* in a knowledge base is the set of individuals. None of these individuals may be instantiated by another individual.

Example 5.3.2 (Layer). As a simple example, imagine a book, for example “The name of the rose” by Umberto Eco. This specific book (or rather, an individual that represents the book) is located on the data layer. It is an instance of the general concept “book”, which is located on the first metalayer. “Book”, in turn, is an instance of a “concept” (in description logics terminology), or a “class” (in semantic web terminology, which we will use in this instance). “Class” is located on the second metalayer.

Omitting namespaces, the two instance assertions can be modelled in OWL as follows:

```
TheNameOfTheRose    type    Book .
Book                 type    Class .
```

Definition 5.3.3 (Metaconflict). A metalayer contains a *metaconflict* iff it contains two classes c_1 and c_2 , where c_1 is an instance of c_2 .

The only exception to this rule is the OWL class `Class`, which is also an instance of itself.

A knowledge base contains a metaconflict iff any of its metalayers contain a metaconflict.

Definition 5.3.4 (Maximal Metalayer). A metalayer m_i is *maximal* iff the addition of any new class c_1 from the knowledge base to m_i would either introduce a metaconflict in m_i , or c_1 is not in a relationship with any class c_2 from m_i or m_{i-1} , where m_{i-1} is the layer below m_i .

A knowledge base is *maximal* iff all its metalayers are maximal.

Example 5.3.5 (Multiple Metalayers). Continuing example 5.3.2, the book “The name of the rose”, represented by the individual `TheNameOfTheRose`, is an abstraction as well: it abstracts from the concrete instances of this book, for example from the specific printed version in Umberto Eco’s library (again, represented by an individual). In OWL, this reads as:

```
UmbertoEcosExemplar type TheNameOfTheRose .
TheNameOfTheRose    type    Book .
Book                 type    Class .
```

Now, in addition the the data layer that contains `UmbertoEcosExemplar`, we have three metalayers: one containing `TheNameOfTheRose`, one containing `Book`, and one containing `Class`. All of these metalayers are maximal, and do not contain metaconflicts.

Definition 5.3.6 (Tower of Meta). A finite stack of knowledge bases $tm = (kb_0, \dots, kb_n)$, where no object from a knowledge base kb_i instantiates a class from a knowledge base kb_j with $j < i$, is called a *tower of meta*. In other words, in a tower of meta, instantiation relationships are never defined “downwards”.

A single knowledge base kb that can be split into a finite stack of knowledge bases $tm_{kb} = (kb_0, \dots, kb_n)$ is also called a tower of meta, iff all instantiation relationships from kb are also represented in tm_{kb} and tm_{kb} is a tower of meta.

Example 5.3.7 (Tower of Meta). All knowledge bases in this section are towers of meta.

We will disregard the top metalayer containing the notions of “class” or “concept”, as they are a fixed part of the OWL or description logics formalisms, respectively. Description logics can only deal with a single data layer and a single metalayer, as there is no mechanism to treat concepts as instances of what could be called “second order concepts”. OWL does allow the specification of

multiple metalayers in OWL Full, but at a steep price: the semantics of OWL Full are not based on description logics but on first order predicate logics, which makes the language undecidable.

OWL 2 introduces the concept of *punning*, which is basically a retraction of the unique name assumption (cf. description 3.2.30) for individuals, classes, and properties. It does not, however, have any impact on the semantics of a knowledge base: it is merely a modelling convenience without logical consequences. Punning allows for the creation of an individual and a class with the same name, but both are still treated as separate entities by inference services, leaving towers of meta still without sufficient inference support.

A solution for this problem is to split a knowledge base with multiple metalayers into a stack of multiple knowledge bases, each containing only one data layer and a single metalayer. The first knowledge base in this stack contains the original data layer and the first metalayer. The second knowledge base contains the first metalayer as its data layer, and the original second metalayer as its only metalayer. This continues until all metalayers are contained in at least one knowledge base.

Since each knowledge base in this stack only has a single metalayer, their semantics are compatible with description logics. Inference services can now exploit this fact and can be employed on each knowledge base separately. Finally, the knowledge bases can be merged again to obtain the full range of modelled knowledge, including the newly inferred knowledge.

Example 5.3.8 (Splitting Ontologies). *The second knowledge base from example 5.3.5 can be split into two knowledge bases, the first*

UmbertoEcosExemplar type TheNameOfTheRose .

contains the individual UmbertoEcosExemplar on the data layer, and the class TheNameOfTheRose on the metalayer. The second knowledge base

TheNameOfTheRose type Book .

contains the individual TheNameOfTheRose on the data layer, and the class Book on the metalayer.

Listing 5.4 shows an algorithm in Java-pseudo code for splitting a knowledge base. It is defined on a `Model` data structure, named in reference to the data structure of the same name in the Jena-framework (cf. section 3.3.6), that basically represents a list of OWL statements consisting of subject, predicate and object (cf. section 3.3.3).

The algorithm consists of one primary and four auxiliary methods with the following signatures:

- ▶ `splitKnowledgeBase(Model) → List<Model>`,
- ▶ `getClasses(Set<Resource>, Model) → Set<Resource>`,
- ▶ `getIndividuals(Set<Resource>, Model) → Set<Resource>`,
- ▶ `getModel(Set<Resource>, Set<Resource>, Model) → Model`, and
- ▶ `containsOnlyTopElements(Set<Resource>, Model) → boolean`.

The `splitKnowledgeBase` method (lines 1 through 21) collects the individuals of the original data layer in line 2. These are the individuals that are not instantiated by anything and that are in no class-specific relationships like specialisations. In line 6, it collects the classes that are directly instantiated by any of these individuals, or that are equivalent to or are specialisations of one of these classes. Line 7 initialises the result list of split knowledge bases. The first knowledge

base containing the original data layer and the first metalayer is added to the result in line 8. The loop from line 9 to line 14 creates the additional knowledge bases. First, the old individuals are replaced by the classes of the previous knowledge base (line 10) as a starting point for the new data layer. Then, the classes that are directly instantiated by the new individuals are found (line 11). Next, in line 12, the list of individuals is updated with all individuals that instantiate any of the classes, no matter on which layer the individual originally appeared. Finally, a new knowledge base is created and added to the result (line 13). This is repeated until no higher metalayers are found, i.e., until the classes do not instantiate anything other than `Class` itself. Now, the final for-loop in line 15 finds any “orphaned” classes, i.e., classes that are in a relationship that defines them as classes instead of individuals, but that are not instantiated anywhere. These classes are then added to the metalayer of the first (lowest) knowledge base (line 19).

The `getClasses` method (lines 23 through 29) returns all classes that are directly instantiated by a given set of individuals. It then adds any classes that are equivalent to or a specialisation of one of the classes already found, until no new classes are found.

The `getIndividuals` method (lines 31 through 35) returns all individuals that directly instantiate any class from a given set of classes.

The `getModel` method (lines 37 through 58) creates a new knowledge base with the given individuals on the data layer and the given classes on the metalayer. It adds the statements from the original knowledge base that involve any of the individuals, but it filters out all statements indicating that an individual is also a class (lines 39 through 48). It also adds the statements from the original knowledge base that involve any of the classes, but it filters out all statements indicating that a class is also an individual (lines 50 through 56).

The `containsOnlyTopElements` method (lines 60 through 67) checks if a given set of classes contains no classes that are also individuals. It specifically disregards the OWL class `Class`.

The algorithm relies on two sets of properties, namely the set `transMeta = {type, domain, range}` that contains all OWL properties that cross the border between individuals and classes, and the set `intraMeta = {subclassOf, equivalentClass, subPropertyOf, equivalentProperty}` that contains all OWL properties that remain within a metalayer.

```

1 public List splitKnowledgeBase(Model model) {
2     Set individuals = /*
3         get all objects from model that are
4         - not subject of a statement (s,p,o) where p ∈ intraMeta, and
5         - not object of a statement (s,p,o) where p ∈ intraMeta ∪ transMeta */
6     Set classes = getClasses(individuals, model);
7     List result = new List();
8     result.add(getModel(individuals, classes, model));
9     do {
10        individuals = classes;
11        classes = getClasses(individuals, model);
12        individuals = getIndividuals(classes, model);
13        result.add(getModel(individuals, classes, model));
14    } while (!containsOnlyTopElements(classes, model));
15    for (Object c1: model)
16        if (/* none of the models in result contains c1 */)
17            if (/* model contains a statement (c1,p,c2) or (c2,p,c1),
18                with p ∈ intraMeta */)
19                /* then add (c1,p,c2) or (c2,p,c1) to the metalayer of the first model
20                    from line 8 */
21    return result;
22 }
23 private Set getClasses(Set individuals, Model model) {
24     Set result = /*
25         get all objects from model that an individual from individuals is an

```

```

        instance of,
26     then add all objects that are in a intraMeta relationship with one of the
        objects that have already been found
27     repeat until no new objects are found */
28     return result;
29 }
30
31 private Set getIndividuals(Set classes, Model model) {
32     Set result = /*
33         get all objects from model that instantiate a class from classes */
34     return result;
35 }
36
37 private Model getModel(Set individuals, Set classes, Model model) {
38     Set result = new Set();
39     for (Object i: individuals) {
40         Set statements = /*
41             get all statements  $(i,p,o)$  from model
42             - where  $p \notin \text{intraMeta}$  and
43             - where either  $p \neq \text{type}$  and  $o \notin \text{classes}$  or
44              $p = \text{type}$  and  $o \notin \text{individuals}$  and  $o \in \text{classes}$ , and
45             get all statements  $(s,p,i)$  from model
46             - where  $p \notin \text{intraMeta} \cup \text{transMeta}$  and
47             - where  $s \notin \text{classes}$  */
48         result.add(statements);
49     }
50     for (Object c: classes) {
51         Set statements = /*
52             get all statements  $(s,p,o)$  from model
53             - where  $c$  is either the subject or the object, and
54             - where  $p \in \text{intraMeta}$  */
55         result.add(statements);
56     }
57     return new Model(result);
58 }
59
60 private boolean containsOnlyTopElements(Set classes, Model model) {
61     if (/* classes contains an object o1,
62         where  $o1$  is an instance of another object  $o2$ 
63         and  $o2$  is not the class Class */)
64         return false;
65     else
66         return true;
67 }

```

Listing 5.4: Algorithm for separating multiple metalayers

This algorithm, however, can only be employed if a clean separation of the metalayers is possible on the knowledge base. In particular, it must be possible to separate the knowledge base into one data layer and one or more metalayers that are maximal and do not contain any metaconflicts. In addition, there must be a hierarchy between these layers that is defined by instantiations: no individual may instantiate a class that is on a lower layer. In other words, the original knowledge base must be a tower of meta. In particular, this precludes instantiation cycles like

```

a type b .
b type a .

```

We believe this to be a reasonable restriction, since the semantics of such circular instantiations are unclear and should therefore be avoided in modelling anyway.

Statements with arbitrary properties that cross layer boundaries other than those properties

in `transMeta` are disregarded by the algorithm. They do not prevent a clean separation of layers, so they do not foil the algorithm. However, they have no clear place in the hierarchy of layers, so they are omitted in the separated knowledge bases.

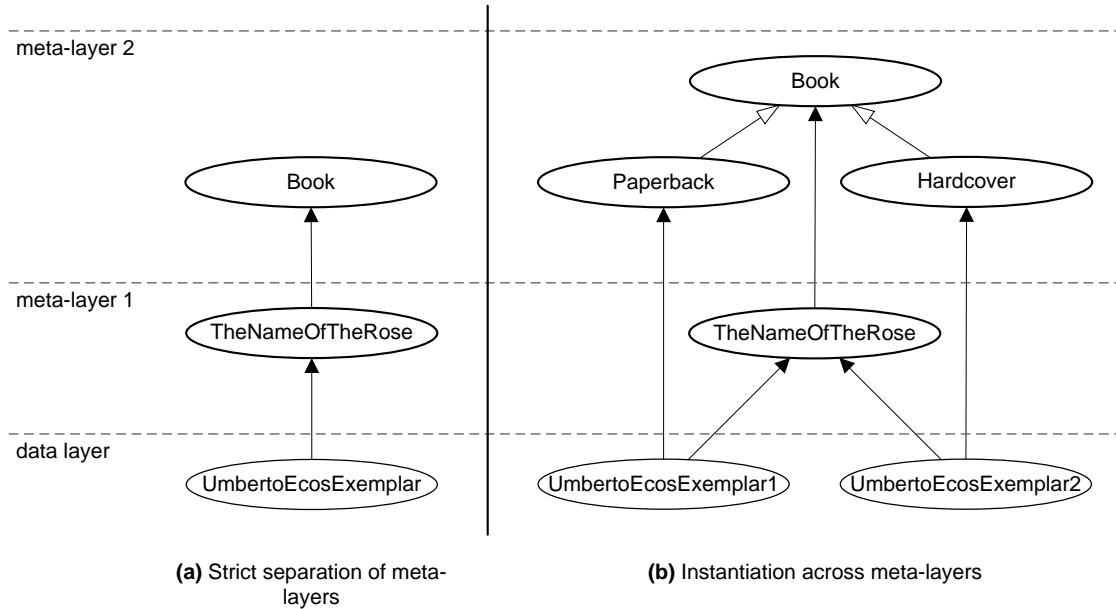


Figure 5.3: Metalayers of examples 5.3.5 and 5.3.9 (black arrowheads represent instantiation relationships, empty arrowheads represent generalisation relationships)

Example 5.3.9 (Multiple Metalayers (2)). *Let us extend example 5.3.5 so that the separation of metalayers is more complex. Umberto Eco now owns two exemplars of his book, one in hardcover and one in paperback format.*

```

UmbertoEcosExemplar1 type TheNameOfTheRose .
UmbertoEcosExemplar1 type Hardcover .
UmbertoEcosExemplar2 type TheNameOfTheRose .
UmbertoEcosExemplar2 type Paperback .
TheNameOfTheRose type Book .
Hardcover subclassOf Book .
Paperback subclassOf Book .

```

Both *Hardcover* and *Paperback* are modelled as specialisations of *Book*.

How this affects the metalayers is illustrated in figure 5.3. Note, however, that this illustration is not a normative partitioning of the knowledge base into metalayers but rather a simplified illustration. When instantiating the actual tower of knowledge bases, several objects occur on multiple layers across multiple knowledge bases. For example, *UmbertoEcosExemplar1* will occur on the data layer of the first knowledge base as an instance of *TheNameOfTheRose*, and on the data layer of the second knowledge base as an instance of *Paperback*.

The algorithm will again produce two knowledge bases. The first one contains both exemplars as individuals, and all four classes on its metalayer. It does not, however, contain the

instantiation of *TheNameOfTheRose* as a *Book*.

```

UmbertoEcosExemplar1 type TheNameOfTheRose .
UmbertoEcosExemplar1 type Hardcover .
UmbertoEcosExemplar2 type TheNameOfTheRose .
UmbertoEcosExemplar2 type Paperback .
Hardcover subClassOf Book .
Paperback subClassOf Book .
    
```

The second knowledge base also contains the exemplars, but without instantiating *TheNameOfTheRose*. Additionally, *TheNameOfTheRose* is included as an individual, instantiating *Book*.

```

UmbertoEcosExemplar1 type Hardcover .
UmbertoEcosExemplar2 type Paperback .
TheNameOfTheRose type Book .
Hardcover subClassOf Book .
Paperback subClassOf Book .
    
```

We will now use a larger example to show how the algorithm from listing 5.4 works.

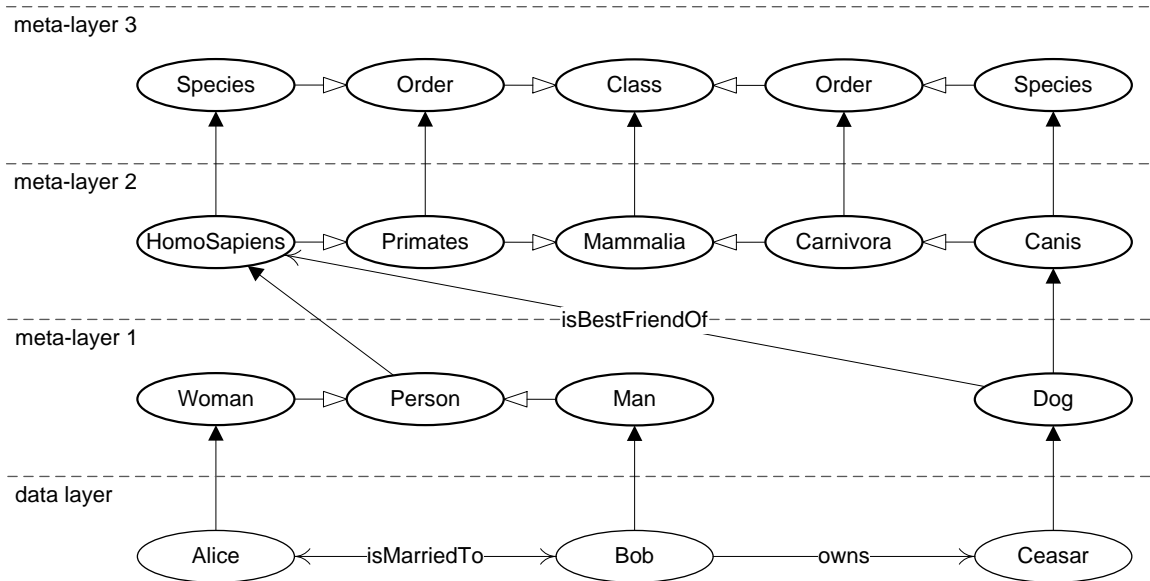


Figure 5.4: Metalayers of example 5.3.10 (black arrowheads represent instantiation relationships, empty arrowheads represent generalisation relationships, open arrowheads represent other relationships)

Example 5.3.10 (Multiple Metalayers (3)). *This example models a married couple and their*

dog in a biological taxonomy. An illustration of its metalayers is shown in figure 5.4.

```

Alice      isMarriedTo  Bob .
Bob        isMarriedTo  Alice .
Alice      type         Woman .
Bob        type         Man .
Bob        owns         Ceasar .
Ceasar     type         Dog .
Woman      subClassOf   Person .
Man        subClassOf   Person .
Person     type         HomoSapiens .
Dog        type         Canis .
Dog        isBestFriendOf HomoSapiens .
HomoSapiens subClassOf   Primates .
Primates   subClassOf   Mammalia .
Canis      subClassOf   Carnivora .
Carnivora  subClassOf   Mammalia .
HomoSapiens type         Species .
Primates   type         Order .
Mammalia   type         Class .
Canis      type         Species .
Carnivora  type         Order .
Species    subClassOf   Order .
Order      subClassOf   Class .

```

The algorithm first isolates all individuals, namely *Alice*, *Bob*, and *Ceasar*. It then lists all classes that are instantiated by any of these individuals, namely *Woman*, *Man*, and *Dog*. It now adds any equivalent classes or generalisations, namely *Person*, until no more classes are found. From this, a knowledge base containing the individuals, classes, instantiation relationships, relationships between individuals, and relationships between classes is created:

```

Alice      isMarriedTo  Bob .
Bob        isMarriedTo  Alice .
Alice      type         Woman .
Bob        type         Man .
Bob        owns         Ceasar .
Ceasar     type         Dog .
Woman      subClassOf   Person .
Man        subClassOf   Person .

```

Now, the set of classes is used as a starting point to find the classes on the next metalayer: all classes that are instantiated by any of the “old” classes are listed as the new set of classes, namely *HomoSapiens* and *Canis*. Again, this set is extended by more general classes until no more are found, resulting in the final set of classes {*HomoSapiens*, *Primates*, *Mammalia*, *Canis*, *Carnivora*}. For these classes, the instantiating individuals *Person* and *Dog* are found. A new knowledge base is created:

```

Person     type         HomoSapiens .
Dog        type         Canis .
HomoSapiens subClassOf   Primates .
Primates   subClassOf   Mammalia .
Canis      subClassOf   Carnivora .
Carnivora  subClassOf   Mammalia .

```

Again, the “old” classes serve as the basis for identifying the new ones: *Species*, *Order* and *Class*. This time, the instantiating individuals are the same as the old classes, leading to the

following new knowledge base:

```

Dog          isBestFriendOf HomoSapiens .
HomoSapiens type           Species .
Primates    type           Order .
Mammalia    type           Class .
Canis       type           Species .
Carnivora   type           Order .
Species     subclassOf    Order .
Order       subclassOf    Class .

```

Proposition 5.3.11 (Correctness of the Algorithm). *We will show that the algorithm satisfies three basic properties for knowledge bases that are towers of meta:*

1. *it constructs a stack of knowledge bases that fulfills the tower of meta properties,*
2. *none of the knowledge bases in this stack contains a metaconflict, and*
3. *each knowledge base in this stack is maximal.*

Proof of Proposition 5.3.11, Item 1. The algorithm creates a stack of knowledge bases (kb_0, \dots, kb_n) , the first (lowest) in line 8, and the others in line 13. Since the bottom knowledge kb_0 base may instantiate objects from any other knowledge base kb_i ($i > 0$), we only need to regard the knowledge bases created in line 13.

The only instantiation relationships integrated into a knowledge base kb_i are defined in line 40ff., and line 44 explicitly states that only elements from the set of classes may be instantiated. So it remains to be shown that if a class is instantiated in kb_i , it does not also occur in any kb_j , with $j < i$.

Assuming that such a class c , instantiated by an individual o , exists in kb_i , and that c also occurs in a kb_j ($j < i$), then there must exist a sequence of objects (o_0, \dots, o_m) , with o_0 instantiating both c and o_1 , o_k instantiating o_{k+1} , and o_m instantiating c .

This sequence must exist, because for any object to be placed on the metalayer of a knowledge base kb_k , it must be instantiated by an object that was on the metalayer of knowledge base kb_{k-1} (lines 24 and 10). For c to be on the metalayer of kb_j , o_0 must exist (with possibly $o_0 = o$ or $o_1 = c$). For c to be on the metalayer of kb_i , o_m must exist (with possibly $o_m = o_0$ or $o_m = o$). For o_m to be on the data layer of kb_i , it must be on the metalayer of a knowledge base below kb_i , for example kb_{i-1} . For o_m to be on the meta layer of kb_{i-1} , o_{m-1} must exist, and so forth. This is illustrated in figure 5.5.

However, if this sequence exists, then o_0 as an instance of c must be in kb_i . But then there is an instantiation relationship from kb_j to kb_i (o_0 instantiating c) and instantiation relationship from kb_i to kb_j (o_0 instantiating o_1). This violates the precondition that the original knowledge base is a tower of meta, because it precludes such cycles. Therefore, no such class c may exist. \square

Proof of Proposition 5.3.11, Item 2. No knowledge base created by the algorithm can contain two classes c_1 and c_2 on its metalayer, with c_1 instantiating c_2 . This is ensured in line 44, which explicitly forbids the instantiation of a class on the same metalayer. \square

Proof of Proposition 5.3.11, Item 3. Every knowledge base kb_i created by the algorithm contains all classes that are either in a relationship with another class from kb_i , or with an individual from the data layer of kb_i . This is ensured by lines 26 and 33, respectively. \square

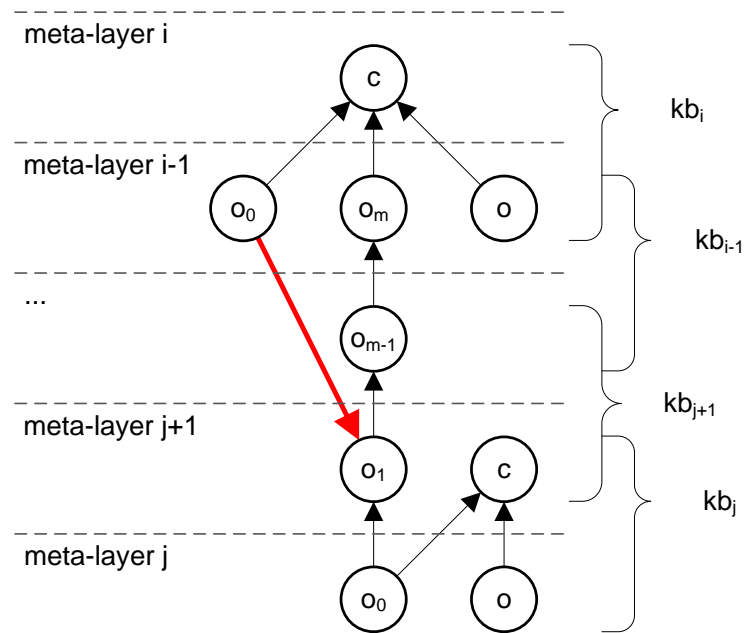


Figure 5.5: Metalayers and knowledge bases (black arrowheads represent instantiation relationships)

Proposition 5.3.12 (Runtime Complexity of the Algorithm). *The complexity of the algorithm is bounded by $O(|o|^2 \cdot |s|)$, with $|o|$ the number of objects and $|s|$ the number of statements in the knowledge base.*

In other words, splitting a suitable knowledge base into a tower of meta with the algorithm shown in listing 5.4 requires effort that is cubic in the size of the original knowledge base.

Proof of Proposition 5.3.12. The algorithm is dominated by its `while`-loop (lines 9 through 14). In the worst case, each object in the original knowledge base is on a separate layer. In this case, the loop has to be executed $|o|$ times, with $|o|$ the number of objects in the knowledge base.

The complexity of line 11 is determined by the `getClasses` method. First, for each object in a given set, it has to check statements that involve this object. This results in at most $|o| \cdot |s|$ execution steps, with $|s|$ the number of statements in the knowledge base. Then, for each object that has been identified as a class, the method has to find statements that connect this object to other classes. This process is repeated at most $|o|$ times, resulting in a maximum of $|o|^2 \cdot |s|$ steps.

This can, however, be reduced to $|o| \cdot |s|$ because only newly found classes need to be scrutinised again. This reduces the maximum number of objects for which statements need to be checked from $|o|^2$ to $|o|$, since no object needs to be looked at twice.

This results in a total upper bound of $O(|o| \cdot |s| + |o| \cdot |s|) = O(|o| \cdot |s|)$ for the `getClasses` method.

For line 12, the complexity is determined by the `getIndividuals` method, which for each object in a given set has to check all statements involving said object. This makes $O(|o| \cdot |s|)$ an upper bound for the `getIndividuals` method.

The `getModel` method determines the complexity of line 13. This method has to iterate twice over sets of objects and regard statements for each object, leading, again, to an upper bound complexity of $O(|o| \cdot |s| + |o| \cdot |s|) = O(|o| \cdot |s|)$.

For similar reasons as the `getIndividuals` method, the `containsOnlyTopElements` method has an upper bound of $O(|o| \cdot |s|)$.

In total, the complexity of the algorithm is bounded by $O((|o| \cdot |s| + |o| \cdot |s| + |o| \cdot |s| + |o| \cdot |s|) \cdot |o|) = O(|o|^2 \cdot |s|)$. \square

Conclusion

In this section we have discussed a challenge in modelling complex worlds. We have presented a novel approach to dealing with this challenge under certain conditions and have given an effective and efficient algorithm that implements the approach.

Chapter 6

Background Knowledge

In chapters 4 and 5, we have already made extensive use of background knowledge. We have seen how background knowledge, formalised as an ontology, can be used to enrich document models. Specifically, semantic document models make use of two background knowledge ontologies: the structural ontology \mathcal{O}_S , and the terminological ontology \mathcal{O}_T . We have also seen how background knowledge can enable a transformation process between document models, specifically in the form of a keyword taxonomy \mathcal{X}_K and a style taxonomy \mathcal{X}_S used in transformation rules.

In this chapter, we will explore the notion of background knowledge in more detail, with particular focus on how to obtain and formalise background knowledge.

So far, we have only seen background knowledge in the form of ontologies. In general, however, background knowledge can exist in many other forms, most of which are not formalised. The transformation rules introduced in section 5.1, for instance, make use of background knowledge as part of the transformation logic. Not only do they use taxonomical knowledge, but knowledge about the composition of the document models is also incorporated directly into the design of the rules.

Description 6.0.13 (Background Knowledge). *Background knowledge is knowledge about an entity, a process, or a relationship that is not explicitly contained within its object.*

Example 6.0.14 (Background Knowledge). *The ontologies \mathcal{O}_S and \mathcal{O}_T from example 4.2.15, the ontologies \mathcal{O}_S and \mathcal{O}_T from example 4.2.36, and the taxonomies \mathcal{X}_K from example 5.1.1 and \mathcal{X}_S from example 5.1.4 are formalisations of background knowledge. The transformation rule in example 5.1.18 contains background knowledge about how the structure of a base document model can best be transformed into a semantic document model.*

Description 6.0.15 (Domain Knowledge). *Domain knowledge is knowledge that is specific for a domain, and that is potentially only valid there. Some background knowledge is specific to a domain, and some domain knowledge is also background knowledge. Therefore, sometimes domain knowledge can be used as background knowledge.*

Example 6.0.16 (Domain Knowledge). *As discussed in section 4.2.1, both ontologies \mathcal{O}_S and \mathcal{O}_T from example 4.2.15 are domain knowledge, and more specifically, domain specific background knowledge.*

6.1 Obtaining Knowledge from Wikipedia

Social collaboration projects offer a large source of knowledge about many different domains. They are created by volunteers, who are often either interested laypersons or professionals in certain domains. In the latter case, they bring valuable expertise in their domain to the project.

However, not all information contained in such a social collaboration project is necessarily correct. Human error is not always detected (and corrected) immediately. For some topics there are also different opinions, and often not all of them can be integrated into the project. Sometimes only a single opinion can be implemented, even if there is no general consensus on which opinion is the “correct” one, or if there even is a correct opinion. For very controversial issues, where contributors cannot agree on a subject that they feel strongly about, sabotage or so-called “edit wars” can happen, with different users continuously overwriting contributions of others.

Diverging or evolving options can lead to content changes in a project over time, so the “knowledge” contained in such a project is not fixed (cf. also the discussion on fixed vs. fluid in section 4.1).

Wikipedia is currently the largest and most well-known social collaboration project. Through its size, especially through the number of contributors, it can act as a normative reference. A huge number of people use and implicitly trust the content of Wikipedia, which reinforces this interpretation. In mid-2012, the German district court in Tübingen ruled in favour of Wikipedia, citing that it served the public interest of supplying information¹. Different from standardisation bodies or many other normative authorities, Wikipedia’s supposed normativity has not been artificially created and is not supported by contracts or laws, but has emerged naturally and is silently accepted in many cases.

Yet even (or maybe especially) a project as large as Wikipedia is not free of factual errors, omissions, misconceptions, or misrepresentations [Dal09]. Some options of dealing with this are shown in section 6.1.1. Wikipedia also suffers from the issue of multiple conflicting opinions on topics, as named above.

These issues also apply to its category structure, which makes the categorisation (or classification) somewhat arbitrary. It can be argued that classification systems are always arbitrary or dependent on a cultural domain. Jorge Luis Borges illustrates this in his 1942 response to a 1668 work of John Wilkins [Wil68], which he takes ad absurdum by listing a completely ridiculous (yet factually correct) taxonomy [Bor99].

This means, however, that projects like Wikipedia do not actually contain “hard” knowledge. They contain a soft form of knowledge that represents a best-effort approach, or the current consensus on some data. It might be questioned if something like “hard” knowledge even exists, or at least if it can be produced with any reliability by fallible humans. Even “hard” facts like measurements or historical data depend on the reliability of instruments, on the correctness of scientific theories, and on the reliability of historical sources.



Science does not aim at establishing immutable truths and eternal dogmas; its aim is to approach the truth by successive approximations, without claiming that at any stage final and complete accuracy has been achieved.” – Bertrand Russel

We will, however, delegate this discussion into the realm of philosophy and simply recognise

¹File number 7 O 525/10, original quote “Mit dieser Gewährleistung korrespondiert insbesondere das Interesse der Öffentlichkeit an einer ausreichenden Versorgung mit Informationen.”, available online at <http://openjur.de/u/582363.html>, last accessed on 05-2013.

that most human knowledge is prone to be incomplete or even erroneous. Further discussion can for example be found in [Röt99].

How does this impact our attempt to obtain background knowledge from Wikipedia? Primarily, it means that all results based on this background knowledge must be taken with a grain of salt as they may not be entirely reliable. However, since virtually all (background) knowledge may contain errors, this is not really anything new. At the same time, it can even be advantageous when changes in how events or relations are commonly perceived, or simple factual changes, also make their way into our document models.

Before we continue, we will give a simplified definition of Wikipedia that we will use throughout this section.

Definition 6.1.1 (Wikipedia). *Wikipedia in a language l is a tuple $w_l = (C_l, s_l, A_l, r_l, m_l, n_l, t_l)$, where*

- C_l is a set of categories,
- $s_l \subseteq C_l \times C_l$ is a sub-category relation,
- A_l is a set of articles,
- $r_l \subseteq A_l \times A_l$ is a reference relation between articles,
- $m_l \subseteq A_l \times C_l$ is a membership relation between articles and categories,
- n_l is a function that assigns to each article $a \in A_l$ and to each category $c \in C_l$ a name that is unique within A_l or C_l , respectively, and
- t_l is a function that assigns to each article $a \in A_l$ its textual content.

The complete Wikipedia is a tuple $\mathcal{W} = (L, W, t_{(l_1, l_2)}^c, t_{(l_1, l_2)}^a)$, where

- L is a set of languages,
- $W = \{w_l \mid l \in L\}$ is a set of Wikipedias in different languages,
- $t_{(l_1, l_2)}^c \subseteq C_{l_1} \times C_{l_2}$, $l_1 \neq l_2 \in L$, is a set of translation relations between categories, and
- $t_{(l_1, l_2)}^a \subseteq A_{l_1} \times A_{l_2}$, $l_1 \neq l_2 \in L$, is a set of translation relations between articles.

This definition ignores aspects of Wikipedia that are not relevant for our use case, such as multiple versions of articles, or references to specific sections of an article.

Remark 6.1.2. *Note that the sub-category relation s_l is not nearly as strict in its semantics as the sub-concept relation \sqsubseteq in description logics. We will discuss this in more detail below.*

Example 6.1.3 (Wikipedia). *As an example, we will regard a partial version $\mathcal{W} = (L, W, t_{(l_1, l_2)}^c, t_{(l_1, l_2)}^a)$ of the complete Wikipedia, reduced to the set of languages $L = \{\text{en, de, nl, fr, it, pl, es, ru, pt}\}$, namely English, German, Dutch, French, Italian, Polish, Spanish, Russian, and Portuguese.*

Table 6.1 shows some statistics for the categories of the various languages, as they were found in Wikipedia at the end of May 2012. $|C_l|$ represents the number of categories for language l . $\phi|n_l|$ represents the average length (number of characters) of category names for language l , while $\phi||n_l||$ represents the average number of tokens, i.e., whitespace-separated words, in category names. $\phi|t_{(l,)}^c|$ represents the average number of translations from a category in language l into any other language from L . $\phi|s_l|$ represents the average number of sub-categories for a category.*

Similarly, table 6.2 shows some statistics for the articles of the various languages. $|A_l|$ represents the number of articles for language l . $\phi|n_l|$ represents the average length of article names for language l , while $\phi||n_l||$ represents the average number of tokens in article names. $\phi|t_{(l,)}^a|$ represents the average number of translations from an article in language l into any other language from L . $\phi|m_l|$ represents the average number of category membership relationships for articles, and $\phi|r_l|$ represents the average number of reference relationships for articles.*

l	$ C_l $	$\phi n_l $	$\phi n_l $	$\phi t_{(l,*)}^c $	$\phi s_l $
en	852,960	28.24	3.96	0.72	2.01
de	144,355	21.51	2.40	2.16	1.93
nl	67,531	20.17	2.55	3.55	2.66
fr	203,450	24.37	3.51	2.15	1.89
it	162,345	25.77	3.77	1.98	1.81
pl	97,835	24.32	3.12	2.92	1.77
es	176,116	25.46	3.85	2.29	1.96
ru	203,359	25.55	3.26	2.11	1.83
pt	137,137	23.53	3.60	2.65	1.69

Table 6.1: Statistics for the Wikipedia categories from example 6.1.3, as of 2012-05-30.

l	$ A_l $	$\phi n_l $	$\phi n_l $	$\phi t_{(l,*)}^a $	$\phi m_l $	$\phi r_l $
en	9,489,965	19.51	2.80	0.53	1.57	14.68
de	2,627,753	18.03	2.11	1.23	1.90	17.72
nl	1,528,477	17.92	2.19	1.85	1.11	15.21
fr	2,703,643	21.28	2.82	1.33	1.48	19.54
it	1,396,641	18.25	2.59	2.28	0.93	28.24
pl	1,212,297	18.35	2.38	2.41	1.80	23.35
es	2,291,957	20.93	3.04	1.29	1.00	16.55
ru	2,067,504	20.25	2.44	1.25	1.12	15.87
pt	1,430,570	20.01	2.77	1.86	1.08	15.78

Table 6.2: Statistics for the Wikipedia articles from example 6.1.3, as of 2012-05-30.

In total, W contains 2,045,088 categories and 24,748,807 articles in nine languages.

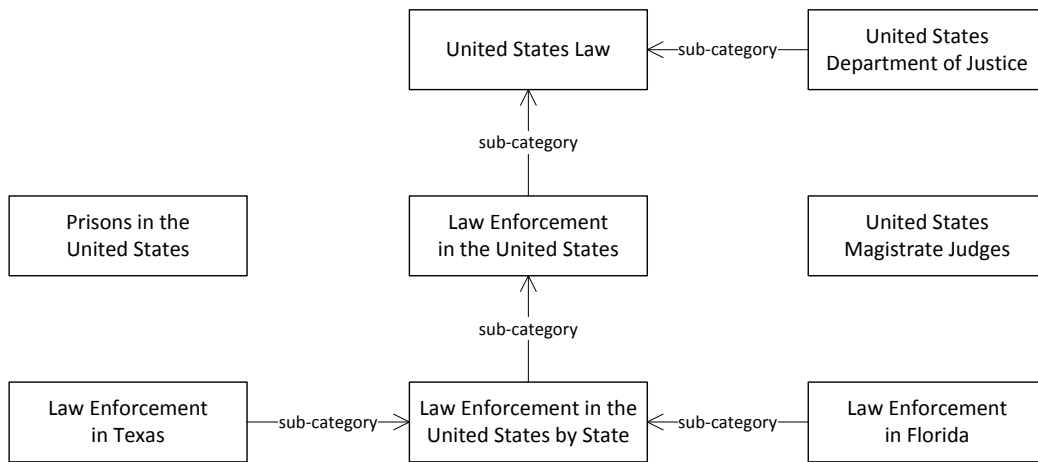
It is interesting to note that the average number of translations from the English language version is decidedly smaller both for categories and for articles than in other languages. This is caused by the larger absolute numbers of categories and articles in English, leaving many with no counterpart in other languages. Vice-versa, the number of translations for Dutch categories is well above average, caused by the small number of existing categories in this language.

It is also noteworthy that articles in Italian have the highest number of references among themselves, while at the same time they have the lowest number of category membership relationships. Similar, if less extreme, peculiarities exist for most languages, which is most likely a result of the evolving habits of different Wikipedia communities.

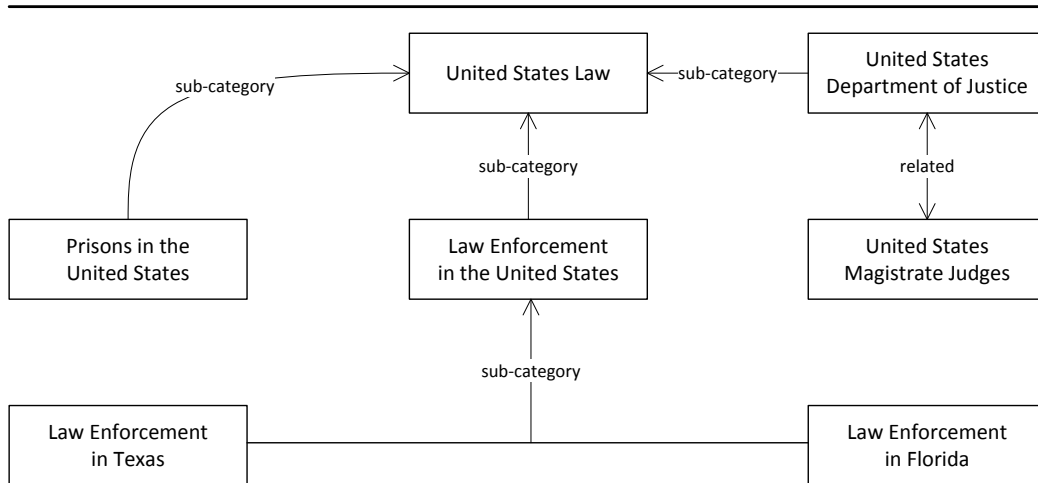
6.1.1 Preparing Wikipedia for Knowledge Extraction

We have explored several options of making Wikipedia more suitable for knowledge extraction, i.e., of correcting omissions and removing unnecessary content [SPF10].

Example 6.1.4 (Preparing Wikipedia). *Figure 6.1 (a) shows an excerpt of the English Wikipedia as of 2009-11-30, with eight categories and five sub-category relationships. We will use this example to illustrate the options discussed in this section. Figure 6.1 (b) shows the resulting category structure after applying the extension approaches.*



(a) Original Wikipedia (English) categories as of 2009-11-30



(b) Extended Wikipedia (English) categories as of 2009-11-30

Figure 6.1: Wikipedia categories before (a) and after (b) extension

Utilising Translations (ϕ_{trans})

Our first step is to utilise other language versions to extend the s_l , m_l , and r_l relations for a language l . W.l.o.g., we will describe this approach for s_l , but it can be used for m_l and r_l analogously.

Let $l \in L$ be a specific language, and $c_1, c_2 \in C_l$ be two categories from w_l for which no sub-category relationship has been defined, i.e., with $(c_1, c_2) \notin s_l$. We will now see if such a relationship has been defined in another language for translations of c_1 and c_2 . If

$$\exists l' \in L : \exists c'_1, c'_2 \in C_{l'} : (c_1, c'_1) \in t_{(l,l')}^c \wedge (c'_1, c'_2) \in s_{l'} \wedge (c'_2, c_2) \in t_{(l',l)}^c,$$

then add (c_1, c_2) to s_l . In other words, if two categories are in a sub-category relationship in another language, they should also be in a sub-category relationship in the language l .

This approach depends on the accuracy of the t^c relation, and on the assumption that omitting a t^c relationship is more likely than inserting an erroneous one. To mitigate the effects of these dependencies, it is possible to require that a s_l relationship exists in not one but two (or more) other languages.

Unfortunately, such a requirement greatly reduces the effectiveness for the English language. The English version of Wikipedia is by far the largest, which necessarily results in the lowest average number of translations per category (cf. table 6.1), because many categories simply do not exist in other languages. Therefore, requiring not one but two translations for a pair of categories greatly reduces the number of candidates.

Example 6.1.5 (Utilising Translations to Extend Relations). *The English Wikipedia w_{en} as of 2009-11-30 contains two categories “United States Law” and “Prisons in the United States” that are not in a sub-category relationship, as seen in example 6.1.4.*

On the other hand, the French Wikipedia w_{fr} of the same date contains two categories “Droit des États-Unis” and “Prison aux États-Unis” that are in a sub-category relationship, and Wikipedia contains two translation relationships (“United States Law”, “Droit des États-Unis”) $\in t_{\text{en,fr}}^c$ and (“Prison aux États-Unis”, “Prisons in the United States”) $\in t_{\text{fr,en}}^c$.

From this, we can infer that “Prisons in the United States” is a sub-category of “United States Law”. Given the broad semantics of the sub-category relation, this can be regarded as accurate.

The complexity of ϕ_{trans} is bounded by $O(|C_l|^2)$, because for every category ($|C_l|$), each outgoing reference (sub-category relationship) has to be checked ($|C_l|$). The translations back and forth can all be checked in constant time, leaving the complexity quadratic in the number of categories. However, in practise, the number of outgoing references is strictly limited and can be regarded as a constant factor (cf. table 6.1), leading to a more realistic complexity of

$$\Theta(|C_l|).$$

Utilising Lexical Databases (ϕ_{lex})

Our next approach uses a *lexical database* to find relationships based on semantic similarity.

Definition 6.1.6 (Lexical Database). *A lexical database is an ontology with words as individuals, and at least the roles **synonym** and **related** that are used to represent semantically equivalent words and semantically similar words, respectively. Both roles are symmetric.*

Example 6.1.7 (Lexical Database). *Wordnet [Mil06] is a well-known lexical database for the English language. Another powerful form of lexical databases are thesauri, such as the Wiktionary² project.*

²<http://www.wiktionary.org/>, visited 05/2013

Again, we will describe the approach for categories, but it can be similarly used on articles.

For the name $n = n_l(c)$ of a category c consisting of $\|n\|$ words, let $n[i]$ denote the i^{th} word of n , with $0 < i \leq \|n\|$. We will disregard so-called *stop words*, words that carry no or little relevance, such as “the”, “on”, or “is”.

Let $\mathcal{O}_L = (\emptyset, \{\text{synonym, related}\}, I_L, X_L)$ be a lexical database. For each pair of categories $c_1, c_2 \in C_l$, we calculate a semantic similarity score $s_3(c_1, c_2)$ as follows, where $n_1 = n_l(c_1)$ and $n_2 = n_l(c_2)$:

$$s_3(c_1, c_2) = \frac{\sum_{i=1}^{\|n_1\|} \sum_{j=1}^{\|n_2\|} s_d(n_1[i], n_2[j])}{\frac{\|n_1\| + \|n_2\|}{2}}$$

This score determines a semantic similarity for each pair of words in the names of c_1 and c_2 (numerator), normalised in relation to other scores by the average number of tokens of the two names (denominator). As described in section 4.2, we will assume that every word is in a grammatical base form.

We define the semantic distance $s_d(w_1, w_2)$ between two words as:

$$s_d(w_1, w_2) = \begin{cases} 0.8 & \text{if } w_1 = w_2 \text{ or } \text{synonym}(w_1, w_2) \in X_L \\ 0.5 & \text{if } \text{related}(w_1, w_2) \in X_L \\ 0.0 & \text{otherwise} \end{cases}$$

The constants 0.8 and 0.5 have been chosen arbitrarily, but have been validated through experiments on several thousand category names in multiple languages.

While other, more complex score functions might lead to even better results, the high quality of the results obtained with s_3 (see below) coupled with its low calculation cost make s_3 a good choice. In future work, we plan to compare the precision and efficiency of s_3 with other score functions, in particular with even simpler functions that trade precision for speed.

If the semantic similarity score for two categories is above a certain threshold that is empirically determined for each language, then the two categories are considered to be related to each other. For articles, this can be represented directly through references, using the r_l relation. For categories, this can be represented either as a mutual sub-category relationship $s_l(c_1, c_2)$ and $s_l(c_2, c_1)$, which is consistent with the loose sub-category semantics discussed below, or it can be used as confirmation of a sub-category relationship discovered in the first extension step outlined above.

Example 6.1.8 (Utilising a Lexical Database to Extend Relations). *The English Wikipedia w_{en} as of 2009-11-30 contains two categories “United States Department of Justice” and “United States Magistrate Judges” that are not in a sub-category relationship, as seen in example 6.1.4.*

Two words, “United” and “States”, occur in both category names. The words “Department” and “Magistrate” have no relation to each other, and the stop word “of” is ignored. Finally, the words “Justice” and “Judge” are found to be related according to Wordnet. This leads to a score s_3 of $(0.8 + 0.8 + 0.0 + 0.5)/((4 + 4)/2) = 0.525$. For the English Wikipedia, a threshold of 0.52 has been determined empirically, so the two categories are considered to be related.

Experiments have shown that this approach can only be used reliably if the respective category names contain three words or more. Otherwise, there is not enough evidence to support any perceived semantic relation. For instance, homonyms (i.e., words with the same spelling or pronunciation but with different meaning) can lead to false positives if their occurrence is not offset by a quantity of other words. This also necessitates great care when determining the threshold.

As an example, regard the two categories “Parks and Commons in Edinburgh” and “Creative Commons”. Not only do they share one word, but the same word, “commons”, is actually a somewhat archaic word for “park”. This raises the score for this pair of categories to a respectable 0.52. Only because the second category name is too short for a meaningful analysis can it be disregarded.

The score function is symmetric, i.e., $s_3(c_1, c_2) = s_3(c_2, c_1)$ for categories $c_1, c_2 \in C_l$ because $\sum_{i=1}^{\|n_1\|} \sum_{j=1}^{\|n_2\|} s_d(n_1[i], n_2[j]) = \sum_{j=1}^{\|n_2\|} \sum_{i=1}^{\|n_1\|} s_d(n_1[i], n_2[j])$ (commutativity of addition). Therefore, each pair of categories only has to be regarded once, resulting in $\sum_{i=1}^{|C_l|} |C_l| - i = |C_l|^2 - |C_l| \cdot \frac{|C_l|+1}{2}$ applications of the score function. This is bounded by $O(|C_l|^2)$.

The score function itself requires effort that is quadratic in the number of tokens in a category (or article) name $\phi\|n_l\|$, which is a constant with an average value smaller than 4. The complexity of the score function is thus only determined by the complexity of accessing the lexical database. This complexity can be approximated as finding a role assertion in \mathcal{O}_L , which can be accomplished with a binary search in $O(\log |I_L|)$.

An upper bound for the overall complexity of extending a Wikipedia w_l in a language l is therefore

$$O(|C_l|^2 \cdot \log |I_L|).$$

If the lexical database can make use of an index structure, the complexity of an assertion lookup for finding terms that are related to a given term is reduced to $O(1)$, resulting in an upper bound for ϕ_{lex} of

$$O(|C_l|^2).$$

Using an inverted index on the category names further reduces the complexity. An inverted index that maps words onto sets of categories containing this word allows us to only regard pairs of categories that either have at least one word in common, or that use a synonym or related word. In particular, for every category ($|C_l|$), for every word in this category ($\phi\|n_l\|$, constant factor c_1), we look up every synonym and related word in the lexical data base (constant factor c_2). The number of these related words is theoretically bounded by $|I_L|$, the size of the lexical database. However, in practice, this number is smaller than 20 in most cases, allowing us to regard it as a constant factor as well (c_3). Now, for each of these related words, we find the categories that also contain one of these words using the inverted index (constant factor c_4). This leads to a lower and upper bound for the overall complexity for ϕ_{lex} of $\Theta(|C_l| \cdot c_1 \cdot c_2 \cdot c_3 \cdot c_4) =$

$$\Theta(|C_l|).$$

Unfortunately, while this method scales well, the numbers for $|C_l|$ as well as some of the constant factors are very high, which leads to very high absolute runtime costs (see below).

Removing Superfluous Categories (ϕ_{remC})

Wikipedia, as an online encyclopaedia, has a structure that is optimised for browsing and viewing, not for knowledge representation. This is apparent in categories whose sole purpose is the sorting or some other visual re-representation of data. These categories usually have a super-category whose content they arrange differently, for example ordering or grouping it by some criterion. They are indicated by the keyword “by” in English, as in “Law Enforcement in the United States by State”, or by the phrase “lists of”. Similar keywords exist for other languages.

Our third step is to remove these superfluous categories to optimise the structure for knowledge representation. This step is only applied to categories, but has ramifications for the relations m_l and s_l , as well.

Superfluous categories must have a super-category with the same topic, but without the specific ordering or grouping. In particular, a category named “ X by Y ”, where X and Y consist of one or more words, must have a super-category named “ X ” to be recognisable as superfluous. Similarly, a category named “Lists of X ” must have a super-category named “ X ”. There are many categories that contain one or more keywords, but that are not just visual representations of the content of another category. Most of these are ignored by our approach (see below) because they lack the telling super-category.

If a category c_x is recognised as superfluous, it is removed from C_l . Its super-category c_p is identified, and (c_x, c_p) is removed from s_l . Then, all occurrences of c_x in the sub-category and article-membership relations s_l and m_l are replaced by c_p . This means that all its sub-categories and member articles are directly attached to its super-category.

Example 6.1.9 (Removing Superfluous Categories). *The English Wikipedia w_{en} as of 2009-11-30 contains four categories “Law Enforcement in the United States”, “Law Enforcement in the United States by State”, “Law Enforcement in Texas” and “Law Enforcement in Florida”. The first is a super-category of the second, and the last two are sub-categories of the second, as seen in example 6.1.4.*

*The category “Law Enforcement in the United States by State” can be recognised as a superfluous ordering category by its name pattern of **topic-keyword-ordering condition** and by the existence of a super-category with the same topic. When removed, its two sub-categories become direct sub-categories of “Law Enforcement in the United States”.*

However, not all categories that appear to be superfluous at first glance are devoid of new information. The category “Writers by Genre”, a sub-category of “Writers”, clearly contains merely a grouping of the contents of its super-category. On the other hand, the category “Novels by Jane Austen”, a sub-category of “Novels” may very well be regarded as carrying additional information. It is not merely a grouping by author, but an attribution to a specific author. However, this distinction is in the eye of the beholder and should be determined by the manager of the knowledge base obtained from Wikipedia.

For reasons similar to those for ϕ_{trans} , ϕ_{remC} is bounded by

$$\Theta(|C_l|).$$

Removing Superfluous References (ϕ_{remR})

The primary focus of Wikipedia is not to represent knowledge, but to make knowledge accessible to its readers. This is a subtle but important difference. Next to superfluous categories as discussed above, it also leads to many references between articles that – from a knowledge representation perspective – are superfluous. These are references that expand on terms used to describe the topic of an article, but that do not expand on the topic itself. We call these references *meta references*.

Example 6.1.10 (Meta Reference). *For example, an article about the “United States” (of America) states that its de facto national language is English, with references to “de facto”, “National Language”, and “English”. While “English” is relevant to the topic, both “National Language” and “de facto” contain meta information that is required to understand the assertions about the United States, but that have no relevance for the content of the topic. Both references are meta references.*

Our forth and final step in preparing Wikipedia for knowledge extraction is to remove these meta references. Our approach is to calculate a weight for every reference relationship that

indicates how strongly two articles are indirectly connected even without the direct reference. This is based on the assumption that two topics that are semantically related are also strongly connected in the Wikipedia graph structure. If this weight is above a certain threshold, then the reference is retained; otherwise it is removed from r_l .

There are several approaches to calculating weights for references between Wikipedia articles in the literature. [MWM08, MW08, GHP10] define weights based on how strongly the categories that the articles belong to are connected. [VTP08] define weights based on the number of common categories, i.e., categories that both articles belong to. The score function s_3 that uses a lexical database to determine the semantic similarity can also be used to calculate weights for existing references [SPF10].

In [Wil11], several such approaches were analysed for the plausibility of their results, their efficiency, and their applicability. Using the semantic similarity score depends on the quality and availability of the lexical database, which differs for various languages. It is also not applicable to short topic names. Additionally, it can only be a positive indicator, but not a negative one: two topic names that do not contain any common or related words may well be related anyway. The best overall results are obtained by combining two of the connection-based approaches.

There are, however, some cases where these approaches are either ineffective, or require very careful adjustment of the threshold value, because some meta references are very common across a large number of articles with similar topics. For example, in the Wikipedia as of 2009-11-30, most articles about a city contained the city's size in square kilometres, with a meta reference to the article on "Metre". Only the weighting function defined in [VTP08] gives these references a low score (the approach using lexical databases is not applicable because the article name "Metre" is too short).

Our assumption that semantic relation correlates with strong connection in the graph can be substantiated for most articles and is indeed made – even if often only implicitly – in the literature. Yet it is impossible to entirely rule out cases where this assumption does not hold. Combining multiple weighting functions can mitigate this effect.

In general, all of the approaches for extending and preparing Wikipedia can introduce new errors, because they rely on data that may already contain errors, and because they can only heuristically try to recognise the intentions of the authors. None of the approaches can offer more than a best guess on why a certain reference or category membership was defined.

Evaluation and Results

Evaluating Wikipedia-related approaches is generally challenging due to the immense volume of data involved, but especially so for any qualitative evaluation. A quantitative evaluation is mainly challenging in terms of time and space requirements.

All time measurements in this section were performed on a computer with an Intel Core2 2.1 GHz CPU and 2 GB of working memory. The operating system was Windows 7 (x64), with Java 7 installed. Microsoft SQL Server 2008 R2 was used as a database server.

For Wikipedia as of 2009-11-30, the XML data dumps for categories and articles available at <http://dumps.wikimedia.org/>³ require 25.5 GB and 6.7 GB harddrive space alone for the English and German versions, respectively. They contain 528,128 and 75,454 categories, and 1,022,255 and 130,127 sub-category relationships, respectively.

In the first step, ϕ_{trans} , we identified 146,912, or 14 %, additional sub-category relationships for the English Wikipedia. This took 178 seconds. For the German Wikipedia, we identified 59,514, or 46 %, new sub-category relationships in 111 seconds. The large difference in relative

³visited 05/2013

effectiveness (14 % vs. 46 %) is caused by the relatively small number of translations available for the English Wikipedia versus the relatively large number for the German Wikipedia. A similar distribution was also apparent in table 6.1 for the Wikipedia as of 2012-05-30.

The second step, ϕ_{lex} , yielded 144,318, or another 14 %, new sub-category relationships for the English Wikipedia, but only 3,723, or 3 %, for the German Wikipedia. This can be attributed directly to the different lexical databases employed: the Wordnet database for English, which is both very large and of high quality, and an early version of the OpenThesaurus⁴, which in 2009 was still very small. The flip side is that finding the new relationships took over 15 hours for the English language, including about 5 minutes for creating an inverted index, but less than a minute for the German language.

During step three (ϕ_{remC}), 22,702, or 4.3 %, and 1,278, or 1.7 %, categories were removed from the English and German Wikipedia, respectively. The smaller percentage for the German Wikipedia reflects the smaller number of sorted aggregation pages there.

Due to the huge time and main-memory requirements for step four (ϕ_{remR}), we were unable to complete measurements for the full Wikipedia. We did, however, succeed in making tests for a subset of Wikipedia, as described in section 6.1.2.

We considered two possibilities for measuring the precision of our approach, i.e., the number of new relationships we identified in steps one and two that would not hold up to closer scrutiny. Formally, we define the precision of an approach a as $p(a) = \frac{|s_i| - |s_i^{\text{err}}|}{|s_i|}$, where s_i^{err} are those new sub-category relationships that are erroneous.

The first possibility is to actually write the new relationships into the live Wikipedia, and see how many have been removed by the community after a certain time. This should give an indication of how many of these relationships do not have merit. However, this approach is clearly very problematic from an ethical point of view, so we refrained from using it.

The second possibility is to extract a relatively small random sample from the newly generated data and have it manually evaluated by volunteers. To this end, we found four independent users. Each of them was given a different random sample of 100 sub-category relationships, with the task to rate their plausibility on a scale of 1 (very plausible) to 4 (completely implausible). We used an even number of choices to discourage “I-am-not-sure” choices and to force the users to decide in favour or against a specific relationship.

Calculating the sum of all 1 and 2 choices, and normalising it by the sample size yielded a precision value in the range $[0, 1]$, as desired. We then calculated the average precision value ϕp for all users. For the English Wikipedia, the results were $\phi p(\phi_{\text{trans}}) = 0.96$ and $\phi p(\phi_{\text{lex}}) = 0.99$, with similar values $\phi p(\phi_{\text{trans}}) = 0.96$ and $\phi p(\phi_{\text{lex}}) = 0.98$ for the German Wikipedia.

For information extraction approaches, these values are very encouraging, especially considering that knowledge representation is very often a matter of opinion. Additionally, errors and inconsistencies between language versions in Wikipedia may contribute to the slightly lower values for $\phi p(\phi_{\text{trans}})$.

Unfortunately, there is no “perfect” and “complete” version of Wikipedia (even if such a thing could exist) against which we could compare our results against. Therefore, calculating an absolute recall value, i.e., the number of new relationships that were missed by steps one and two, is impossible.

A possible way to at least get an indication of the recall is to apply our approach to an old data set, and then compare it with a more recent version. This assumes, however, that a new version is automatically a “better”, i.e., more correct and complete, version. While this does not have to be the case, we also realised that there may be large structural changes in Wikipedia

⁴<http://www.openthesaurus.de>, visited 05/2013

over time that do not constitute actual “corrections”, but merely a different way to represent the data.

For example, the category structure around “United States Law”, as described in example 6.1.4 for 2009-11-30, does not exist in this form any more in the Wikipedia as of 2012-05-30. This greatly decreases the value of such a comparison. However, a total increase of 28 % and 49 % for English and German, respectively, is a strong indication that our approach has merit, even if it would miss some relationships.

We decided against a formal evaluation of step three (ϕ_{remC}), because its results are too subjective. In various discussions we found vastly diverging opinions on what constitutes superfluous data and what does not. We will therefore leave it in the hands of knowledge engineers to decide on a case-by-case basis whether or not to employ this step.

Finally, we tested the suitability of Wordnet as a lexical database. We used a list of word-pairs with a manually judged degree of how closely they are related, as a point of reference. This list was created in [RG65], where a number of students were paid to manually judge the degree of relatedness for each word pair. This score was then averaged over the number of students. The original score, called “judged synonymy”, was in the range of $[0, 4]$, which we reduced to $[0, 1]$ (called “adjusted judged synonymy”, or AJS) as shown in table 6.3.

Because the score s_3 is obtained in a completely different way, we could not make a direct comparison between the values for s_3 and AJS. Instead, we sorted the sample set twice, once according to the order imposed by s_3 , and once according to the order imposed by AJS. We then compared the relative order of samples for each sorting, noting where s_3 put a sample in a different position than AJS. We disregarded the relative order of samples with identical values.

First, we calculated the score s_3 for each of the 65 word pairs given in table 6.3, resulting in nine word pairs with a score of 0.8, eleven word pairs with a score of 0.5, and 45 pairs with a score of 0. Out of the top nine, there are three pairs that were ranked lower by AJS; four pairs out of the middle tier were ranked either lower or higher by AJS; and three pairs out of the 0-tier were ranked higher by AJS. Thus, ten word pairs in total were ranked differently between using the Wordnet-based score and the “judged synonymy” based score.

While most of these rankings were not too different, there were a few notable exceptions. For example the pair (“serf”, “slave”) has an AJS of 0.865, but scored 0 for s_3 . In Wordnet, the word “serf” is related to “thrall”, which in turn is related to “bond servant”, which is finally related to “slave”. This indirect relationship is stretched too far to be automatically recognisable.

Additionally, Wordnet does not provide a weight for relationships, only a discrete yes/no value. This is apparent for the pair (“boy”, “lad”). According to Wordnet, they are not synonyms, but related, resulting in an s_3 score of 0.5. However, while they may not be exactly synonymous, they are very close to it, which is reflected by an AJS value of 0.955. This degree of precision cannot be captured by Wordnet.

Now, we extended our sample base to see how longer terms would influence the results. We generated a random sample of 64 term pairs, each term consisting of four words. We then calculated both s_3 and a normalised AJS for each pair. For example, for the pair (“cushion bird hill gem”, “jewel woodland mound jewel”), we calculated an s_3 score of $\frac{0+0+0.5+0.8}{4} = 0.325$, and a normalised AJS of $\frac{0.1125+0.31+0.8225+0.985}{4} = 0.5575$.

The longer term length increased the effect observed for the first sample set: 39 out of 64 samples were in a different order. This increase is caused by the larger number of different values for the s_3 score: when only two words were compared, 0, 0.5, and 0.8 were the only possible values for s_3 . For two terms of four words each, the possible number of values is much larger. Therefore, when comparing the order of the samples sorted by s_3 and AJS, respectively, the number of samples with the same position is smaller. For instance, with only two words, a large

number of samples had an s_3 score of 0. These samples all held the same relative position, greatly decreasing the possibilities for inconsistencies with the order defined by AJS.

These results show that there are obviously differences of opinion how linguistic relationships are to be weighted, or at least how they are to be represented. This also leads to differences in how Wikipedia relationships should be weighted, making the selection of a lexical database an important step that has to be undertaken carefully.

6.1.2 Harvesting Wikipedia

There have been a number of attempts to gather knowledge from Wikipedia.

[HSB07] argue that Wikipedia articles and categories have names that are unambiguous in the domain of Wikipedia and can thus be used as entities in an ontology. They further argue that these names can be understood intuitively by humans, which makes them even more useful for knowledge representation purposes.

The “Semantic WikiMedia” is an extension of the MediaWiki framework proposed by [KVV⁺07] that attempts to represent semantic knowledge explicitly in the data. It lets authors add semantic information to wiki pages, enriching links and text fragments by annotating semantic data. The drawback of this approach is that content authors need to handle the semantic representation manually, which makes authoring harder, more time consuming, and introduces new possibilities for human error. Also, the amount of semantic knowledge that is actually available is currently rather small.

In Wikipedia, there are templates for representing content of certain types, such as default data points for cities, plants, or people. [AL07, BLK⁺09] utilise this standardised encoding of information to extract data tuples. This data can be used to augment an existing ontology.

Similarly, [WW08] creates a knowledge base from Wikipedia articles based on templates, using default data points as properties. Similar attributes, Google query results, and Wordnet relationships are used to infer additional relationships between articles.

[SKW07] proposes YAGO, an OWL-based ontology that uses both Wikipedia and Wordnet as sources, representing Wikipedia categories and Wordnet synsets as classes, Wikipedia articles as objects, and category and synset relationships as properties between classes. This representation makes it incompatible with standard description logics (cf. section 5.3). While it derives additional properties for objects from articles, it does not attempt to harmonise the sets of classes derived from Wikipedia and Wordnet, respectively, leaving them entirely separate.

[NS08] posit that new category relationships are encoded in category names. They use grammatical rules of the English language to analyse these names and extract “is-a”, “narrower-than”, and “related” relationships with good reliability. This process can be used to complement our own approach.

[dMW10] attempt to extract a taxonomy from Wikipedia. Their focus is on creating a multi-lingual taxonomy, rather than using different languages to improve a single language knowledge base. Similar to YAGO, they try to integrate Wordnet into their results.

Mapping the Wikipedia structure onto an ontology structure is – at first glance – straightforward: categories are mapped onto concepts, articles onto individuals, sub-category relationships onto sub-concept relationships, membership relationships onto concept assertions, and reference relationships onto role assertions.

While such a simple mapping will suffice for large parts of Wikipedia, it is unsuitable in general. The sub-category relation implies a similar semantics to that of the sub-concept relation, but in reality this semantics is neither formally defined nor is it consistently applied. This observation has been substantiated by [PS07].

Word A	Word B	AJS	Word A	Word B	AJS
cord	smile	0.0050	hill	woodland	0.3700
rooster	voyage	0.0100	car	journey	0.3875
noon	string	0.0100	cemetery	mound	0.4225
fruit	furnace	0.0125	glass	jewel	0.4450
autograph	shore	0.0150	magician	oracle	0.4550
automobile	wizard	0.0275	crane	implement	0.5925
mound	stove	0.0350	brother	lad	0.6025
grin	implement	0.0450	sage	wizard	0.6150
asylum	fruit	0.0475	oracle	sage	0.6525
asylum	monk	0.0975	bird	crane	0.6575
graveyard	madhouse	0.1050	bird	cock	0.6575
glass	magician	0.1100	food	fruit	0.6725
boy	rooster	0.1100	brother	monk	0.6850
cushion	jewel	0.1125	asylum	madhouse	0.7600
monk	slave	0.1425	furnace	stove	0.7775
asylum	cemetery	0.1975	magician	wizard	0.8025
coast	forest	0.2125	hill	mound	0.8225
grin	lad	0.2200	cord	string	0.8525
shore	woodland	0.2250	glass	tumbler	0.8625
monk	oracle	0.2275	grin	smile	0.8650
boy	sage	0.2400	serf	slave	0.8650
automobile	cushion	0.2425	journey	voyage	0.8950
mound	shore	0.2425	autograph	signature	0.8975
lad	wizard	0.2475	coast	shore	0.9000
forest	graveyard	0.2500	forest	woodland	0.9125
food	rooster	0.2725	implement	tool	0.9150
cemetery	woodland	0.2950	cock	rooster	0.9200
shore	voyage	0.3050	boy	lad	0.9550
bird	woodland	0.3100	cushion	pillow	0.9600
coast	hill	0.3150	cemetery	graveyard	0.9700
furnace	implement	0.3425	automobile	car	0.9800
crane	rooster	0.3525	midday	noon	0.9850
			gem	jewel	0.9850

Table 6.3: Word pairs with their respective Adjusted Judged Synonymy (AJS), after [RG65].

Example 6.1.11 (Wikipedia Relation Semantics). *Regard, for example, the category “Crusades”, which as of 2012-05-30 is a sub-category of “History of Europe”. This, in turn, is a sub-category of “Europe”, which is a sub-category of “Continents”. Applying a sub-concept semantics, this would make every crusade into a specialisation of a continent, which is clearly inconsistent with the intended semantics of the knowledge represented in these categories.*

Additionally, mapping membership relationships onto concept assertions would make the “Holy Lance”, an article in the “Crusades” category, into a crusade and – combined with the sub-concept semantics, into a continent.

Instead, we map the Wikipedia structure onto an ontology with less stringent semantics.

Definition 6.1.12 (Wikipedia Ontology). *Let $\mathcal{O}_W = (C_W, R_W, I_W, X_W)$ be an ontology with*

$$\begin{aligned} C_W &= \{\text{Term}\} \text{ and} \\ R_W &= \{\text{hasNarrower}, \text{hasMember}, \text{hasReference}\}. \end{aligned}$$

Every category name $n_l(c)$ from Wikipedia is mapped onto an individual $i_c \in I_W$, and every article name $n_l(a)$ is mapped onto an individual $i_a \in I_W$. For every individual $i \in I_W$, a concept assertion $\text{Term}(i)$ is added to X_W .

Sub-category relationships $s_l(c_1, c_2)$ are mapped onto role assertions $\text{hasNarrower}(i_{c_2}, i_{c_1})$, membership relationships $m_l(a, c)$ are mapped onto role assertions $\text{hasMember}(i_c, i_a)$, and reference relationships $r_l(a_1, a_2)$ are mapped onto role assertions $\text{hasReference}(i_{a_1}, i_{a_2})$.

Then \mathcal{O}_W is a Wikipedia ontology. It can be implemented, for example, using the SKOS model.

Example 6.1.13 (Wikipedia Ontology). *Recall the extended Wikipedia categories from figure 6.1 (b). Then $\mathcal{O}_{6.1(b)} = (C, R, I, X)$ is a Wikipedia ontology, with*

$$\begin{aligned} C &= \{\text{Term}\}, \\ R &= \{\text{hasNarrower}, \text{hasMember}, \text{hasReference}\}, \\ I &= \{\text{United States Law}, \text{Prisons in the United States}, \dots\}, \text{ and} \\ X &= \{\text{Term}(\text{United States Law}), \\ &\quad \text{Term}(\text{Prisons in the United States}), \\ &\quad \dots, \\ &\quad \text{hasNarrower}(\text{United States Law}, \text{Prisons in the United States}), \\ &\quad \dots\}. \end{aligned}$$

While a Wikipedia ontology allows for fewer assertions to be inferred from the knowledge base than the straightforward mapping mentioned above, it is still sufficient to serve as a keyword taxonomy $\mathcal{X}_K = (C_K, R_K, I_K, X_K)$ (cf. section 5.1) or as a terminological ontology $\mathcal{O}_T = (C_T, R_T, T, X_T)$ (cf. section 4.2.1).

However, a full Wikipedia ontology is far too large to be used in practice. We therefore extract a “core” of knowledge, centred on a specific topic. Starting from the article representing this topic, we gather all articles and categories that are in a direct or indirect relationship with the article. Every direct relationship must have a weight above a minimal threshold t_{min} to ensure that it is relevant. Every indirect relationship must have an aggregated weight over all its components that is below a maximal threshold t_{max} to ensure that it is not too far removed.

Specifically, we use the centre article as the initial “core”, and then successively select all articles that are referenced by an article in the core and that also share at least one category with an article from the core. This is repeated up to a specific distance from the centre element.

Formally, we use a weight function w that assigns a value of 1 to two articles a_1, a_n if $(a_1, a_n) \in r_l$, a_1 is in the core, and a_n shares a category with an article in the core. It assigns a value of $\sum_{i=1}^{n-1} w(a_i, a_{i+1})$ if there is a shortest sequence of articles (a_2, \dots, a_{n-1}) with $(a_i, a_j) \in s_l$ for $1 \leq i < j \leq n$. Otherwise, the function assigns a value of 0.

<i>l</i>	Centre article	$t_{max} = 2$			$t_{max} = 3$		
		Time	Art.	Cat.	Time	Art.	Cat.
pih	“List of Countries”	0.6 s	14	33	2.8 s	70	343
sco	“Fortune 500”	1.7 s	1	6	7.1 s	111	752
sco	“List of Countries”	1.2 s	2	4	38.5 s	799	6,536
en	“Ballroom Dance”	340.3 s	7	40			
en	“Data Structure”	195.3 s	14	401			
en	“Computer Science”	2,661.8 s	104	8,498			
en	“Germany”	9,602.4 s	282	26,578			

Table 6.4: Statistics for Wikipedia cores, as of 2012-05-30.

Empirically, we have obtained useful results for a minimal threshold $t_{min} = 1$ and a maximal threshold $t_{max} \in [2, 3]$. This creates a core of relevant articles that are at most three “hops” removed from the centre. However, there is still a need for manual corrections, leaving room for optimisations. Possible approaches are additional weight functions as cited in section 6.1.1, step four, and a combination with methods cited at the beginning of this section.

Table 6.4 shows extraction times and sizes of Wikipedia cores for several centre articles in three languages. As of 2012-05-30, the Norfolk (pih) version of Wikipedia has 145 categories and 1515 articles, and the Scots (sco) version has 13,659 categories and 89,917 articles, making them good candidates for evaluation purposes. We chose English (en) as the third language, because it is the largest Wikipedia to date.

As centre articles for Norfolk and Scots, we used articles with a high number of references from other articles, i.e., with a high indegree. These articles are in both instances the “List of Countries”, “Lyst o’ Kuntrii” in Norfolk with an indegree of 447, and “Leet o Kintras” in Scots with an indegree of 390. In Scots, we also used the “Fortune 500” article, which has an indegree of 748. For the English Wikipedia, we used four articles that are each thematically central to their topic: “Ballroom Dance”, “Data Structure”, “Computer Science”, and “Germany”.

As can be seen in table 6.4, extraction times steeply increase with the number of articles (columns 4 and 7) and categories (columns 5 and 8) in the core. However, the increase is not proportional, because the extraction time depends on the total number of references that need to be investigated, not on the number of references that are actually relevant.

The approaches presented here can, in principle, also be applied to other wikis. There exist wikis that may be more suited for use as background knowledge, because they cover domain specific topics, such as corporate wikis that are used to aggregate and archive corporate knowledge. Other examples are Diplopedia, a wiki on international relations maintained by the United States Department of State, or Scholarpedia with a focus on scientific topics.

6.1.3 Practical Considerations

For reasons of efficiency, we store the extracted Wikipedia in a relational database system. An EER diagram of the database schema is shown in figure 6.2. The disjoint entity types “Article” and “Category” have a common super-type with three attributes “id” (primary key), “name” (n_l) and “language” (l). Translation relationships “translates” are defined on both “Article” ($t_{(l_1, l_2)}^a$) and “Category” ($t_{(l_1, l_2)}^c$) to keep them separate. References can connect entities of either type, so the appropriate “references” relationship (s_l, m_l, r_l) is defined on their common super-type.

The relevant database tables and index structures of the implemented database schema are

shown in table 6.5. There are tables for storing the names of articles and categories, and a table for storing translation relationships between articles or categories. There is also one table for storing references between articles and categories, where references between categories are interpreted as sub-category relationships, and references from categories to articles are interpreted as membership relationships. The Boolean attributes `src_cat` and `trgt_cat` indicate whether the source or the target, respectively, is a category or an article.

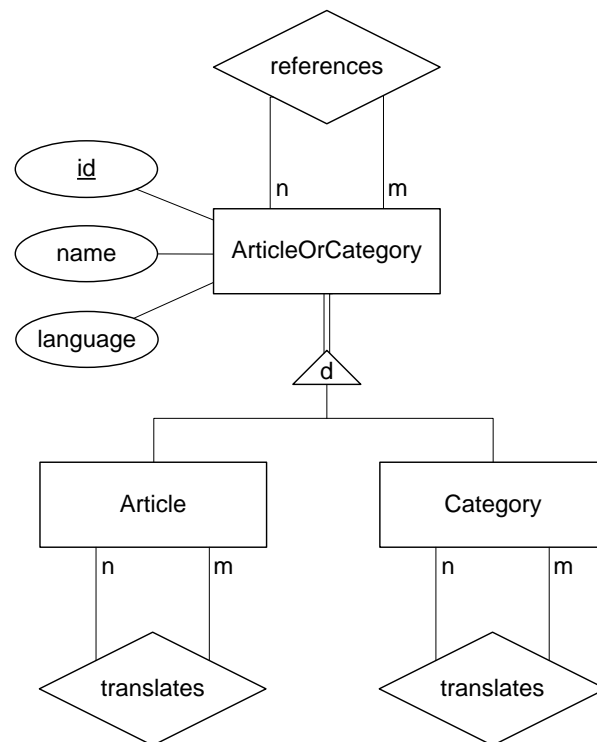


Figure 6.2: Database schema for Wikipedia

This closely mimics the design of Wikipedia, which does not differentiate between different types of references, only interpreting them differently based on the context. In Wikipedia, even translation relationships are not specified differently than other references. Instead, the wiki framework detects cross-language references and interprets them as translations. We moved translations into a separate table to enhance clarity. Experiments with a further partitioning of the data into additional tables, such as a partitioning by language, showed no performance improvements, neither for insertion nor retrieval.

This schema design also allows us to slightly relax the primary key property of the `id` attribute: according to the EER schema, a value for `id` must be unique across *both* the `articles` and `categories` tables. However, using the `*_cat` attributes in `references` allows us to restrict this requirement to each table, thus changing the common primary key attribute of `articles` and `categories` into two primary key attributes of the two respective tables. While this is forbidden in the EER model, it is possible in the implementation.

Table 6.6 lists the times required for the different steps of extracting the Wikipedia structure.

Table name	Attributes	Indices
articles	id, lang, name	primary key (id), unique (lang, name)
categories	id, lang, name	primary key (id), unique (lang, name)
references	src, trgt, src_cat, trgt_cat	indices (src, src_cat), (trgt, trgt_cat)
translations	src, trgt, src_cat, trgt_art	indices (src, src_cat), (trgt, trgt_cat)

Table 6.5: Database indices for Wikipedia

l	XML preprocessing	Reading n_l	Reading $t_{(l,*)}^a, s_l, m_l, r_l$	Σ
en	4,685 s	37,509 s	164,627 s	57.5 h
de	1,357 s	9,560 s	51,598 s	17.4 h
nl	530 s	4,558 s	33,122 s	10.6 h
fr	1,321 s	10,283 s	59,246 s	19.7 h
it	836 s	6,753 s	43,776 s	14.3 h
pl	617 s	4,818 s	32,900 s	10.6 h
es	911 s	7,671 s	44,064 s	14.6 h
ru	1,095 s	7,102 s	38,198 s	12.9 h
pt	558 s	4,083 s	29,664 s	9.5 h
Σ	11,910 s	92,335 s	497,196 s	167 h

Table 6.6: Time measurements for the Wikipedia extraction, as of 2012-05-30.

The first step (XML preprocessing) removes irrelevant content from the raw Wikipedia XML files, such as translations into languages that are not part of the extraction, and transforms the wiki syntax for references into more easily parsable XML code.

The second step reads all article and category names and populates the table **articles** and **categories**, creating unique IDs for each article and category in the process. These IDs abstract from multiple ways of writing a name (e.g., “Ballroom Dance” vs. “Ballroom dance”), which are treated separately in Wikipedia. Treating them separately greatly increases the complexity of the structure, because it requires special redirection references between articles with different versions of the same name. It also holds a large potential for errors, for example when references point to the wrong name version of an article, or when redirection references are missing. At the same time, it offers little additional information, because special disambiguation pages are used whenever different capitalisation in a name implies a different meaning.

The third and final step reads all references, populating the **references** and **translations** tables. Creating the indices took an additional 12,889 seconds, or about 3.5 hours. In total, almost seven days were required to extract the entire Wikipedia structure in nine languages.

Several challenges had to be met during the extraction. For example, Wikipedia categories are only identified and differentiated from articles by a keyword that is prefixed to their names. This keyword, however, is different for each language: “Category” in English, “Kategorie” in German, or “Категория” in Russian. Not only must these keywords be identified and used throughout, it is also important that Unicode characters are supported in every step of the extraction process.

When article and category names are inserted into the database in step two, no duplicate entries must occur. There are several possible solutions for this on the database side:

1. create a unique index on the name and language attributes,
2. use an `IF NOT EXISTS ... INSERT` statement to insert the data,

3. combine solutions 1 and 2,
4. use a MySQL specific `INSERT IGNORE` statement to insert the data, or
5. combine solutions 1 and 4.

The actual `INSERT` statements used for solutions 2 and 4, respectively, are shown for articles in listing 6.1.

```

1 IF NOT EXISTS (SELECT * FROM articles WHERE lang=? and name=?) INSERT INTO
  articles (lang, name) VALUES (?, ?)
2 INSERT IGNORE INTO articles (lang, name) VALUES (?, ?)

```

Listing 6.1: `INSERT` statements for Wikipedia articles

Using MySQL 5.5 proved to be too slow for our purpose: for a sample dataset with 31 articles and 18 categories, solution 1 took 11,333 ms. The MySQL-specific solution 4 took even longer (11,575 ms), and their combination (solution 5) took the longest with 12,015 ms. All times were measured five times, with the mean of the last three results taken as the final result time.

Solution 1 with Microsoft SQL Server 2008 only took 504 ms. Solution 2 took 1,393 ms, and solution 3 was executed the fastest, with 502 ms. To be able to better differentiate the results of only using a unique index and of combining a unique index with an `IF NOT EXISTS ... INSERT` statement (solutions 1 and 3), we repeated the test for a larger dataset, namely Wikipedia in the three languages Ænglisc, Norfolk, and Scots, with a total of 121,839 articles and 15,362 categories. Here, the combination of solutions 1 and 2 clearly won out, with a total time of 217,789 ms over 281,986 ms for solution 1.

Based on these results, we used a combination of a unique index and an `IF NOT EXISTS ... INSERT` statement with an MS SQL Server for the actual extraction process.

Conclusion

In this section, we have contributed several novel optimisations for semantically exploiting Wikipedia, which we have also implemented and evaluated successfully. Options for optimising efficiency have been discussed and assessed, as well as the suitability of Wordnet as the most prominent lexical database available.

6.2 Obtaining Knowledge from other Sources

Other sources for background knowledge are existing ontologies like the Cyc knowledge base of everyday knowledge [WBC⁺03], the Library of Congress Subjects Headings [LCS09], the LKIF ontology of basic legal concepts [HBBB07], or the OpenGALEN ontology for medical terms and procedures [RRZvdH03].

However, for many domains, no ontologies are available. Yet often there are sources of semi-formalised knowledge available, for example in the form of glossaries or listings. Using the ITIL (IT Infrastructure Library)⁵ terminology as a show case to demonstrate feasibility and effectiveness, we will discuss a five-step procedure of how a formalised knowledge base can be obtained from such sources.

⁵“ITIL” and “IT Infrastructure Library” are registered trademarks of the government of the United Kingdom

Obtaining Knowledge from Semi-Formalised Public Sources

The first step is to determine the semantics inherent in the source data: what kinds of concepts are covered by the data, what relations are defined, and what are the semantics of these relations? This step should preferably be done or supported by a domain expert. This step also determines if knowledge extraction is worth the effort, or if it is even possible at all.

Next, the detected semantics need to be formalised: ontology concepts and roles need to be defined that can represent the data, and axioms need to be specified that define their semantics.

The third step is to identify keywords, formatting styles, and other indicators for relevant data and semantic relations.

Then, using these indicators, the source data is parsed and individuals, as well as concept and role assertions are interpreted from the data and asserted in the resulting ontology.

Finally, the obtained knowledge base should be evaluated. A manual comparison against the source data based on a random sample of the ontology can be used to judge the extraction quality. A comparison with an existing knowledge base covering a related topic can be used to ascertain the quality of the source data, provided that the extraction has both a high precision and recall.

Example 6.2.1 (Extracting an ITIL Ontology from a Glossary). *The ITIL terminology is available online at <http://www.itil.org/en/glossar/glossarkomplett.php>⁶. It contains a collection of terms that describe ITIL concepts, each with a description. This description may contain textual references to more general concepts, to related concepts, and to equivalent concepts (or, more precisely, to synonymous terms). We therefore need to represent instances of concepts, and three different relations.*

Instead of defining a new schema, we will use the SKOS model as a basis for the formalisation. In particular, we will represent ITIL concepts as `skos:Concepts`, and concept generalisation, relatedness, and equivalence as `skos:broadMatch`, `skos:relatedMatch`, and `skos:exactMatch`, respectively. In addition, concept names will be specified using `skos:prefLabel`, and the concept description will be attached to each concept instance with `skos:note`.

We identified a formatting style for terms, and two keywords and one writing convention for relations. Terms, or concept names, are always formatted in bold. Related concepts are textually referenced in the last sentence of a description following the keyword “see”. Equivalent concepts are textually referenced in the description following the keywords “synonym for”. Generalisations are written in parentheses at the beginning of a description.

A short (less than 200 lines of code) Java program facilitates the actual extraction process based on these indicators and writes the obtained ontology to a file in Turtle syntax.

A manual matching of 50 randomly selected concepts (including all of their outgoing relationships) from the ontology against the source data detected no errors, and the number of concepts defined in the source matches the number of concepts in the ontology, which indicates both high precision and recall values. Since the source is an accepted domain authority, we will regard the ontology as normative, rather than attempting to verify it against less authoritative ontologies like the one obtained from Wikipedia.

The source glossary is available in two languages, so a simple extension of the keywords allowed us to obtain a multi-lingual ontology. This ontology does not, however, contain any translation relationships, which arguably makes it more into two parallel ontologies in different languages on the same topic. In a postprocessing step it might be possible to obtain such translations automatically from Wikipedia, from Wiktionary, or from Google Translate, but we did not attempt this.

⁶visited 05/2013

In total, the dual-language ontology contains 1,089 concepts (544 in English, 545 in German), 794 generalisations, 304 relatedness relationships, and 52 equivalences. The entire extraction process, including data transmissions and writing the ontology file, took 1.9 seconds.

Other approaches, such as Cyc, focus on domain experts to create and formalise ontologies. While the quality of a manually and expertly created knowledge base is – at least for now – clearly above that of automatically harvested ones [WBC⁺03], these approaches are not only very expensive in terms of manpower, but they also ignore the knowledge and resources that are already available.

Obtaining Knowledge from Documents

A different source of background knowledge can be found in the very documents for which this knowledge is required. They are by definition relevant for the domain, and they often contain a wealth of terminology. The general methodology is the same as for other sources of knowledge, as described above. However, for these documents the methods described in chapter 5 have usually already been implemented, so they are available for tasks like extracting headlines or topics. This makes the implementation of step four easier, and possibly more effective.

Unfortunately, most documents are written to be actually *read*, and not to just look up some data points. Moreover, they are written for *human* readers. As a consequence, many documents are written with the human process of understanding in mind, i.e., they are structured based more on didactic requirements than on semantic coherence. This is aggravated by consideration for human “foibles” such as aesthetics, and makes inference from where terms appear in a document’s structure to relationships between these terms unrealistic in general. For example, topics that are discussed on the level of chapters are not necessarily more general than terms discussed on the level of sub-sections. Further research in this area might uncover less obvious bases for inference, but for the time being we will be content with extracting unstructured terminologies, i.e., lists of terms, from documents.

Additionally, authors expect a reader’s human-level ability for abstraction and for classification. For example, a human reader is usually able to differentiate between the different uses of parentheses in “Example: XML (continued)” and “Example: XML (OWL)”. In the first case, the parentheses contain meta-information about the example, namely that it is the continuation of an earlier example. In the latter case, the parentheses contain a clarification. The latter information is part of the example’s topic, the former is not. It might be possible to capture some of these ambiguities using text analysis techniques, but that is beyond the scope of this work.

While it is possible to isolate a number of special cases and treat them specifically, this is not feasible in general. Cross-checking with existing terminologies can also help raise the quality of the extracted terms and topics. In particular, validating the domain specific terminology obtained from a document against a large but general purpose terminology like the one obtained from Wikipedia may improve the precision of the extracted data. This comes, however, at the cost of a lower recall, because many specialised terms are not part of Wikipedia (or some other general terminological corpus) and are thus stricken from the extracted terms (see below).

Yet even background knowledge of low quality may be useful, or at least better than having no background knowledge at all. In particular, it can be instrumental in detecting occurrences of terms that are not in any way marked in the document, i.e., a relevant term that is simply used in a text section. Knowledge that this particular text section deals with a certain topic indicated by that term may be relevant later, so catching the term is important.

On the other hand, detecting large numbers of irrelevant terms and integrating them into a document model quickly deteriorates the overall quality of said model. Yet if the background

knowledge contains many erroneous or irrelevant terms, this is exactly what may happen. It is therefore advisable to put the focus on the precision of the background knowledge, even to the detriment of the recall, which brings us back to validating the extracted terminology against a large but not domain-specific corpus.

An important question is where such background knowledge may actually be used – in particular, if it may be used in extracting the data models or in the semantic modelling of the same documents from which it was obtained, or if the two corpora of documents must be kept strictly separate. In principle, such a separation is clearly desirable to avoid a propagation of errors. An irrelevant term that has been extracted from a document d as part of the background knowledge b_d would directly become part of the semantic document model for d , if b_d were used in extracting and modelling d . Other documents might not contain this irrelevant term, limiting the harm done by the imprecise background knowledge if it were only applied to documents other than d .

If such background knowledge is validated as discussed above before using it, while there is still a need for caution, there does not seem to be a compelling reason to force a complete separation between the document corpora. This assumption could be substantiated in tests conducted in a laboratory environment, but these result largely depend on the quality and size of the terminology used for validation.

Example 6.2.2 (Background Knowledge Obtained from E-Learning Documents). *From two corpora of e-learning documents, which we will simply call “Passau” and “Rostock” (see chapter 10), we attempted to extract background knowledge terminologies. Since there already exist transformation rules for extracting semantic document models from these documents, we used these rules for obtaining the terminology by first extracting semantic document models for each document, and then aggregating the topics, headlines and terms in these models. The knowledge base schema for a simple word list is rather straightforward and consists of a single concept Term.*

Extracting the semantic document models took 359 seconds for the “Passau” corpus (1,747 s for the “Rostock” corpus), and aggregating the background knowledge from these models took another 20 seconds (“Rostock”: 58 s). The resulting terminologies contained 2,265 and 13,433 terms for “Passau” and “Rostock”, respectively.

Validating these terms against the Wikipedia terminology reduced the numbers to 307 and 2,237 terms, or about 14 and 17 percent of the original list, respectively. Assuming idealised Wikipedia data that is completely devoid of errors, this provides us with a reasonably large domain-specific terminology of high quality.

Conclusion

In this section, a methodology for obtaining formalised knowledge from semi-formalised sources was discussed. We have successfully applied the methodology to both publicly available data and to corpora of documents from a specific domain that are not publicly available. This increases the availability and attainability of domain specific background knowledge, thus enhancing the effectiveness of approaches developed in chapters 4 and 5. We have also discussed the quality of the newly obtained background knowledge and the ramifications of errors in the knowledge base.

Part III

Implementation

Chapter 7

System Architecture

In this chapter, the new software framework that implements the notions introduced in part II will be described. The actual implementation is part of a larger framework that was developed for the VERDIKT research project [FWJS08]. We will limit our discussion here to the components that were created as part of this thesis.

7.1 System Overview

In this section, we will provide an overview of the system architecture, how document models are implemented, how background knowledge is implemented, and which interfaces are relevant for document processing.

Figure 7.1 shows the data flow of the system at a high level of abstraction. A digital document is run through a preprocessing chain to obtain a representation of a connected base document model (cf. definition 4.1.29 on page 75). Through a semantic processing step that uses background knowledge, a representation of a semantic document model (cf. definition 4.2.14 on page 82) is obtained, which is then run through a postprocessing chain to obtain the desired final result, for example a temporal model or a taxonomy. The primary postprocessing step (cf. section 9.3) can be specified using a graphical tool [Sch10].

The package structure of the implementation is shown in figure 7.2. `de.uni_passau.verdikt` is the base for all relevant packages. The `DocumentModel` sub-package contains the interfaces used to represent semantic document models. The actual, Jena-based implementation is located elsewhere. `DocumentModel` has two sub-packages: `adapter` and `BackgroundKnowledge`. The latter contains the interfaces and classes used to represent background knowledge, while the former contains the interfaces used for processing the document models. Their various implementations are located in `adapter's implementation` sub-package, which in turn contains the `VerificationModel` package that holds classes and interfaces required for specifying how a verification model is to be obtained from a semantic document model.

We will use XML documents (namely instances of `org.w3c.dom.Document`) to represent base document models. As discussed in section 4.2.1, this is possible because each XML element can be seen as a separate media object, with its successor defined by the document order. This is, however, a very cumbersome way to represent XML documents. We will later see more convenient methods for specific XML formats. Metadata associated with a base document model can be represented as XML attributes.

XML is a convenient format for this purpose because not only is it possible to represent almost any other document format in XML, but the conversion is often simple (in the sense of

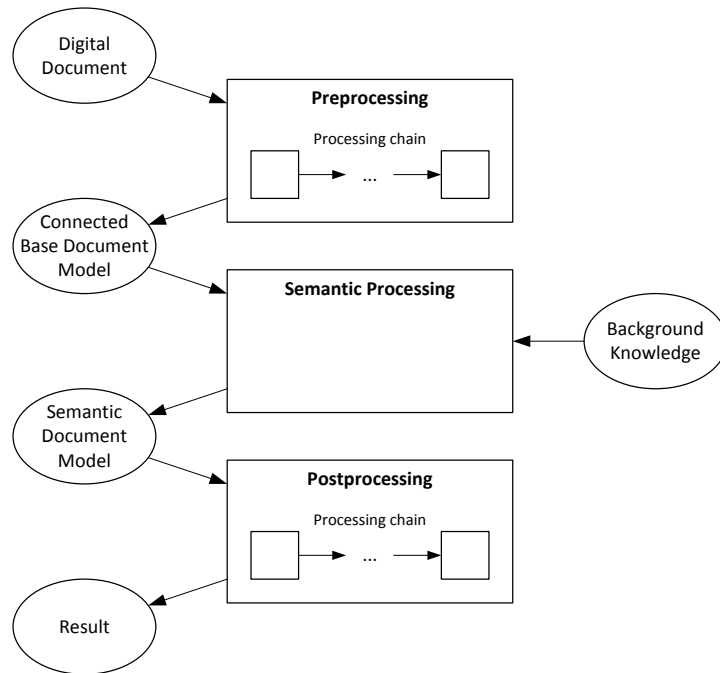


Figure 7.1: System architecture

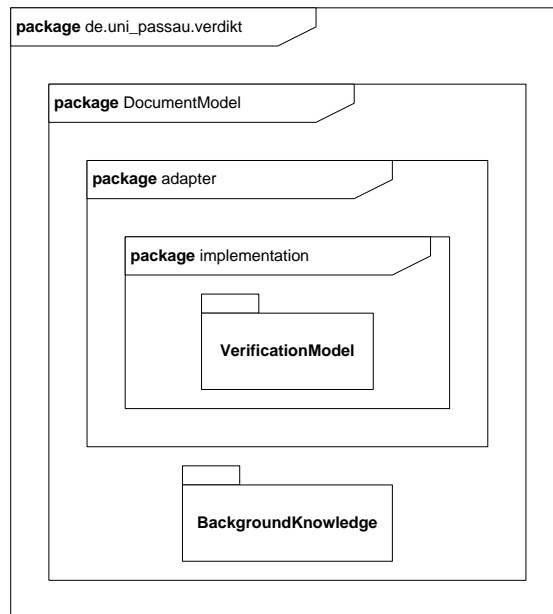


Figure 7.2: Package overview

being straight forward, but not necessarily easy). XML is also a convenient format for further preprocessing.

How semantic document models are represented is shown in the UML diagram in figure 7.3. The central entity is the `DocumentModel` interface that consists of a hierarchy of `Fragments`, starting with a root fragment. Fragments represent parts of a document model. They can hold references (`Annotations`) to other fragments (in particular, has-part and successor references), references to ontology classes (in particular, to classes of a structure ontology), and references to literal data values (in particular, to terminological data). The `Fragment` interface provides several convenience methods for obtaining and manipulating specific references (`getAnnotations()`, `getChildren()`, `getParent()`, `getSuccessor()`, `addChild()`, ...).

The `DocumentModel`, in addition to aggregating fragments, also provides several methods for retrieving fragments and other resources, for example by their identifier (`findFragment()`), by their relations to other fragments or resources (`findSubjects()`, `findObjects()`), or by specifying graph queries (`queryFragments()`, `queryLiterals()`).

A factory pattern is used in the creation of new fragments and annotations. As stated above, an implementation exists that is based on Jena.

A document model relies on a `Vocabulary` that holds the names of all annotations that can represent relations on the document model. For special relations, such as the successor relations, the has-part relation, or the relation that binds ontology classes to fragments, methods exist that return a default name for the appropriate annotations (`getDefaultReferenceAnnotationName()`, `getDefaultChildAnnotationName()`, `getDefaultTypeAnnotationName()`). For annotations with domain or range restrictions, these restrictions can be queried (`getAnnotationDomain()`, `getAnnotationRange()`).

The vocabulary also holds the available class names of the structural ontology (`getFunctionTypes()`, `getStructuralTypes()`). The classes are divided into two groups: the classes that solely represent structural elements such as chapters or paragraphs, and the classes that also indicate that the represented element serves an additional function, such as a definition or an example. A list of the entire document vocabulary developed in the course of this thesis can be found in appendix A.

If an annotation name is not known, then `getAnnotationName()` can be used to find the closest match to a given string in the available names. This makes the handling of the vocabulary easier, since knowledge engineers do not have to know the exact name of an annotation (e.g., has-part, has-part, or hasPart).

As many other classes and interfaces in this implementation, document models are `Configurable` with an instance of the `IConfiguration` interface. Available configuration options for document models depend on the implementation. For the Jena-based implementation, they include persistent storage information for both the vocabulary and for the document model itself.

The `BackgroundKnowledge` package shown in figure 7.4 contains all interfaces and classes relevant for representing and dealing with background knowledge. `BackgroundKnowledgeBase` is the base interface for knowledge bases. It provides “term groups”, which serve a similar, if simplified, purpose as concepts in ontologies. They serve as named aggregations of terms, which can be seen as an abstraction of individuals. The interface provides access methods for terms and term groups, in particular the `getTermGroups()` method. It returns a `Map` that maps term group names onto sets of terms.

The `Ontology` class serves as a wrapper for the Jena ontology implementation. It provides additional access methods to obtain terms based on their relations to other terms. `InferenceOntology` extends `Ontology` with inference services. `SKOSOntology` can be used to represent ontologies in the SKOS format (cf. section 3.3.4). It supports multiple languages and

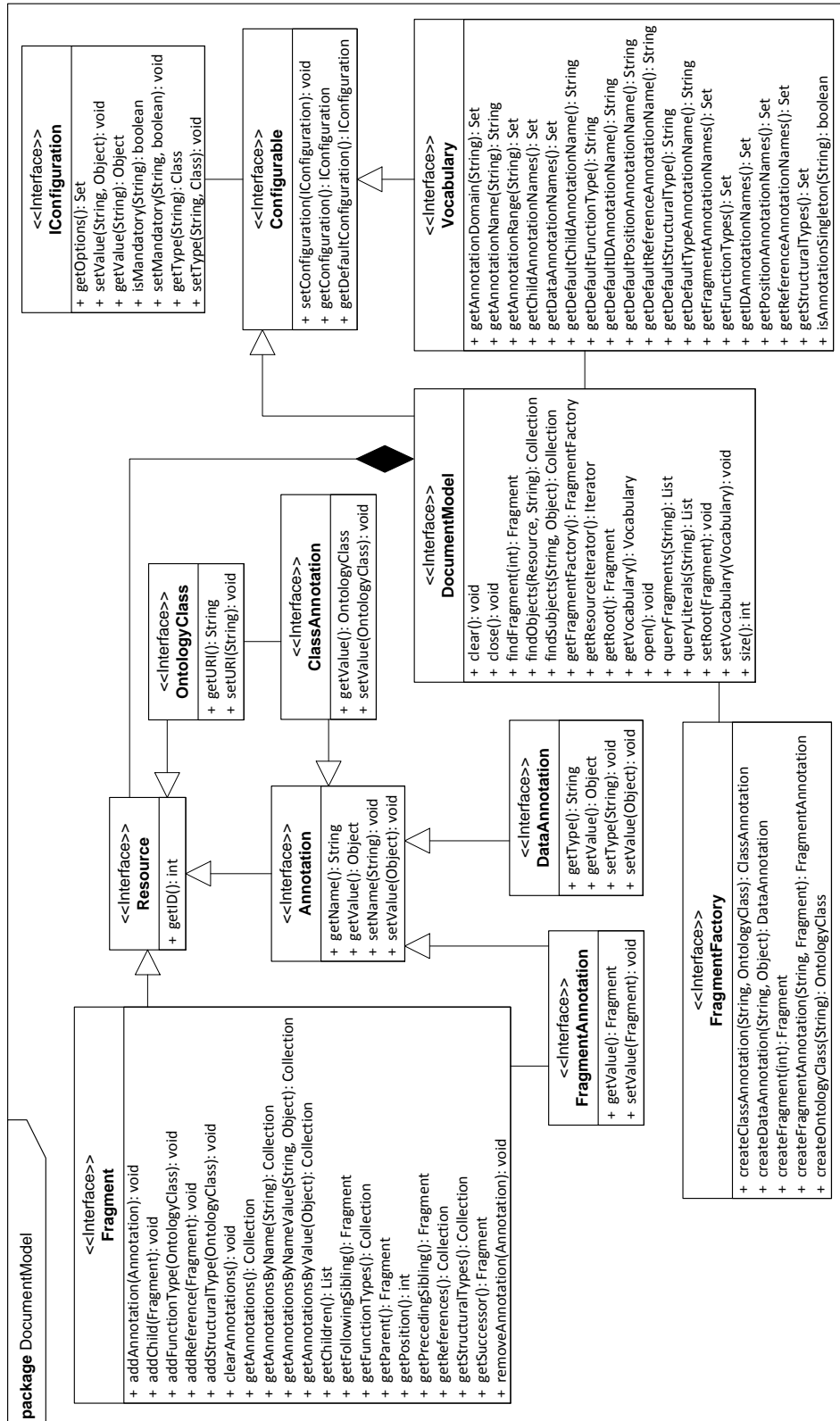


Figure 7.3: UML diagram: document model

provides methods for obtaining the **broader** and **narrower** relationship instantiations of terms. **KnowledgeExtendable** is an interface for all classes that make use of knowledge bases.

DataMapping represents a mapping as used in section 5.1. It contains domain- and format-dependent information about what structural types or relations are represented by specific XML elements and attributes.

HugeOntology is an interface for dealing with very large ontologies. There is an in-memory implementation for computers with a large amount of working memory, a database-backed implementation, and a hybrid implementation (**HugeOntologyMemDB**). The latter keeps the **TBox** in main memory and moves the **ABox** to a database. Reasoning is only done on the **TBox**. The **getObjectsForClass()** method aggregates the objects for a specific class and all its sub-classes from the database. This implementation is far more efficient than a purely database-backed solution, and far more scalable than a purely main-memory-backed solution (cf. section 11.5). Inference services are restricted to subsumption and instantiation, however.

Figure 7.5 shows part of the **adapter** package. It is centred on the **GenericDocumentAdapter**, the base interface for all adapters that read or process documents.

The **ExternalExtractionDocumentAdapter** is an interface for obtaining an XML document from a URI. The **XMLProcessingDocumentAdapter** is an interface for preprocessing such an XML document. The **ExtractionDocumentAdapter** is an interface for obtaining a **DocumentModel** from an XML document, i.e., it can be used to extract a semantic document model from a base document model. The **ExtractionAdapterPipeline** unites these adapters into a single pipeline from an XML file to a **DocumentModel** to make handling easier.

The **DocumentModelProcessingDocumentAdapter** is an interface for postprocessing a **DocumentModel**. The **GenerationDocumentAdapter** is an interface for obtaining an XML document from a **DocumentModel**. The **ExternalGenerationDocumentAdapter** is an interface for writing an XML document to a location identified by a URI. The **GenerationAdapterPipeline** unites these adapters into a single pipeline from a **DocumentModel** to an XML file.

The abstract **RuleDocumentAdapter** class implements several of these interfaces, allowing for the extraction, generation, and processing of files and models using rules.

The **DocumentAdapterManager** is a management class that can be used to list and access available adapters.

Figure 7.6 shows the second part of the **adapter** package. The **GlobalContext** and **LocalContext** interfaces serve as implementations of global and local environments, respectively (cf. definitions 5.1.16 and 5.1.14). They also provide many convenience methods for accessing background knowledge and for constructing a **DocumentModel**. The **visit()** method of the **GlobalContext** keeps track of processed files to prevent infinite loops when following circular references. The **GlobalContextRegister** serves as a register for multiple **GlobalContexts**.

Each instance of **LocalContext** is meant to hold exactly one media object of the base document model's implementation, e.g., one XML node. It is also meant to hold an optional reference to a **Fragment**, i.e., to a document fragment in a document model. This is adequate because usually a single node in a base document model may lead to the creation of at most a single fragment in a semantic document model, simply because a single atomic media object (the node) cannot hold more structural information than for a single fragment. However, if there exist some special case in which this assumption is not true, it is still easily possible to create two or more local contexts that hold the same media object, but different fragments. Yet in the course of this work, we have never encountered such a case.

The **DocumentException** and its two sub-classes are used to encapsulate errors that occur during the processing of documents or document models.

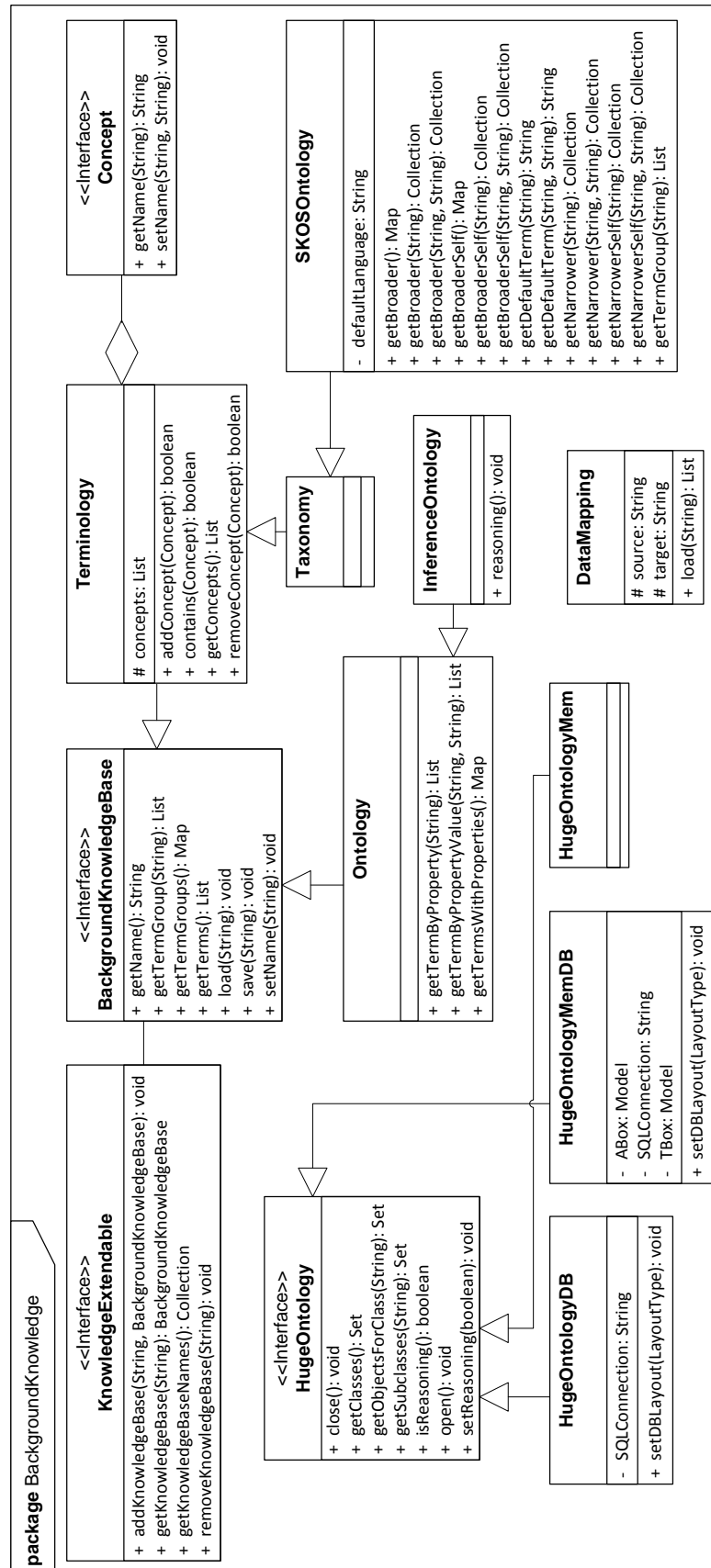


Figure 7.4: UML diagram: background knowledge

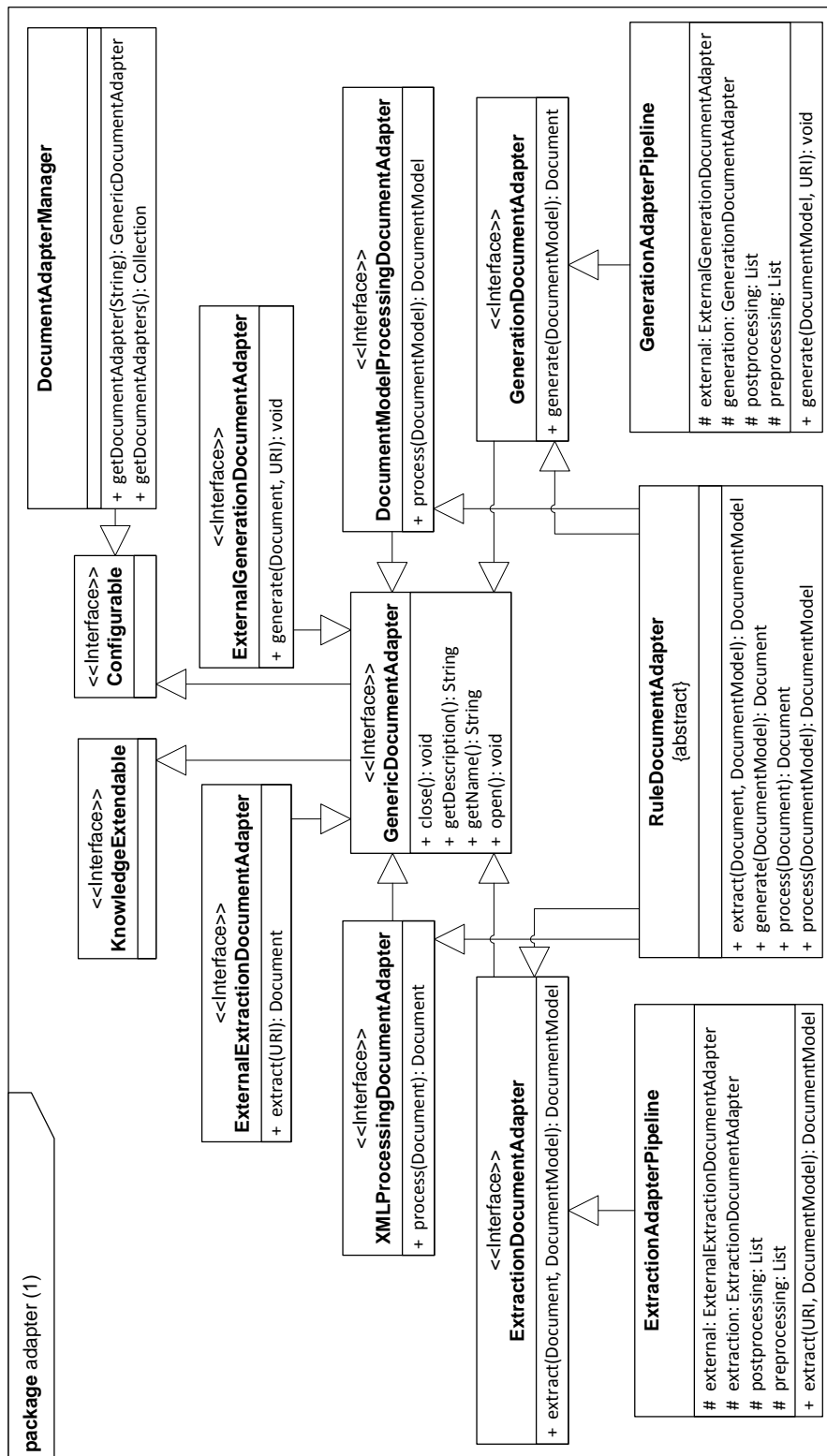


Figure 7.5: UML diagram: document adapters (1/2)

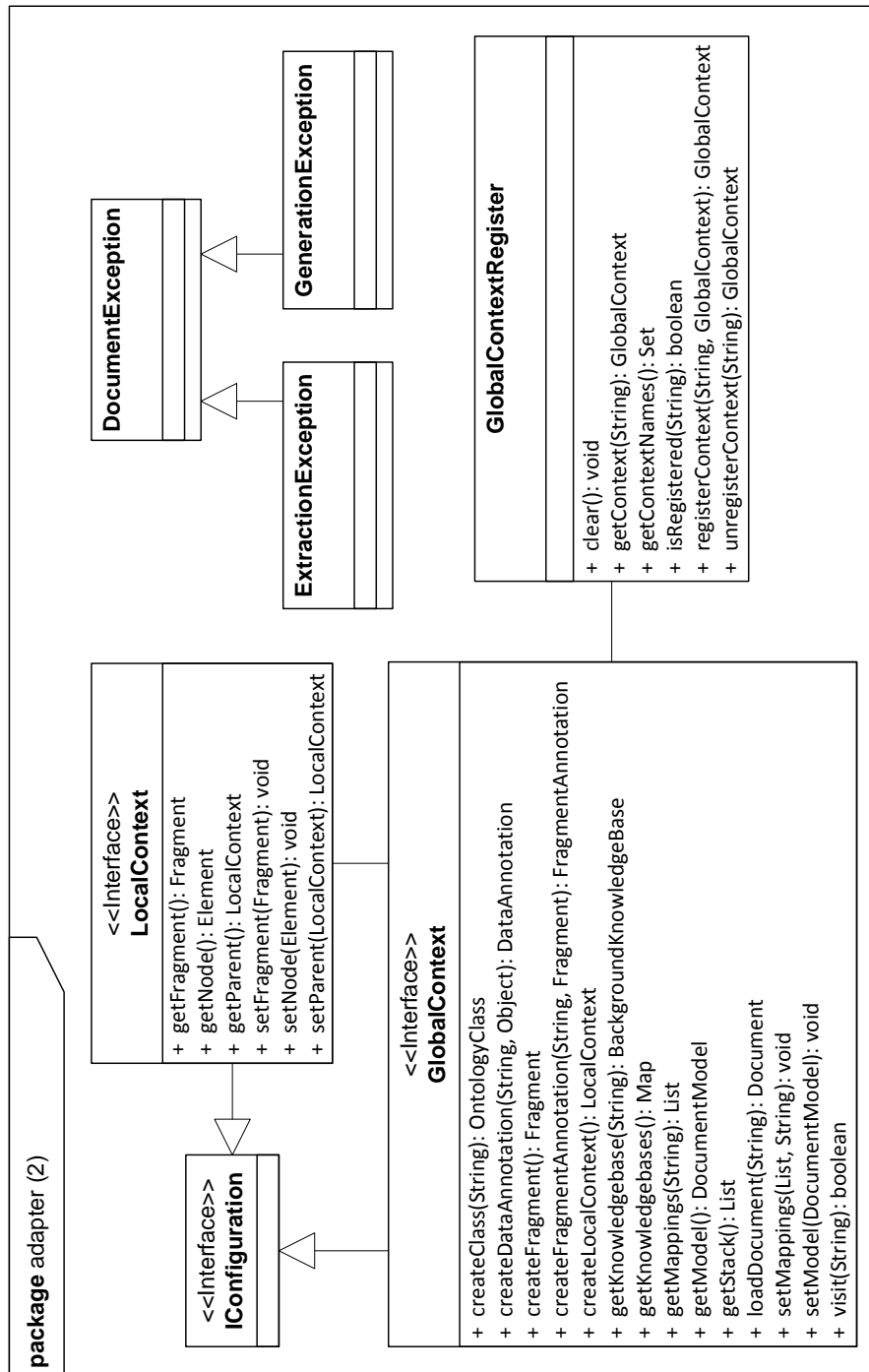


Figure 7.6: UML diagram: document adapters (2/2)

7.2 Preprocessing

Figure 7.7 shows the classes from the `implementation` package that implement preprocessing functionality.

The `XMLDocumentAdapter` simply reads and writes XML documents from and to files. The `XMLTidyDocumentAdapter` reads XML documents from files and attempts to correct documents that are not well-formed. This is especially useful for reading HTML documents that often do not adhere to the strict requirements of XML documents.

The `LatexDocumentAdapter` attempts to parse a file in \LaTeX format and represent it as an XML document. \LaTeX environments, such as `\begin{itemize} ... \end{itemize}`, are converted to opening and closing XML elements. \LaTeX commands without parameters, such as `\pagebreak`, are converted to empty XML elements. \LaTeX commands with parameters, such as `\emph{}` are converted to XML elements with attributes and/or child elements, depending on the type of parameters. Due to the very high complexity and versatility of the \LaTeX syntax, this adapter only supports environments and commands that were previously defined. Such a list must be provided to the `LatexDocumentAdapter` as a parameter.

The `WordDocumentAdapter` attempts to parse a file in Microsoft Word format (pre-XML, file extension `.doc`). It uses an external tool written in `C#` for this thesis, that makes use of the Visual Studio Tools for Office (VSTO). The VSTO provides an API for office documents, including word documents. Using the VSTO has several disadvantages, however, the most important being their meagre performance in terms of efficiency and reliability, and their reliance on Microsoft Office being installed on the machine using the tools.

The `DocXDocumentAdapter` attempts to avoid these disadvantages. It can only be used for files in a newer version of the Microsoft Word format (XML-based, file extension `.docx`). Instead of relying on external tools, it parses the XML files of a Word document directly and extracts both the textual content and its formatting, including named styles.

The `GZipDocumentAdapter` provides read and write access to files compressed in `gzip`-format. This is particularly useful when processing archived corpora of documents.

The `CombinationDocumentAdapter` allows for combining multiple XML files into a single XML document. This is particularly useful when processing documents that consist of multiple files, or when processing corpora of multiple documents as a single contiguous entity.

The `XQueryDocumentAdapter` allows using XQuery for specifying extraction or generation instructions. XQuery programs can be used to preprocess XML documents, to extract and process `DocumentModels`, and to postprocess models.

The `ML3DocumentAdapter` provides custom preprocessing for documents in the ML3 e-learning format. This includes resolving the inclusion of other files, and dealing with multiple versions of a document in a single set of files (see below).

7.3 Semantic Processing

Figure 7.8 shows the classes from the `implementation` package that implement semantic processing functionality.

The `XMLModelDocumentAdapter` provides serialisation functionality for `DocumentModels`. It can be used to read and write processed models from and to XML documents.

The `DroolsDocumentAdapter` extends the abstract `RuleDocumentAdapter` class with an implementation based on the JBoss Drools rule language (cf. section 3.4). Drools rule sets can be used to preprocess XML documents, to extract and process `DocumentModels`, and to postprocess models. Since Drools rule sets need to be compiled before they can be used, the adapter provides

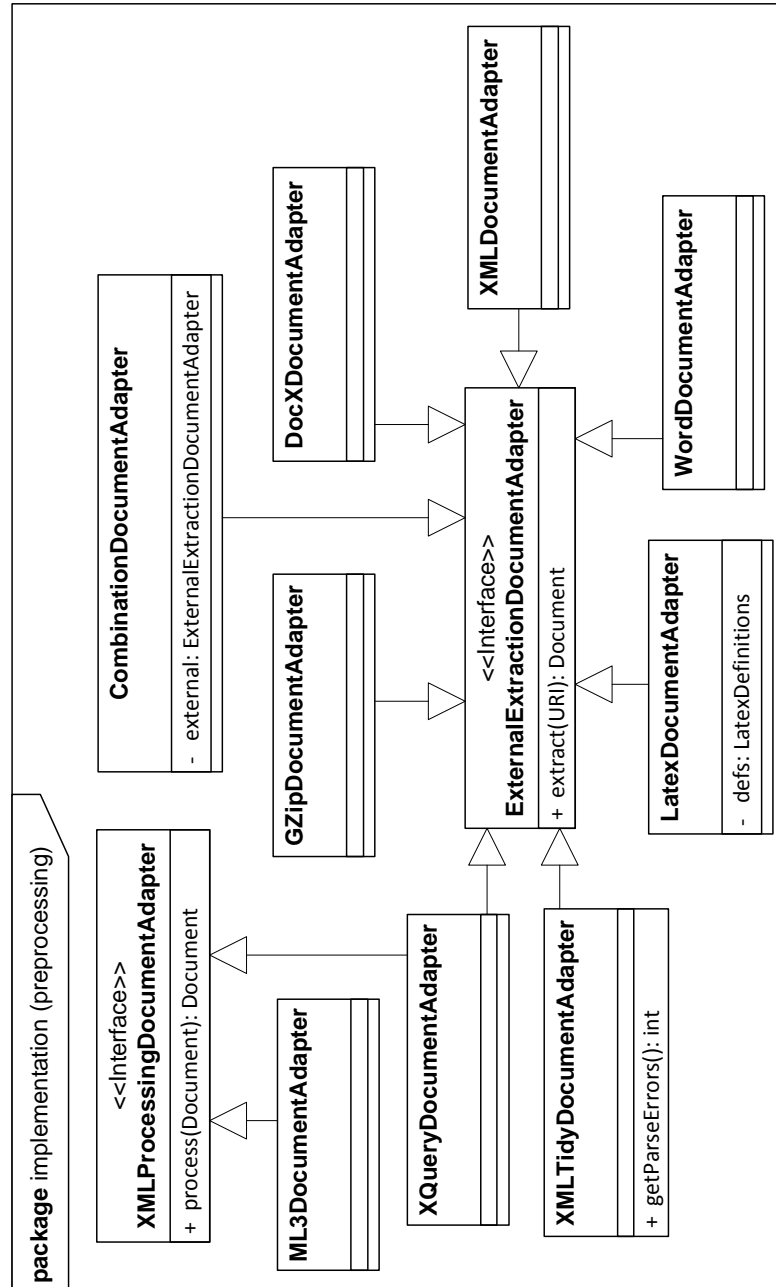


Figure 7.7: UML diagram: preprocessing

a method for pre-compiling them. This allows the implementation to cache a pre-compiled version of the rules and use it without additional compilation overhead until the rule set changes.

`DroolsGlobalContext` and `DroolsLocalContext` are Drools-specific extensions of the abstract `AbstractGlobalContext` and `AbstractLocalContext` classes, respectively.

The `XQueryDocumentAdapter` has already been described above. `XQueryGlobalContext` and `XQueryLocalContext` are XQuery-specific extensions of the abstract `AbstractGlobalContext` and `AbstractLocalContext` classes, respectively.

The `RDFExtractionDocumentAdapter` attempts to read `DocumentModels` directly from a document in RDF format, provided that the RDF document uses RDF vocabulary that is compatible with semantic document models (cf. appendix A).

7.4 Postprocessing

Figure 7.9 shows the classes from the `implementation` package that implement postprocessing functionality.

`XMLDocumentAdapter`, `GZipDocumentAdapter`, `XQueryDocumentAdapter`, `XMLModel-DocumentAdapter`, and `DroolsDocumentAdapter` have already been described above.

The `CTLDocumentAdapter` creates a temporal model for CTL model checking (cf. section 3.5.1) from a `DocumentModel`. The layout of this temporal model is specified as a `VMSpecification` parameter (see below).

Similarly, the `ALCCTLDocumentAdapter` creates a temporal model for \mathcal{ALC} CTL model checking (cf. section 3.5.2) from a `DocumentModel`. The layout of this temporal model is also specified as a `VMSpecification` parameter.

The `HTMLDocumentAdapter` creates an HTML document for a `DocumentModel`. The HTML document is a generic example of a document for which the `DocumentModel` represents a semantic document model.

The `GraphDocumentAdapter` shows a graph view of a `DocumentModel`, using the JUNG Graph library¹.

The `BGKGenerationDocumentAdapter` attempts to extract background knowledge from a `DocumentModel`, as discussed in section 6.2. It uses the `Vocabulary` of the `DocumentModel` to determine which annotations may hold suitable information. In particular, it uses the data annotations defined in the vocabulary, except specific annotations like identifiers.

Counterpart to the `RDFExtractionDocumentAdapter`, the `RDFGenerationDocumentAdapter` creates an RDF document that represents a `DocumentModel`.

Figure 7.10 shows the classes from the `VerificationModel` package that are used to define the layout of a temporal model for model checking.

The `VMSpecification` class combines all parts of the layout specification for a temporal model. It contains one instance of `VMStateSpecification` that specifies which parts of a `DocumentModel` should be represented by states in the temporal model. Most commonly, fragments that either represent chapters or paragraphs in the original document serve as the boilerplates for states in the temporal model.

The `VMSpecification` also contains one instance of `VMStartingStateSpecification` that defines a single starting state for the model checking problem. It contains one instance of `VMSuccessorSpecification` that defines the successor relation between states.

Finally, the specification contains lists of `VMConceptSpecifications` and `VMRoleSpecifications` that define concepts and roles and their respective interpretations in each state. The role definition first specifies a base, relative to which both role partners are specified. In other

¹<http://jung.sourceforge.net/>, visited 05/2013

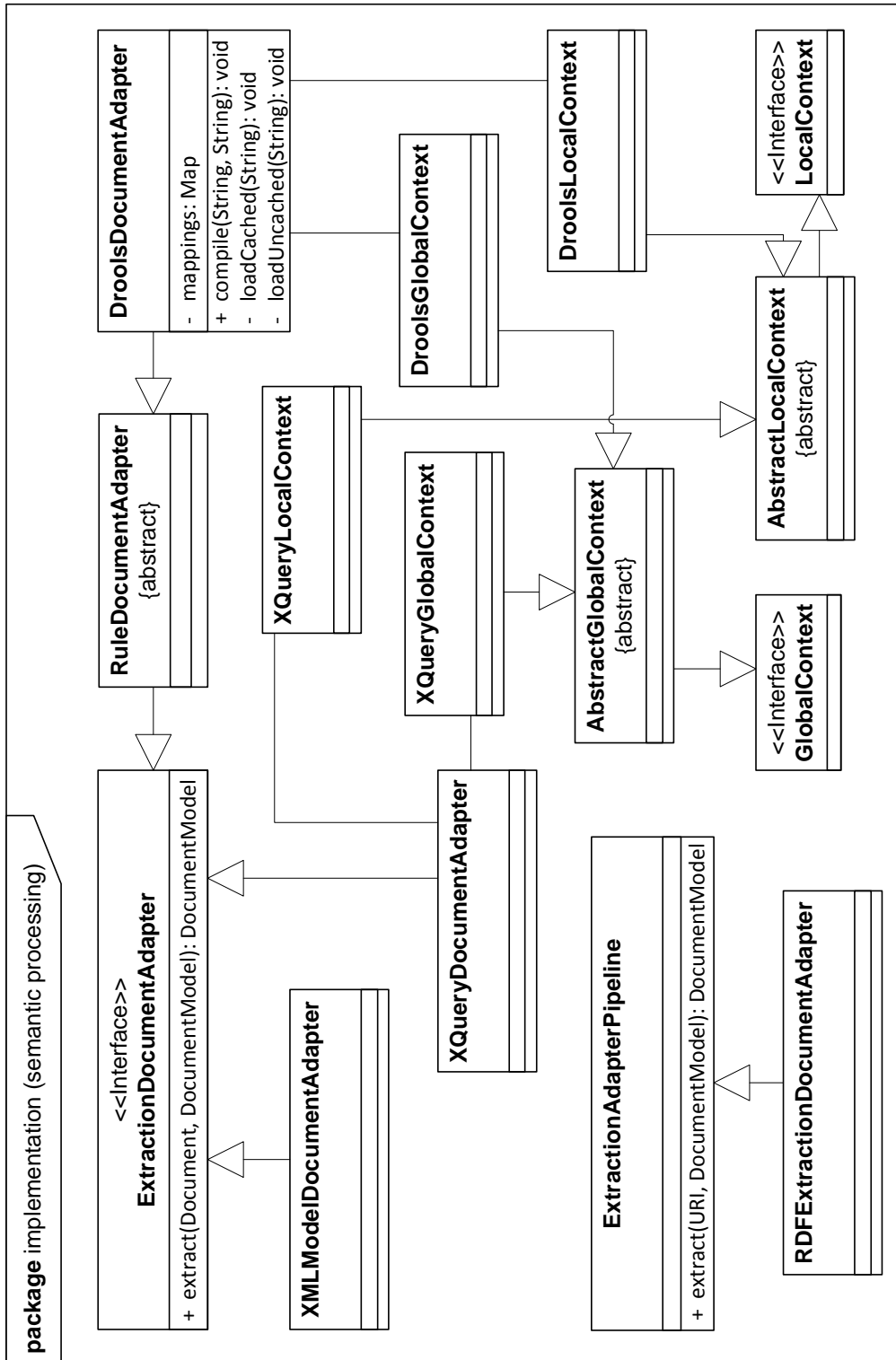


Figure 7.8: UML diagram: semantic processing

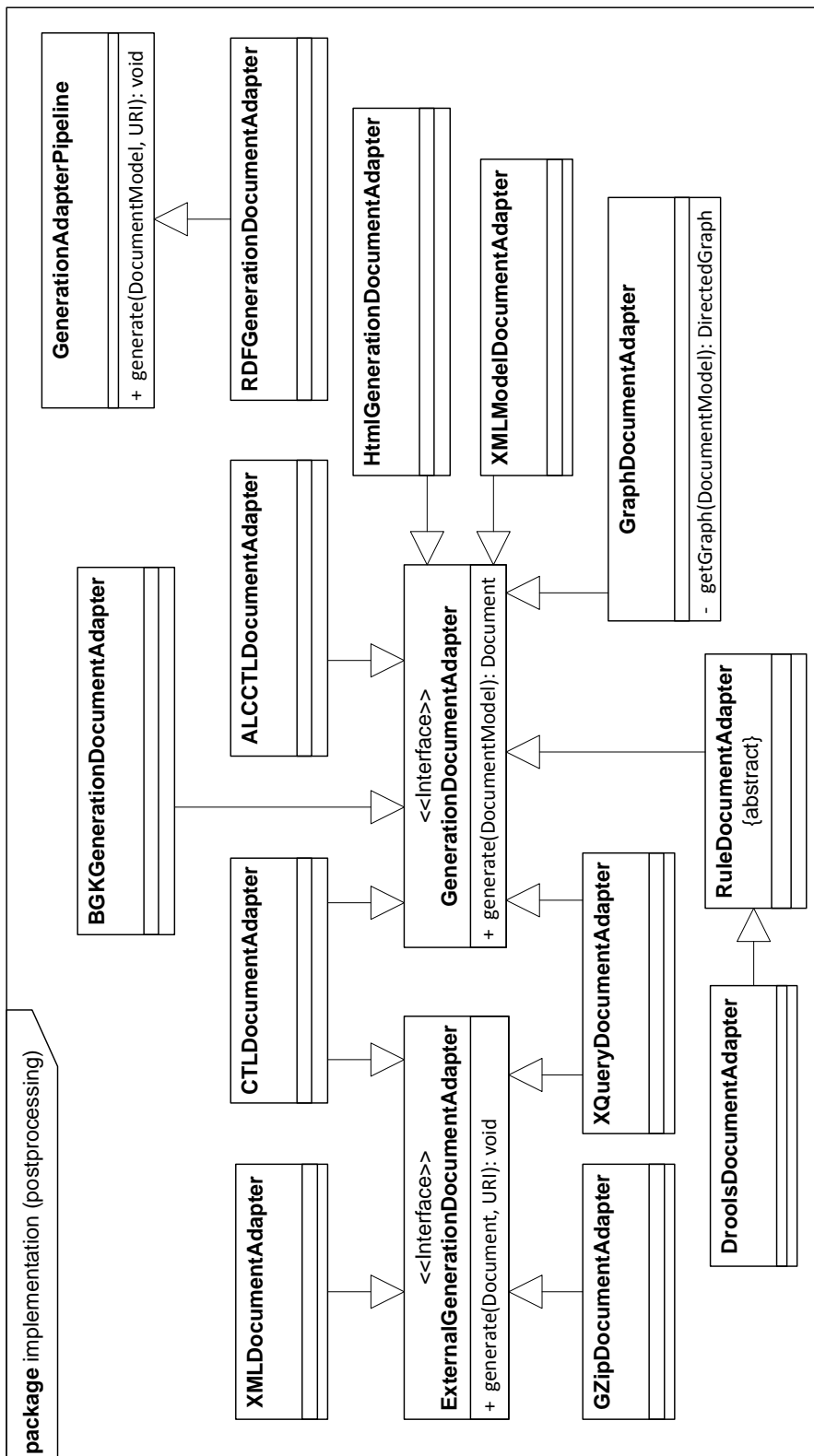


Figure 7.9: UML diagram: postprocessing

words, it points to a location in the `DocumentModel`, for example to all fragments that represent definitions. From this base point, it points to two relative locations, for example to the fragment identifier and to the topic associated with the fragment. The resulting role interpretation would pair definitions with topics, as in a `hasDefinedTopic` role.

All specifications are based on SPARQL queries that can be executed against a `DocumentModel`.

Both the concept specification and the role specification make use of `VMDataPostProcessing` instances that further process the data retrieved for the interpretations. They can for example be used to enforce uniform spelling conventions.

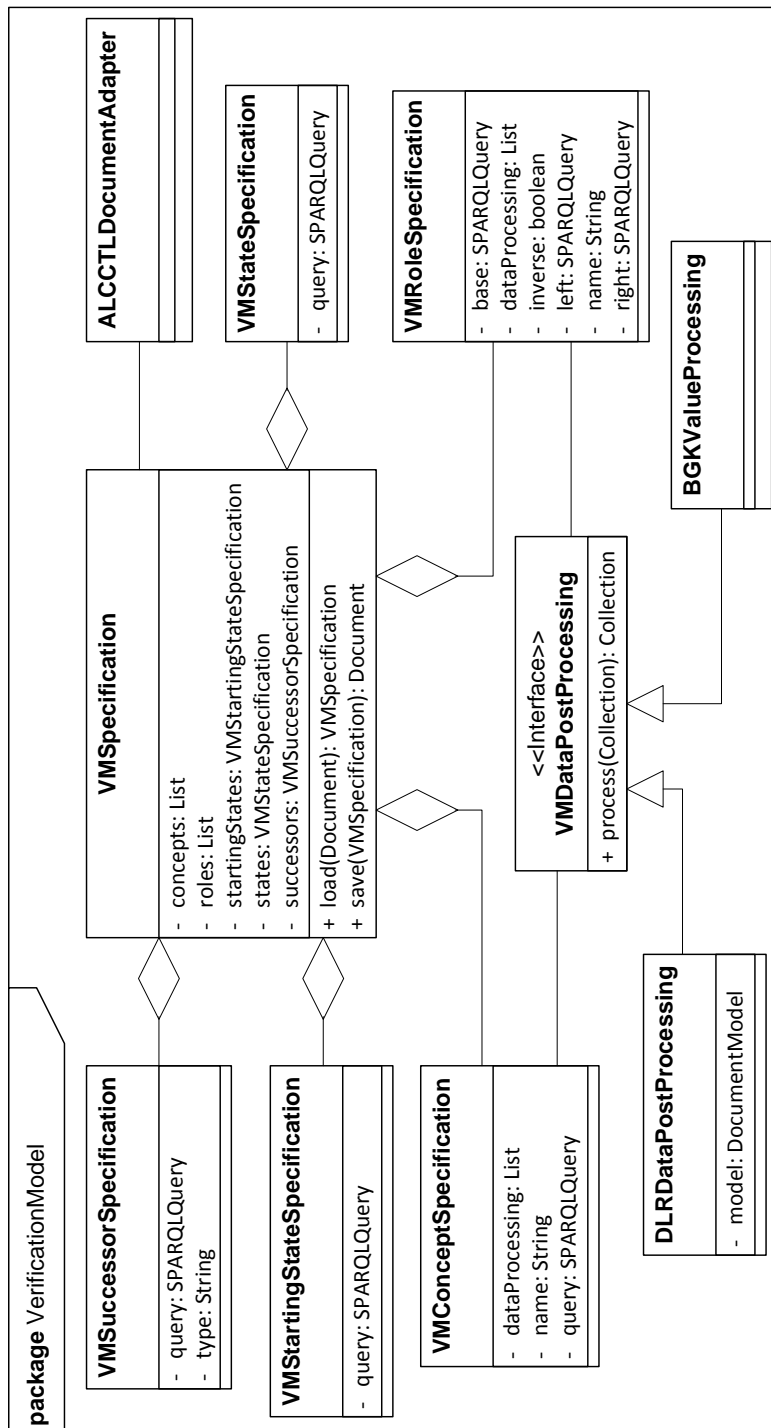


Figure 7.10: UML diagram: verification model

Chapter 8

Implementing Document Models

In this chapter, we will discuss implementation options for document models, and the implementation we chose for this thesis.

8.1 Implementation Basics

For implementing document models, there are several options to choose from. Important properties of such an implementation are an efficient representation of both the structure and the semantics of a model based on description logics.

The first option is to develop some custom data format for representing the model, and to use a rule language and interpreter to realise its semantics. There exist, a number of resources that make a completely new development unnecessary. Nonetheless, this option requires a large effort in creating components that might already exist for other options.

Another option is to implement document models in Datalog or Prolog. While the necessary inference rules would also have to be specified manually, this poses no great challenge. However, both Datalog and Prolog employ backward chaining as their reasoning method, thus working backwards from a specified goal. But when using document models, the goal may not be known beforehand, or there may be more than one goal. This makes the forward chaining method more useful, since it only needs to be done once and it can be done before the model is used.

This makes OWL, combined with RDF, a good candidate for implementing document models that we will now investigate further.

When OWL/RDF is used to model data, there exist a number of implementations that can be used, such as Jena [Jen] or Sesame [Ses]. The features of such an implementation that are crucial for working with semantically rich document representations are *scalability* and *reasoning support*. Scalability is important because the data that needs to be stored can become very considerable in size, either in the form of very large document models, or in the form of extensive background knowledge (cf. chapters 6 and 11). Scalability can most easily and reliably be achieved by providing a database-backend for the data storage, which is available in many RDF frameworks. In addition, the underlying structure of RDF data is a graph structure, which allows for a very efficient and natural implementation of the relation-based structure of document models (see below).

Reasoning support is needed to infer new and relevant data from the existing statements. This includes additional type information for fragments (i.e., the transitive closure of `rdfs:subClassOf`) and equivalence classes (i.e., the transitive closure of `owl:sameAs` and its

derivatives). Most RDF frameworks either offer a direct implementation of the required reasoning algorithms, or they provide an interface for using external reasoners such as Pellet or HermiT. RDF/OWL reasoning is based on forward chaining, driven by the available assertions, and OWL DL semantics are based on description logics. This makes it an ideal choice for the implementation of semantic document models.

Also important but less crucial is support for schema specification languages like RDF(S) and query languages like SPARQL. A schema specification not only provides assertions that can be used to infer new data, but it also limits modelling in a way that allows for finding inconsistencies. A query language can be used to retrieve specific parts of a document model, which is important when extracting data or when generating new types of models from a document model.

In general, it is unimportant which specific schema or query language is used. For example, earlier versions of Sesame did not support SPARQL, but provided a custom query language named SeRQL with similar capabilities. However, support for standardised data and query formats clearly facilitates easier exchange of data. At the time of this writing, the existing frameworks support all relevant standards.

Both Jena and Sesame provide the required scalability and reasoning support. In current implementations, there is an important caveat, however. When the number of RDF statements grows so large that they need to be taken out of main memory, i.e., if they need to be written to a database, the reasoning over these statements becomes cripplingly inefficient. We will address this issue in section 11.5.

8.2 Implementing Document Fragments

Fragments of a semantic document model are best implemented as RDF named resources. To this end, each fragment is assigned a unique numerical identifier. Combined with a constant URI prefix like `http://www.verdikt.uni-passau.de#ID`, this identifier yields a URI to represent the named resource, for example `http://www.verdikt.uni-passau.de#ID04680347`.

This allows for an easy and global identification of document fragments, even across document models, which is useful when combining multiple document models to a larger model of a document corpus. It also allows for an efficient implementation, because a simple number is often sufficient for identification and handling, instead of a more cumbersome URI.

Unique identifiers can be obtained by calculating checksums on the source data, e.g., on the absolute and unique XPath expression leading to the XML element or other media object that induces the current fragment. Extended with the filename, such an XPath expression can look like this: `/documents/file.html#html/body/div[3]/p[2]`.

RDF statements can be used for annotating type information in the form of RDF/OWL classes to fragments. Data-valued statements can be used for annotating literal values, such as topic information or metadata.

The vocabulary for predicates and classes is described in appendix A.

8.3 Implementing Relations

Relationships between fragments can be implemented as object-valued RDF statements.

8.3.1 Implementing Include Relations

Some document formats offer the possibility of including (parts of) other documents, for example the `\input{}` command in L^AT_EX. In case a particular document or document part is included

multiple times, there are two ways to implement this. In the formal semantic document model, each included document or document part is represented as a unique fragment (cf. chapter 4). The first option is to let the implementation reflect this and to represent the included fragments multiple times. This results in additional space requirements and redundancy in the data.

The second option is to implement the included fragments only once and to reference it multiple times. The drawbacks for this option are more severe, however. When processing a document model, each fragment needs to be regarded in its local context, which includes its place in the document structure. For a fragment that is included multiple times, this context depends on the path that was taken to reach it. For example, a fragment can inherit the topics of its parent fragment. In a fragment that is included multiple times, the actual topics change with the path taken to reach the fragment, because the parent fragment changes.

Since the additional space requirements are less costly than the effort required to differentiate between multiple contexts of a single fragment that was included in multiple places in a document structure, we implement include relations en par with their representation in the document model, namely by duplicating the included fragments.

8.3.2 Implementing Reference Relations

In most types of documents, references from one part of the document to another are in a linear order. For example, if there are two outgoing references from chapter 2, one pointing to chapter 3 and one pointing to chapter 4, then one of these references comes before the other. This order should be reflected in the implementation.

Figure 8.1 illustrates four options for implementing reference relations using a small hyperdocument with four fragments as an example. On the left hand side (a), the order of the reference relationships is implemented by annotating the edges with numbers. For example, the reference from *A* to *B* comes before the reference from *A* to *C*. Next (b), the order is implemented by annotating the target nodes with numbers. As evident in node *C*, this is not sufficient in a graph structure because the reference point for the numbering – namely, the predecessor – is not unique. It is therefore unclear if *C* is the first reference of *A* and the second reference of *B*, or vice versa.

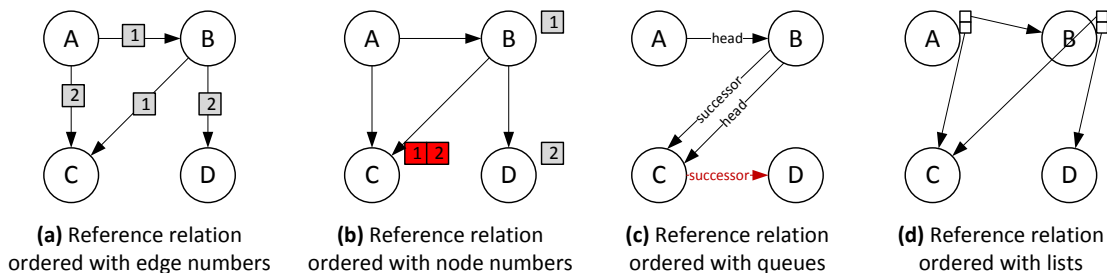


Figure 8.1: Implementation options for reference relations

Implementing the order with a queue of a single head relationship followed by a sequence of successor (tail) relationships (c) is insufficient for similar reasons. *A* has references to both *B* and *C*, in that order. This is implemented as a *head* relationship between *A* and *B*, and a *successor* relationship between *B* and *C*. Other references from *A* would be implemented as a

chain of further successors, starting in C . However, C is also in a *head* relationship with B , because B references C . It is therefore unclear if the successor relationship between C and D is a continuation of the successor chain from B , or if it is the beginning of a new successor chain from C .

On the right hand side (d), implementing the order of the reference relation using a list structure is depicted. Both options (a) and (d) are valid possibilities. Annotating the edges (a) allows for the efficient determination of the relative order of two references simply by comparing the annotated numbers. It also avoids the cumbersome handling of OWL lists, with positive effects on the readability of the program code. Using a list structure makes iterating in the correct order over all outgoing edges of a node more efficient.

Unfortunately, not all formalisms allow the implementation of annotated edges. For example, description logics do not support annotated role assertions. OWL does allow annotated statements via reification, but does not provide a consistent semantics for it. Since experience has shown that the order of references is only relevant in rare cases, we implement reference ordering via reification, and use a custom SPARQL query to retrieve this order when necessary.

Remark 8.3.1. *Note that reference relations are no ordering relations, because the possibility of cycles conflicts with the antisymmetry requirement of a (partial) order: two fragments that reference each other would have to be equal, which is not generally the case.*

In a document model, it is important to differentiate between immediate successors and indirect (transitive) successors. Therefore, successor relations should not be implemented as transitive, i.e., they should not be implemented as sub-properties of `owl:TransitiveProperty`, because after a reasoning step the inferred relationships cannot be distinguished from the original ones. Transitive super-properties of successor properties, however, can be useful for conveniently determining the transitive closure of these properties.

For example, for a successor relation implemented as `successor`, an additional relation `successor_trans` with `successor_trans` \sqsubseteq `successor_trans`⁺ and `successor` \sqsubseteq `successor_trans` can be defined. Provided that `successor_trans` has no other sub-properties, it represents exactly the transitive closure of `successor`.

8.3.3 Implementing Has-Part Relations

Different from reference relations, has-part relations are defined as left unique, i.e., no fragment in a document model may be part of more than one other fragment. This restriction puts options (b) and (c) from figure 8.1 back on the table for has-part relations.

Using a queue to implement the order of has-part relationships, i.e., which sub-fragment comes before another, makes handling inconvenient. It also increases the cost of determining the relative order of two fragments.

While numbered edges are not supported by all formalisms, numbered nodes pose no such problem. Since many use cases involve the determination of the relative order of fragments or appending a new fragment after the last existing fragment, where numbered nodes are more efficient than lists, we use numbered nodes to implement has-part relations.

As with successor relations, has-part relations should not be implemented as transitive, but transitive super-properties can be useful. They should, however, be implemented as injective (`owl:InverseFunctionalProperty`) to guarantee left uniqueness, i.e., $\top \sqsubseteq \leq \text{has-part}^-$. Other tree properties must be ensured by the document model framework.

8.3.4 Implementing Literal-Valued Relations

As stated above, relationships with a literal value can be implemented as literal-valued statements. In particular, terms that are topics of fragments can be implemented as literals. However, literals cannot be the subject of a statement. This leads to problems when trying to implement the terminological ontology \mathcal{O}_T of a semantic document model. \mathcal{O}_T contains assertions about terms, where these terms can be both subjects and objects of statements, e.g., `hasBroader(Binary Tree, Data Structure)`.

This leaves two options for implementing terms. The first option is to replace all literal values with resources, for example instances of SKOS concepts, and to give these resources an annotation that contains the actual literal value, for example `skos:prefLabel`. This results in statements like

```
vdk:ID21      rdf:type      vdk:Paragraph ;
              vdk:topic    vdk:DataStructure .
vdk:DataStructure  rdf:type      skos:Concept ;
                  skos:prefLabel  "Data Structure"@en .
vdk:BinaryTree   rdf:type      skos:Concept ;
                  skos:prefLabel  "Binary Tree"@en ;
                  skos:broader    vdk:DataStructure .
```

The second option is to leave all terms as literal values in the document model, but to use resources, for example instances of SKOS concepts, for the implementation of \mathcal{O}_T . This results in statements like

```
vdk:ID21      rdf:type      vdk:Paragraph ;
              vdk:topic    "Data Structure"@en .
vdk:DataStructure  rdf:type      skos:Concept ;
                  skos:prefLabel  "Data Structure"@en .
vdk:BinaryTree   rdf:type      skos:Concept ;
                  skos:prefLabel  "Binary Tree"@en ;
                  skos:broader    vdk:DataStructure .
```

We use the second option, because it requires fewer joins on the data. Terms do not need to be resolved to their literal value, which is a frequent requirement when processing a document model. In addition, this reduces the dependency on the background knowledge, so that the core part of the document model can be used and understood on its own.

We will now show an example implementation of a semantic document model based on the decisions made in this chapter.

Example 8.3.2 (Document Model Implementation). *The semantic document model from example 4.2.15 can be implemented in RDF/OWL as follows.*

Note that the `vdk:position` statements indicate the relative order of child fragments w.r.t. the has-part relation, e.g., fragment `vdk:ID42` with a position value of “2” is the second child of

fragment *vdk:ID4*.

```

@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix skos: <http://www.w3.org/2004/02/skos/core#> .
@prefix vdk: <http://www.verdikt.uni-passau.de#> .
vdk:ID0          rdf:type      vdk:Document ;
                  vdk:part     vdk:ID1 ;
                  vdk:part     vdk:ID2 ;
                  vdk:part     vdk:ID3 ;
                  vdk:part     vdk:ID4 ;
                  vdk:part     vdk:ID5 .
vdk:ID1          rdf:type      vdk:Chapter ;
                  vdk:position  "0"^^xsd:int ;
                  vdk:reference vdk:ID2 .
vdk:ID2          rdf:type      vdk:Chapter ;
                  vdk:position  "1"^^xsd:int ;
                  vdk:reference vdk:ID3 ;
                  vdk:reference vdk:ID4 ;
                  vdk:part     vdk:ID21 .
vdk:ID3          rdf:type      vdk:Chapter ;
                  vdk:position  "2"^^xsd:int ;
                  vdk:reference vdk:ID5 ;
                  vdk:part     vdk:ID31 .
vdk:ID4          rdf:type      vdk:Chapter ;
                  vdk:position  "3"^^xsd:int ;
                  vdk:reference vdk:ID5 ;
                  vdk:part     vdk:ID41 ;
                  vdk:part     vdk:ID42 .
vdk:ID5          rdf:type      vdk:Chapter ;
                  vdk:position  "4"^^xsd:int .
vdk:ID21         rdf:type      vdk:Definition ;
                  vdk:position  "0"^^xsd:int ;
                  rdf:type      vdk:Paragraph ;
                  vdk:topic     "Data Structure"@en .
vdk:ID31         rdf:type      vdk:Example ;
                  vdk:position  "0"^^xsd:int ;
                  rdf:type      vdk:Paragraph ;
                  vdk:topic     "Data Structure"@en .
vdk:ID41         rdf:type      vdk:Definition ;
                  vdk:position  "0"^^xsd:int ;
                  rdf:type      vdk:Paragraph ;
                  vdk:topic     "Binary Tree"@en .
vdk:ID42         rdf:type      vdk:Illustration ;
                  vdk:position  "2"^^xsd:int ;
                  rdf:type      vdk:Paragraph ;
                  vdk:topic     "Binary Tree"@en .
vdk:DataStructure rdf:type      skos:Concept ;
                  skos:prefLabel "Data Structure"@en .
vdk:BinaryTree   rdf:type      skos:Concept ;
                  skos:prefLabel "Binary Tree"@en ;
                  skos:broader   vdk:DataStructure .

```

8.4 Document Lifecycle

Documents that are part of a complex lifecycle can exist in multiple versions. So far, multiple versions of a document have not been modelled as part of a document model. We will now briefly discuss how different versions can be represented in a single semantic document model, and how this affects the implementation.

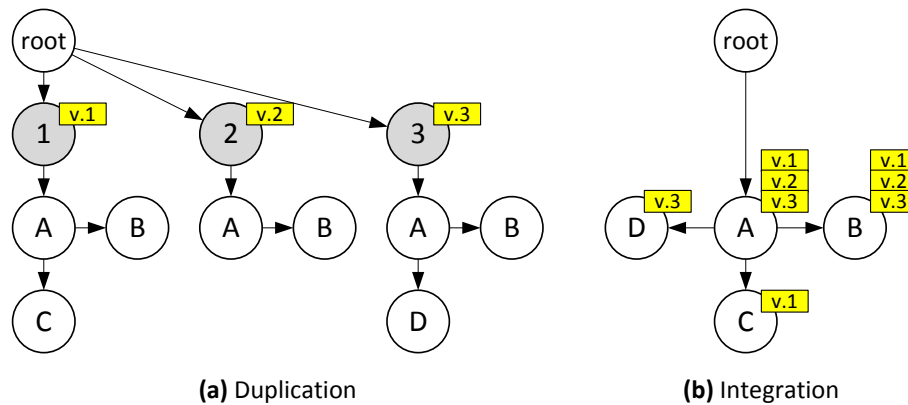


Figure 8.2: Implementation options for document versions (arrows represent has-part relationships)

The simplest way to include multiple versions in a single document model is to introduce new fragments, one for each version, and then use has-part relationships to put them directly beneath the root fragment. Each document version is then modelled as usual, using the corresponding new version fragment as its root fragment. This is shown in figure 8.2 (a), where the version number is highlighted in yellow and the new version fragments are indicated in grey.

The drawback of this approach is a potentially large number of duplicates. By integrating the different versions into a single model and annotating the relevant version numbers directly to each fragment as shown in figure 8.2 (b), the complete document can be modelled without duplicate fragments. On the other hand, this appears to increase the retrieval overhead, because every version-dependent processing step (such as the generation of a verification model) must extricate the fragments relevant for a specific version from the complex model. In the first approach (a), this can be achieved with a simple selection of the appropriate version fragment.

However, in a query language like SPARQL, approach (a) is actually more complex because SPARQL provides no automatic mechanism for selecting an entire sub-tree starting in a specific root node. This leads to a query like

```

1 SELECT ?s ?p ?o
2 WHERE {
3   ?r hasVersion "1" .
4   ?r hasPartTrans ?s .
5   ?r hasPartTrans ?o .
6   ?s ?p ?o
7 }

```

for extracting all statements that belong to version “1”, where `hasPartTrans` is a transitive super-property of the `has-part` property.

On the other hand, extricating the relevant statements in approach (b) leads to a query like

```
1 SELECT ?s ?p ?o
2 WHERE {
3     ?s hasVersion "1" .
4     ?o hasVersion "1" .
5     ?s ?p ?o
6 }
```

which, by the simple virtue of requiring one less sub-graph pattern match, is less costly to execute.

Barring future research, we therefore tentatively recommend modelling and implementing document versions by integrating the version information into the (single) document model.

8.5 Implementing Process Models

Process models can be implemented in a similar manner to document models. A different vocabulary for predicates and classes is required, however. This is described in appendix A.

Conclusion

In this chapter, we have discussed several implementation choices for semantic document models and their respective merits. We have outlined the reasoning behind the choices we made for our implementation and given a brief outlook on dealing with document lifecycles.

Chapter 9

Processing Document Models

In this chapter, we will discuss implementation choices for extracting semantic models from digital documents, for inference on semantic models, and for obtaining and materialising different views on a semantic model.

9.1 Extracting Data Models

We will now discuss three options of obtaining an RDF graph that implements a semantic model, for example a semantic document model, from a digital document. This is an implementation of the mapping $B, D_0 \xrightarrow{T, \mathcal{K}} D$, for a connected base document model B , a semantic document model D , an empty semantic document model D_0 , a set of transformation rules T , and a set of background knowledge \mathcal{K} , as defined in definition 5.1.21 on page 106.

The first option is to write a regular program for the specific purpose, for example in Java. The second option is to specify the extraction logic in a query language like XQuery. The third option is to specify the extraction logic in the form of extraction rules.

Before the actual model extraction process is applied to a digital document, it is useful to preprocess this document. Preprocessing can be used to correct errors or inconsistencies in the original document specification, for example by applying an HTML tidy utility to an HTML document that is not compliant with the HTML standard. It can also be used to simplify or to unify a document's technical structure, for example by using an XSLT stylesheet to harmonise semantically equivalent but syntactically different commands like the HTML `` and `` commands. It can also be used to convert proprietary or very complex document formats into simpler formats, for example converting "old style" Word documents into the new, XML-based Word format.

Remark 9.1.1. *A note on term normalisation (e.g., by stemming or by matching against a list of known terms): normalisation can either be done for every term when a document is first processed, or it can be done on demand, i.e., whenever it is deemed necessary (or not at all).*

The former means that everything, including background knowledge, needs to work with normalised forms of terms all the time. This appears sensible, but experience has shown that in many cases it can be omitted. Term matching (i.e., determining whether two strings represent the same term) can then be done either by a strict lexical match, or by evaluating the Levenshtein distance between two strings. If this distance, that measures the number of different letters in the two strings, is below a certain threshold, say "2", then the strings are considered a match.

While this clearly introduces the possibility of errors (mainly false negatives), stemming may also introduce errors (mainly false positives). Since false positives are worse than false negatives in many applications, it is often not worth the effort to normalise terms.

For example, in document verification false negatives may lead to missing assertions that may lead to errors in the verification. These errors, however, can be traced with relative ease. False positives, on the other hand, almost always lead to incorrect assertions that may lead to errors in the verification. These errors are very hard to trace, because the supposed root cause simply does not exist in the original document.

In the following discussions and examples, as in our implementation, we will not use normalisation for terms. This can, however, be changed easily by calling an appropriate function whenever a term is added to the document model. We will also assume that an instance of a semantic document model with an empty root fragment has already been created and is available to any given program (analogue to D_0 in definition 5.1.21). This is merely a convenience to limit the number of special cases that need to be regarded in an implementation.

9.1.1 Program-based Extraction

Obviously, it is possible to use a regular programming language to specify extraction logic. This usually requires a new program for each document format in each application domain. Within such a program, the structure of the source document, represented as a base document model, can be parsed by using a selector language like XPath, by iterating or recursing along the graph structure like a DOM tree, or by a combination of the two.

Using Java, several extraction programs were created and integrated into the document framework as part of the VERDIKT project. In particular, programs were created for the e-learning formats LMML [Kol08] and ML3, as well as for the DITA [Pre08], DocBook, and HTML formats.

Example 9.1.2 (Java Extraction Program). *The following code fragment is part of the extraction program for HTML documents. It has been shortened and reduced to the core function of parsing HTML files with chapters, definitions, examples, terms, and cross-references across multiple files. Auxiliary functions, error checking, and repetitive code has been excluded. Repetitive code includes code that is similar to the one shown, but extends the functionality to, e.g., other structural elements like sections or illustrations.*

Line 1 defines an auxiliary data structure that keeps track of files that were already parsed, in order to avoid infinite loops.

The primary extraction method starts in line 3. It initialises the `visitedFiles` data structure and starts the extraction with the initial file.

Line 8ff. contains a method that starts the extraction of a single HTML file. It checks if the file has already been parsed, loads and tidies the file's DOM tree, and calls the main parse method. The root fragment of the document model serves as the initial parent parameter for this method.

The main parse method starts in line 15. It is a recursive method that processes the DOM tree of an HTML file node by node, keeping track of the document fragment that corresponds to the actual position in the DOM tree via its parent parameter.

It starts off with a case distinction on the current DOM node: `<div>` nodes are processed in line 16, `` nodes are processed in line 40, and `<a>` nodes are processed in line 48. The method concludes with a recursive call to all child elements of the current DOM node in line 56.

If a `<div>` node has a CSS class value that identifies it as a title node (line 17), its content is annotated as a title to the current document fragment. First, the appropriate name for this

annotation is found in the vocabulary associated with the document model. Then, an annotation is created and added to the fragment.

Other `<div>` nodes represent document fragments (line 23). Therefore, a new fragment is created with an identifier based on the file name and the precise position of the current element in the XML DOM tree (`createFragment(node)`). It is annotated with type information according to the CSS class value of the `<div>` node, e.g., “Chapter” or “Definition”. The new fragment is then added as a child of the current fragment (i.e., it is put into a has-part relationship with the parent fragment), and finally the new fragment replaces the old fragment as the currently regarded fragment in line 38. Note that position annotations as shown in example 8.3.2 are created automatically by the `addChild()` method of the `Fragment` class, based on the order of insertion, i.e., if fragment f_1 is inserted before fragment f_2 , then f_1 will have a lower position number than f_2 .

`` nodes with an appropriate CSS class value contain terms that are relevant for the document. Their text content is added as an annotation to the current document fragment.

References, represented by `<a>` nodes, lead to the parsing of the referenced file. In addition, a new reference annotation is added between the current document fragment and the newly parsed fragment.

```

1 private Set<String> visitedFiles;
2
3 public DocumentModel extract(String filename, DocumentModel model) {
4     visitedFiles = new HashSet<String>();
5     extractFile(filename, model);
6 }
7
8 private void extractFile(String filename, DocumentModel model) {
9     if (visitedFiles.contains(filename))
10        return;
11    Document htmlDocument = HTMLTidy.load(filename);
12    parseHtml(htmlDocument.getDocumentElement(), model.getRoot(), model);
13 }
14
15 private void parseHtml(Element node, Fragment parent, DocumentModel model) {
16     if (node.getNodeName().equals("div")) {
17         if (node.getAttribute("class").equals("title")) {
18             String name = model.getVocabulary().getAnnotationName("title");
19             String value = node.getTextContent();
20             Annotation anno = model.getFragmentFactory()
21                 .createDataAnnotation(name, value);
22             parent.addAnnotation(anno);
23         } else {
24             Fragment fragment = model.getFragmentFactory().createFragment(node);
25             if (node.getAttribute("class").equals("chapter")) {
26                 fragment.setStructuralType("Chapter");
27             } else if (node.getAttribute("class").equals("definition")) {
28                 fragment.setStructuralType("Paragraph");
29                 fragment.setFunctionType("Definition");
30             } else if (node.getAttribute("class").equals("example")) {
31                 fragment.setStructuralType("Paragraph");
32                 fragment.setFunctionType("Example");
33             } else if (node.getAttribute("class").equals("illustration")) {
34                 fragment.setStructuralType("Paragraph");
35                 fragment.setFunctionType("Illustration");
36             }
37             parent.addChild(fragment);
38             parent = fragment;
39         }
40     } else if (node.getNodeName().equals("span")) {

```

```

41     if (node.getAttribute("class").equals("term")) {
42         String name = model.getVocabulary().getAnnotationName("term");
43         String value = node.getTextContent();
44         Annotation anno = model.getFragmentFactory()
45             .createDataAnnotation(name, value);
46         parent.addAnnotation(anno);
47     }
48 } else if (node.getNodeName().equals("a")) {
49     extractFile(node.getAttribute("href"), model);
50     Fragment target = findTarget(node.getAttribute("href"), model);
51     String name = model.getVocabulary().getAnnotationName("reference");
52     Annotation anno = model.getFragmentFactory()
53         .createFragmentAnnotation(name, target);
54     parent.addAnnotation(anno);
55 }
56 for (Element child: getChildNodes(node))
57     parseHtml(child, parent, model);
58 }

```

Consider the following HTML document, consisting of five HTML files. It implements the document from example 4.1.24.

```

1 <!-- index.htm -->
2 <html>
3     <body>
4         <div class="chapter">
5             <div class="title">Introduction</div>
6             <a href="chapter2.htm">Link</a>
7         </div>
8     </body>
9 </html>

1 <!-- chapter2.htm -->
2 <html>
3     <body>
4         <div class="chapter">
5             <div class="title">Chapter 2</div>
6             <div class="definition">
7                 <span class="term">Data Structure</span>
8             </div>
9             <a href="chapter3.htm">Link</a>
10            <a href="chapter4.htm">Link</a>
11        </div>
12    </body>
13 </html>

1 <!-- chapter3.htm -->
2 <html>
3     <body>
4         <div class="chapter">
5             <div class="title">Chapter 3</div>
6             <div class="example">
7                 <span class="term">Data Structure</span>
8             </div>
9             <a href="chapter5.htm">Link</a>
10        </div>

```



```

11     </body>
12 </html>

1 <!-- chapter4.htm -->
2 <html>
3   <body>
4     <div class="chapter">
5       <div class="title">Chapter 4</div>
6       <div class="definition">
7         <span class="term">Binary Tree</span>
8       </div>
9       <div class="illustration">
10        <span class="term">Binary Tree</span>
11      </div>
12      <a href="chapter5.htm">Link</a>
13    </div>
14  </body>
15 </html>

1 <!-- chapter5.htm -->
2 <html>
3   <body>
4     <div class="chapter">
5       <div class="title">Conclusion</div>
6     </div>
7   </body>
8 </html>

```

When the above program is applied to this document, it results in a semantic document model implementation similar to the one shown in example 8.3.2.

While these programs serve their intended purpose, they each had to be created separately with little synergy effects from the existing programs. They are also hard to maintain and update in case of format changes, because a lot of background knowledge about the formats and about the domain of the documents is directly embedded into the Java code.

9.1.2 Query-based Extraction

In order to be used for the extraction of document models, a query language needs to be able to combine query results into a coherent output structure. Otherwise, it can only be used from inside another program, like XPath queries from within a Java extraction program.

The XML query language XQuery possesses this ability. While XQuery can only be applied to XML documents, this is not a debilitating limitation, since most other document formats can be converted to XML in a preprocessing step. XQuery can only produce output that is also in XML format, so an XQuery program will produce an XML version of a document model, which is easy to parse into an RDF/OWL-based document model implementation.

Extraction specifications in XQuery are often very complex, because it is better suited for actual queries, not for constructing structured data based on query results. This makes the syntax somewhat verbose and inconvenient, but still usable. A greater challenge is the need for features that go beyond the scope and power of XQuery, such as external tools, efficient context management, and access to data that is not available in XML format (such as background

knowledge, cf. section 5.1 for a discussion on why background knowledge should not be stored in XML format).

We use the Qexo XQuery engine of the GNU Kawa project, because it not only solves the issues named above, but it also performs well and is freely available. It not only allows for executing XQuery programs from within a Java program like the document framework, it also allows for callbacks to the Java program from within the XQuery program. This feature can be used to gain access to external tools or data.

Syntactically, a static method `method` of a class `class` can be called with `class:method(parameters)`. For non-static methods that can only be called on an instance of a class, this instance must be given as the first parameter `class:method(instance, parameters)`.

Example 9.1.3 (XQuery Extraction Program). *The following code is a simplified version of an XQuery implementation for extracting document models from HTML documents. We will omit auxiliary functions, including the implementation details of Java methods that are called from within the XQuery program. The program assumes that the source files are already in XML format, or more specifically in XHTML format.*

Lines 1 through 4 contain the main program. First, it retrieves a context object from a register, using a Java callback on the `GlobalContextRegister` object. From this context object, the program retrieves the initial file name. Calling the `extractFile` function, the program then parses the given file and returns an XML file.

The initial parse function starts in line 6. It starts by checking if the current file has already been processed. The `visit(String)` method returns false if the named file has already been visited, otherwise it returns true and adds the file name to the list of visited files. The `$context` variable is the context instance on which the method is called.

In line 9, the given XML file is parsed into a DOM tree representation. Then, all chapters are parsed in line 11, and finally all other files referenced from the current one are recursively parsed as well (line 12f.).

Parsing of a chapter is done in line 18ff. First, XML code for a new fragment is generated, with an identifier based on the current XML node. Then, the structural type of the fragment is defined, followed by its title(s). In line 23f., the paragraphs contained in the chapter are parsed. Finally, reference relationships are implemented for all outgoing references in line 25f.

Line 31ff. contains the function for parsing paragraphs. It too starts by creating a new fragment and defining its structural type. It then defines a functional type for the fragment based on the current XML node's class attribute value. It concludes with annotating any relevant terms that occur in the paragraph to the fragment (line 34f.).

```

1 let $context := GlobalContextRegister:getContext("xquery"),
2     $filename := GlobalContext:getFilename($context),
3     $document := local:extractFile($filename, $context)
4 return ( <vdk:document> { $document } </vdk:document> )
5
6 declare function local:extractFile($filename, $context) {
7     if (GlobalContext:visit($context, $filename))
8     then (
9         let $doc := doc($filename)
10        return (
11            local:parseChapter($doc//div[@class="chapter"], $context),
12            for $ref in $doc//a/@href
13            return local:extractFile(data($ref), $context)
14        )
15    ) else ()
16 };
17
18 declare function local:parseChapter($node, $context) {

```

```

19 <vdk:fragment id="{GlobalContext:generateId($context, $node)}"> {
20   <vdk:structuralType name="Chapter"/>,
21   for $title in $node/div[@class="title"]/text()
22   return <vdk:title> { $title } </vdk:title>,
23   for $paragraph in $node/div
24   return local:parseParagraph($paragraph, $context),
25   for $ref in $node//a/@href
26   return <vdk:reference
27     target="{GlobalContext:findTarget($context, $ref)}"/>
28 } </vdk:fragment>
29 };
30
31 declare function local:parseParagraph($node, $context) {
32   <vdk:fragment id="{GlobalContext:generateId($context, $node)}"> {
33     <vdk:structuralType name="Paragraph"/>,
34     if ($node/@class = "definition")
35     then ( <vdk:functionType name="Definition"/> )
36     else (
37       if ($node/@class = "example")
38       then ( <vdk:functionType name="Example"/> )
39       else (
40         if ($node/@class = "illustration")
41         then ( <vdk:functionType name="Illustration"/> )
42         else ( ) ),
43     for $term in $node/span[@class="term"]/text()
44     return <vdk:term> { $term } </vdk:term>
45   } </vdk:fragment>
46 };

```

Applied to the HTML document from example 9.1.2, this XQuery program yields the following output, which can easily be transformed into a semantic document model implementation similar to the one shown in example 8.3.2.

```

1 <vdk:document>
2   <vdk:fragment id="ID1">
3     <vdk:structuralType name="Chapter"/>
4     <vdk:title>Introduction</vdk:title>
5     <vdk:reference target="ID2"/>
6   </vdk:fragment>
7   <vdk:fragment id="ID2">
8     <vdk:structuralType name="Chapter"/>
9     <vdk:title>Chapter 2</vdk:title>
10    <vdk:fragment id="ID21">
11      <vdk:structuralType name="Paragraph"/>
12      <vdk:functionType name="Definition"/>
13      <vdk:term>Data Structure</vdk:term>
14    </vdk:fragment>
15    <vdk:reference target="ID3"/>
16    <vdk:reference target="ID4"/>
17  </vdk:fragment>
18  <vdk:fragment id="ID3">
19    <vdk:structuralType name="Chapter"/>
20    <vdk:title>Chapter 3</vdk:title>
21    <vdk:fragment id="ID31">
22      <vdk:structuralType name="Paragraph"/>
23      <vdk:functionType name="Example"/>
24      <vdk:term>Data Structure</vdk:term>
25    </vdk:fragment>
26    <vdk:reference target="ID5"/>
27  </vdk:fragment>

```

```

28 <vdk:fragment id="ID4">
29   <vdk:structuralType name="Chapter"/>
30   <vdk:title>Chapter 4</vdk:title>
31   <vdk:fragment id="ID41">
32     <vdk:structuralType name="Paragraph"/>
33     <vdk:functionType name="Definition"/>
34     <vdk:term>Binary Tree</vdk:term>
35   </vdk:fragment>
36   <vdk:fragment id="ID42">
37     <vdk:structuralType name="Paragraph"/>
38     <vdk:functionType name="Illustration"/>
39     <vdk:term>Binary Tree</vdk:term>
40   </vdk:fragment>
41   <vdk:reference target="ID5"/>
42 </vdk:fragment>
43 <vdk:fragment id="ID5">
44   <vdk:structuralType name="Chapter"/>
45   <vdk:title>Conclusion</vdk:title>
46 </vdk:fragment>
47 </vdk:document>

```

While XQuery programs can be created with reasonable effort for small and simple document types, they become increasingly cumbersome and hard to write and maintain for more complex document formats. Especially troublesome are document commands that do not adhere to the XML tree structure, for example cross references. This is why, in example 9.1.3, references had to be addressed in two different places in the program, while in example 9.1.2 this could be done in one place.

9.1.3 Rule-based Extraction

For implementing a rule-based specification of the extraction logic, we will use JBoss Drools (cf. section 3.4). Syntactically, a Drools rule starts with the keyword **rule** followed by an arbitrary but unique name. The rule head starts with the keyword **when**, the conclusion starts with the keyword **then** and ends with the keyword **end**. Drools uses a fact base of objects against which the rule heads are matched, and matching rules are executed against the objects that they match against. If multiple rules match the same object, each of these rules is executed against this object in the reverse order in which they are defined, i.e., the last rule is executed first. Using the keywords **insert**, **retract**, and **modify**, objects can be added to the fact base, removed from the fact base, or be changed within the fact base. It is possible to define auxiliary functions with the **function** keyword.

We will make use of the `GlobalContext` and `LocalContext` interfaces of the document framework, or more specifically of their `DroolsGlobalContext` and `DroolsLocalContext` implementations. The global context keeps track of the global state of the extraction process, e.g., which files have already been processed, and of the document model that is created in the process. It also provides convenience functions like loading external files or creating new fragments for the document model. The local context keeps track of the original document's structure, in particular of the current XML node that is processed and of the hierarchy of nodes that came before. While the Java implementation of XML DOM nodes provide similar capabilities, thus making it possible to use XML elements directly instead of wrapping them into local contexts, the `DroolsLocalContext` implementation provides several convenience functions that allow for writing simpler and terser code.

We will start with a small example with a similar functionality to that of examples 9.1.2 and 9.1.3, which we will expand on afterwards.

Example 9.1.4 (Drools Extraction Rules (1)). *The following rules are a simplified version of a naive rule-based implementation for extracting document models from HTML documents (cf. example 5.1.23). As in previous examples, we will omit auxiliary functions. It is assumed that when evaluation of the rules begins, the initial HTML file has already been parsed into a DOM tree and its root elements has been wrapped into a local context and added to the Drools fact base. An instance of a global context must also be in the fact base.*

The first rule in line 1ff. matches any combination of global and local context. For every child node of the node wrapped in the given local context, it creates a new local context, puts the child node inside, sets it as a child of the current local context, and adds it to the fact base. This recursively adds all elements of a DOM tree to the fact base, starting with the root element that was put into the fact base initially.

Note that since the order in which nodes are processed determines the order in which new fragments are inserted into the document model, and that the insertion order determines the relative position numbers annotated to fragments, and thus the relative position of child fragments. For example, if two nodes n_1 and n_2 each lead to the creation of a new fragment, f_1 and f_2 , respectively, then these fragments will be added as child nodes of their parent fragment in the order in which n_1 and n_2 are processed. Drools matches objects in the fact base against rules in LIFO (last in, first out) order, i.e., the last object to be inserted is the first to be matched. So to ensure the correct order of document fragments, XML nodes need to be inserted into the fact base in reverse order! This is accomplished by the `getChildNodes()` auxiliary function, which returns the list of child nodes in reverse document order.

The next rule in line 14ff. matches any local context with a `<div>` element and a class attribute value of “chapter”. It first creates a new fragment to represent this chapter, with an identifier based on the current node. It then sets the fragment’s structural type and adds it to the local context. Finally, in line 24, the fragment is added to the document model. The parent fragment is determined by the `getParentFragment()` method by recursively checking the hierarchy of local contexts, starting with the current local context and working its way up. If one of them has a fragment attached to it, this fragment is returned. If none of them have an attached fragment, then the root fragment of the document model is returned.

Line 27ff. holds a rule that matches titles defined in the HTML document. Using the auxiliary `annotate()` function, this title is simply annotated to the current fragment, i.e., to the fragment that was created for the lowest ancestor node of the current node. The `annotate()` function (not shown) validates its parameters and uses the document model’s vocabulary to find the proper relation name to use (similar to line 17ff. in example 9.1.2).

The following three rules (in line 37ff., 51ff., and line 65ff., respectively) match against nodes that represent definitions, examples, or illustrations. They each create a new fragment to represent the paragraph, add both a structural type (“paragraph”) and a function type (“definition”, “example”, or “illustration”, respectively), and insert it into the hierarchy of local contexts.

The term rule starting in line 79 matches against `` elements used to enclose terms. It annotates the normalised term to the current fragment.

Finally, line 89ff. contains the reference rule that matches against `<a>` elements. It determines the file name for the reference and parses the file into a DOM tree. If the file has not yet been processed (determined using `GlobalContext`’s `visit()` method, see above), the file’s root element is wrapped into a local context and added to the fact base for processing. In any case, a new fragment is created for the chapter contained in the file, and a new reference relationship is added from the current fragment to the new one (line 101ff.). Since the new fragment is created with the same identifier that is used when the “actual” fragment for this chapter is created with

the chapter rule, these fragments are considered equivalent by the framework, so that the reference really points to the correct fragment.

```

1 rule "Element Recursion"
2 when
3     $context: DroolsGlobalContext()
4     $lcontext: DroolsLocalContext()
5 then
6     for (Node $child: getChildNodes($lcontext.getNode())) {
7         LocalContext $childcontext = $context.createLocalContext();
8         $childcontext.setNode($child);
9         $childcontext.setParent($lcontext);
10        insert($childcontext);
11    }
12 end
13
14 rule "Chapter"
15 when
16     $context: DroolsGlobalContext()
17     $lcontext: DroolsLocalContext(node.localName == "div"
18         && attributes["class"] == "chapter")
19 then
20     Fragment $fragment =
21         $context.createFragment($lcontext.getNode());
22     $fragment.addStructuralType("Chapter");
23     $lcontext.setFragment($fragment);
24     $context.getParentFragment($lcontext).addChild($fragment);
25 end
26
27 rule "Title"
28 when
29     $context: DroolsGlobalContext()
30     $lcontext: DroolsLocalContext(node.localName == "div"
31         && attributes["class"] == "title")
32 then
33     annotate($context.getParentFragment($lcontext), "title",
34         $lcontext.getTextContent(), $context);
35 end
36
37 rule "Definition"
38 when
39     $context: DroolsGlobalContext()
40     $lcontext: DroolsLocalContext(node.localName == "div"
41         && attributes["class"] == "definition")
42 then
43     Fragment $fragment =
44         $context.createFragment($lcontext.getNode());
45     $fragment.addType("Paragraph");
46     $fragment.addType("Definition");
47     $lcontext.setFragment($fragment);
48     $context.getParentFragment($lcontext).addChild($fragment);
49 end
50
51 rule "Example"
52 when
53     $context: DroolsGlobalContext()
54     $lcontext: DroolsLocalContext(node.localName == "div"
55         && attributes["class"] == "example")
56 then
57     Fragment $fragment =
58         $context.createFragment($lcontext.getNode());
59     $fragment.addType("Paragraph");

```

```

60     $fragment.addType("Example");
61     $lcontext.setFragment($fragment);
62     $context.getParentFragment($lcontext).addChild($fragment);
63 end
64
65 rule "Illustration"
66 when
67     $context: DroolsGlobalContext()
68     $lcontext: DroolsLocalContext(node.localName == "div"
69         && attributes["class"] == "illustration")
70 then
71     Fragment $fragment =
72         $context.createFragment($lcontext.getNode());
73     $fragment.addType("Paragraph");
74     $fragment.addType("Illustration");
75     $lcontext.setFragment($fragment);
76     $context.getParentFragment($lcontext).addChild($fragment);
77 end
78
79 rule "Term"
80 when
81     $context: DroolsGlobalContext()
82     $lcontext: DroolsLocalContext(node.localName == "span"
83         && attributes["class"] == "term")
84 then
85     String $value = $lcontext.getTextContent();
86     annotate($context.getParentFragment($lcontext), "term", $value, $context);
87 end
88
89 rule "Reference"
90 when
91     $context: DroolsGlobalContext()
92     $lcontext: DroolsLocalContext(node.localName == "a")
93 then
94     String $filename = $lcontext.getAttributes().get("href");
95     Document $doc = $context.loadDocument($filename);
96     if ($context.visit($filename) {
97         LocalContext $refcontext = $context.createLocalContext();
98         $refcontext.setNode($doc.getDocumentElement());
99         insert($refcontext);
100    }
101     Element $chapter =
102         $context.evaluateXPath("//div[@class='chapter']", $doc);
103     Fragment $target = $context.createFragment($chapter);
104     $context.getParentFragment($lcontext).addReference($target);
105 end

```

Remark 9.1.5. *Evaluation data has shown that the use of auxiliary functions in combination with rules can slow down the compilation of a rule set considerably, and to a certain degree also its execution [Ste10]. This is a peculiarity of the Drools implementation. In practice, it can be avoided easily by moving the code of auxiliary functions directly into the rules. Since this reduces readability, however, we will continue using auxiliary functions in the code examples shown here.*

Example 9.1.4 has shown how extraction rules can be used in a similar manner to Java or XQuery programs. But that does not make them particularly useful yet. It is only when they go beyond the advantages (and shortcomings) of the other approaches that extraction rules become the approach of choice. The two primary points where extraction rules surpass the other options are *maintainability* and *transferability*.

Rule-based specifications are often easier to maintain, because the rule format encourages a stricter structuring and a more methodical approach than a regular programming language does. Obviously, a well structured extraction specification is easier to understand and to modify than a badly structured one. It is also possible to remove much of the format-specific or domain-specific information from the extraction logic in the form of background knowledge (see below), which means that the extraction rules have to be modified only rarely. Instead, the formalised background knowledge can be modified with the appropriate tools.

We will now show that it is possible to easily transfer a set of extraction rules to a new document format or to a new application domain (or both) if certain conditions are met. The first condition is that (almost) all format- and domain-specific knowledge is formalised as background knowledge and used in the rules. The second condition is that some format-specific knowledge that is hard to formalise in ontological form is instead encapsulated in a handful of auxiliary functions that are also used in the extraction rules. In particular, the following two functions need to be implemented based on the specifics of the file format:

- ▶ `getChildElements(LocalContext)`: `List`. This function returns a list of `LocalContext` objects that wrap the child elements of the given local context in reverse document order. For non-hierarchical document formats, it only has to be called once to return all elements of the document.
- ▶ `getIndicators(LocalContext)`: `String`. This function determines the formatting options of the element wrapped in the given local context, and/or its textual content. If the formatting is based on one or more named styles (like CSS classes or Word templates), then a list of these names is returned. If the formatting consists of a set of unnamed formatting options (like “bold” or “italic”), then a list of appropriate names for these options is returned. If the formatting consists of both named styles and unnamed options, a mixture of style names and option names is returned. If the element has textual content, any relevant terms used are returned as well.

The returned string may also contain information like the name of the XML element wrapped into the given context. If the result consists of multiple data points, they should be placed in a well-defined order (e.g., in alphabetical order) and be clearly separated (e.g., by whitespace). The particulars of how the result is ordered and separated are not important beyond the requirement that they must be consistent with the available background knowledge (see below).

Apart from these auxiliary functions, the extraction specification consists of a set of rules that remain largely constant, of some additional rules that capture special cases, and of background knowledge in the form of ontologies. This is illustrated in descriptions 9.1.6 and 9.1.8.

Before we explore the fully generic case of extracting an implementation of a semantic document model from a digital document seen as a base document model, we will regard a case that is slightly easier to handle. While in general a source document is just a graph of media objects, there are source documents with a more complex structure, namely with a hierarchical relationship between media objects in addition to the simple successor relationships. This hierarchy can be used in the extraction specification. Source documents in XML format often (but not always) contain such a hierarchy, as does the document shown in example 9.1.2, where for example elements that represent paragraphs are contained within elements that represent chapters.

We will now extend the rules from example 9.1.4 to reflect the necessary changes discussed above.

Description 9.1.6 (Drools Extraction Rules (Recursive)). *First, the two required auxiliary functions are defined. In line 1ff., the child elements of the current XML DOM element are listed and returned in reverse document order. In line 10ff., the formatting styles of an XML element are aggregated, including the element’s name and CSS class (if one exists). The underscore (“_”) is used as a separation character for technical reasons: the returned style information can now be used directly to reference an individual name in a SKOS ontology, which may not contain whitespace or many other typical separation characters like commas or semicolons. No keywords are used in this rule set, so the `getIndicators()` function does not return any terminology used. In line 17ff., an identifier is calculated based on a given XML element. The location of this element is specified in the background knowledge.*

The element recursion rule (line 28ff.) is virtually unchanged from example 9.1.4, except that it now uses the new `getChildElements()` function.

Line 106ff. contains the fragment rule that replaces the more specific rules for chapters, definitions and so on. It is moved almost to the end of the rule file to make sure that it is triggered first if multiple rules match one element. Since it creates the fragment that other rules (in particular the data annotation rule, see below) use, it must be executed before them.

The rule matches all XML elements whose formatting style indicate that a new fragment should be created for them. The head of the rule makes use of background knowledge. It accesses a knowledge base named “structure” and retrieves a term group (i.e., a list of terms) named “fragment”. In this case, the term group is implemented as an SKOS concept with multiple labels (see below). The entries of this list are matched against the formatting styles of the current XML element. Note that JBoss Drools provides an abbreviated syntax in rule heads: attributes can be accessed by their name instead of using the appropriate `get()` method, and entries in a key-value map can be accessed as `map[key]`. Using the generic `getIndicators()` function also allows this rule to identify fragments based on keywords, if the background knowledge contains any.

*For XML elements that match the rule’s condition, a new fragment is created. Then a knowledge base of mappings, named “styleFragments”, is accessed. All mappings that correspond to the formatting styles of the current element are retrieved and their target types are annotated to the fragment. For example, an XML element `<div class="definition">` results in a formatting style as returned by the `getIndicators()` function of “div_definition”. There are two mappings with an appropriate source in the “styleFragments” knowledge base, namely one pointing to the concept *Paragraph* and one pointing to the concept *Definition*. Both types are annotated to the newly created fragment.*

Another set of mappings, “keywordFragment”, is also used. It can contain a mapping from certain keywords onto structural types, where for each keyword found in the textual content of the current node, the appropriate type is annotated to the new fragment. However, in this instance, the “keywordFragment” knowledge base is empty, because the semantic document model can be created using formatting information alone.

Similar to the fragment rule, the data annotation rule in line 41ff. matches against XML elements with a specific formatting, which is specified in the background knowledge. If a match is found, a list of data mappings named “data” is retrieved. Data mappings consist of a source that specifies the location of a datum in the source document and of a target that specifies how this datum should be annotated to the document model. The location specified in the source is given relative to the current location in the document, for example relative to the currently regarded XML element. The target consists of the annotation name, e.g., the predicate, with which the data point is annotated to the current document fragment. If no data is found at the source location, no annotation is made.

For each of these data mappings, a value is retrieved from the XML document by evaluating the XPath expression specified as the source of the mapping. If a value could be found, it is then

annotated to the current fragment, using the mapping's specified target as the annotation name. For example, for an XML element ``, the source of the mapping points to the text content of said element, and the target specifies "term" as the annotation name.

In line 58ff. the reference rule matches all XML elements that are identified as references by the background knowledge. The actual target location of the reference is again obtained by using data mappings. Such a target may consist of a file name, a file name and an anchor, or just an anchor. An anchor specifies a specific location within a file. If no file name is given, the anchor points to a position within the current file. For example, the reference `` points to an XML element within the "chapter3.html" file with an identifier named "exa". The auxiliary functions `getFilename()` and `getAnchor()` return the appropriate parts of a reference location, or the empty string if this part is undefined.

If a file is referenced, it is parsed into an XML DOM tree. If the file still needs to be processed, then its root element is wrapped into a local context instance and added to the Drools fact base for processing. If an anchor is specified, then the appropriate element is located in the DOM tree using the "idref" background knowledge. A new fragment with an identifier based on the XML root element of the new file (or the element specified by the reference's anchor) is created and used as the target for a new reference relationship. The name of the reference annotation is determined by the target attribute of the data mapping.

Finally, the new file root rule (line 130ff.) is necessary to establish an end point for references: in general, it is not always possible to determine the exact position in a file that a reference points to, if only the file itself is specified as the target of the reference (for example as in ``). In example 9.1.4, a specific `<div>` element within the specified file was used as an end point for a reference, but clearly this is only possible in special cases. Therefore, a new fragment is created for each file (or, more specifically, for the root element of each file, which is recognised as the only element without a parent). This fragment then serves as an anchor for all fragments created for this particular file, i.e., new fragments are added as children of the file fragment. This rule is placed last to ensure that it is triggered first if multiple rules match a single element.

```

1 function List getChildElements(LocalContext lcontext) {
2     List result = new ArrayList();
3     NodeList children = lcontext.getNode().getChildNodes();
4     for (int i=children.getLength()-1;i>=0;i--)
5         if (children.item(i) instanceof Element)
6             result.add(children.item(i));
7     return result;
8 }
9
10 function String getIndicators(LocalContext lcontext) {
11     String result = lcontext.getNode().getLocalName();
12     if (lcontext.hasAttribute("class"))
13         result += "_" + lcontext.getAttributes().get("class");
14     return result;
15 }
16
17 function int getId(LocalContext lcontext, GlobalContext context) {
18     List<DataMapping> mappings = context.getMappings("id");
19     for (DataMapping mapping: mappings) {
20         Object obj = context.evaluateXPath(mapping.getSource(),
21             lcontext.getNode());
22         if (obj != null)
23             return context.getId(obj);
24     }
25     return 0;
26 }

```

```

27
28 rule "Element Recursion"
29 when
30     $context: DroolsGlobalContext()
31     $lcontext: DroolsLocalContext()
32 then
33     for (Node $child: getChildElements($lcontext)) {
34         LocalContext $childcontext = $context.createLocalContext();
35         $childcontext.setNode($child);
36         $childcontext.setParent($lcontext);
37         insert($childcontext);
38     }
39 end
40
41 rule "Data Annotation"
42 when
43     $context: DroolsGlobalContext()
44     $lcontext: DroolsLocalContext(
45         $context.knowledgebases["structure"].termGroups["data"]
46         contains getIndicators($this))
47 then
48     List<DataMapping> $mappings = $context.getMappings("data");
49     for (DataMapping $mapping: $mappings) {
50         String $value = $context.evaluateXPath($mapping.getSource(),
51         $lcontext.getNode());
52         if (!$value.equals(""))
53             annotate($context.getParentFragment($lcontext),
54             $mapping.getTarget(), $value, $context);
55     }
56 end
57
58 rule "Reference"
59 when
60     $context: DroolsGlobalContext()
61     $lcontext: DroolsLocalContext(
62         $context.knowledgebases["structure"].termGroups["reference"]
63         contains getIndicators($this))
64 then
65     List<DataMapping> $mappings = $context.getMappings("reference");
66     for (DataMapping $mapping: $mappings) {
67         String $reference = $context.evaluateXPath(
68             $mapping.getSource(), $lcontext.getNode());
69         String $filename = getFilename($reference, $context);
70         String $anchor = getAnchor($reference, $context);
71         Node $node = null;
72         if (!$filename.equals("")) {
73             $node = $context.loadDocument($filename)
74                 .getDocumentElement();
75             if ($context.visit($filename) {
76                 LocalContext $rcontext =
77                     $context.createLocalContext();
78                 $rcontext.setNode($node);
79                 insert($rcontext);
80             }
81         }
82         if (!$anchor.equals("")) {
83             for (DataMapping $idref:
84                 $context.getMappings("idref")) {
85                 String $xpath = $idref.getSource()
86                     .replace("?", $anchor);
87                 Object $obj = $context.evaluateXPath($xpath,
88                 $lcontext.getNode());

```

```

89         if ($obj != null) {
90             $node = $obj;
91             break;
92         }
93     }
94 }
95 if ($node != null) {
96     LocalContext $rcontext = $context.createLocalContext();
97     $rcontext.setNode($node);
98     Fragment $target = $context.createFragment(
99         getId($rcontext, $context));
100    annotate($context.getParentFragment($lcontext),
101        $mapping.getTarget(), $target, $context);
102 }
103 }
104 end
105
106 rule "Fragment"
107 when
108     $context: DroolsGlobalContext()
109     $lcontext: DroolsLocalContext(
110         $context.knowledgebases["structure"].termGroups["fragment"]
111         contains getIndicators($this))
112 then
113     Fragment $fragment =
114         $context.createFragment(getId($lcontext, $context));
115     String $ind getIndicators($lcontext);
116     List<DataMapping> $mappings;
117     $mappings = $context.getMappings("styleFragment");
118     for (DataMapping $mapping: $mappings)
119         if ($ind.equals($mapping.getSource()))
120             $fragment.addType($mapping.getTarget());
121     String $text = $lcontext.getNode().getTextContent();
122     $mappings = $context.getMappings("keywordFragment");
123     for (DataMapping $mapping: $mappings)
124         if ($text.contains($mapping.getSource()))
125             $fragment.addType($mapping.getTarget());
126     $lcontext.setFragment($fragment);
127     $context.getParentFragment($lcontext).addChild($fragment);
128 end
129
130 rule "File Root"
131 when
132     $context: DroolsGlobalContext()
133     $lcontext: DroolsLocalContext(parent == null)
134 then
135     Fragment $fragment =
136         $context.createFragment(getId($lcontext, $context));
137     $fragment.setType("File");
138     $lcontext.setFragment($fragment);
139     $context.getModel().getRoot().addChild($fragment);
140 end

```

Note that this more generic approach scales better than the one described in example 9.1.4. JBoss Drools uses the RETE algorithm [Doo95] to check which rules to apply to which objects in the fact base. Since the RETE algorithm needs to check every rule for possible matches, reducing the number of rules by using background knowledge in the premise of rules increases runtime performance, especially since an efficient implementation of the `contains` operation runs in constant time $O(1)$. The background knowledge makes the (number of) rules mostly independent

of the structure and content of the source document.

Example 9.1.7 (Drools Extraction Rules (2)). Consider the following background knowledge, encoded in SKOS format and presented in Turtle syntax, that is available to the extraction rules. Note that the concrete format of the background knowledge is immaterial, as long as it is supported by the framework.

The knowledge base “structure” contains information about which XML elements, formatting styles and keywords should lead to the creation of fragments (aggregated under the SKOS concept `vdk:fragment`), which elements and styles should lead to the annotation of data to existing fragments (`vdk:data`), and which should lead to the creation of new reference relationships (`vdk:reference`). In this instance, it does not contain any keywords, because the formatting styles are sufficient in this example.

```

vdk:fragment    rdf:type      skos:Concept ;
                skos:prefLabel "div_chapter" ;
                skos:altLabel  "div_definition" ;
                skos:altLabel  "div_example" ;
                skos:altLabel  "div_illustration" .
vdk:data        rdf:type      skos:Concept ;
                skos:prefLabel "div_title" ;
                skos:altLabel  "span_term" .
vdk:reference   rdf:type      skos:Concept ;
                skos:prefLabel "a" .

```

Additionally, consider the following background knowledge, encoded as source-target mappings, that is also available to the extraction rules.

The “id” mappings contain information about how a unique id can be found for an XML node. In this case, the node itself is used:

source	target
.	id

The “idref” mappings contain information about how a reference can be resolved within a document. The “?” is replaced by the actual reference string:

source	target
//*[@id="?"]	idref

The “data” mappings contain information about the specific location of the values of titles and terms:

source	target
.[@class="title"]/text()	title
.[@class="term"]/text()	term

The “reference” mappings contain information about the specific location of the target of a reference:

source	target
./@href	reference

The “styleFragment” mappings contain information about which XML elements and formatting styles indicate specific types or classes in the semantic model:

<i>source</i>	<i>target</i>
<i>div_chapter</i>	<i>Chapter</i>
<i>div_definition</i>	<i>Definition</i>
<i>div_definition</i>	<i>Paragraph</i>
<i>div_example</i>	<i>Example</i>
<i>div_example</i>	<i>Paragraph</i>
<i>div_illustration</i>	<i>Illustration</i>
<i>div_illustration</i>	<i>Paragraph</i>

The “keywordFragment” mappings contain information about which keywords indicate specific types or classes in the semantic model. It is empty in this instance:

<i>source</i>	<i>target</i>
---------------	---------------

Applied to the document from example 9.1.2, the extraction process returns an implementation of a semantic document model that is slightly different from the one shown in example 8.3.2, because it contains one additional fragment for each of the five files. Figure 9.1 shows part of the hierarchical source files and the corresponding part of the semantic document model, whose implementation is extracted from the files. The identifiers shown are compatible with the notation adopted in example 9.1.10 to increase comparability.

In addition to being somewhat easier to process, a hierarchical file format may also hold more structural information about the document than a flat format, because it not only clearly defines where a particular section of a document starts, but also where it ends. In many flat or semi-flat file formats like L^AT_EX or (sometimes) HTML, it is often unclear where a structural entity ends.

Consider, for example, the document from example 9.1.2. Through the format’s structure it is clear the the references (the <a> elements) belong to their enclosing <div> element, which represent chapters. In chapters 2 through 4, it is also clear that the references do not belong to any of the definitions or other paragraphs. In a flat file format, however, the latter would not be clear at all, since without a dedicated end command for structural elements like the L^AT_EX `\chapter{}` command, the affiliation of the references is ambiguous. This can be seen in example 9.1.10 below.

While it is possible to process documents encoded in a hierarchical file format using the more generic rules for flat file formats, we do not recommend it. It requires end nodes to be inserted into the flat graph to mark the range of hierarchical elements, thereby increasing the number of nodes to process. This also increases the size of the background knowledge because it must then not only contain indicators for when an elements represents the start of a new fragments, but it must also contain indicators for when an element represents the end of a fragment. It is therefore not only more convenient, but also more efficient to process hierarchical file formats separately.

As stated above, description 9.1.6 shows rules for the special case of a hierarchically structured source format. This simplifies the extraction process, because there is usually no need to keep track of the document model’s structure. The assumption here is that the hierarchical structure of the source format and the hierarchical structure of the document model do at least partially align. More precisely, if elements e_1 and e_2 from the source document lead to the creation of fragments f_1 and f_2 in the document model, respectively, and if e_2 is a sub-element of e_1 w.r.t. the structural hierarchy, then f_2 is a sub-fragment of f_1 .

If this assumption does not hold, or if the source document is not hierarchically structured to begin with, then a more generic set of extraction rules is required.

To obtain a graph of media objects, a document’s files must be parsed file-by-file, observing the order of media objects as defined in the files. If this “physical” order is of no consequence for the resulting document model, i.e., if the structural order is defined by reference commands,

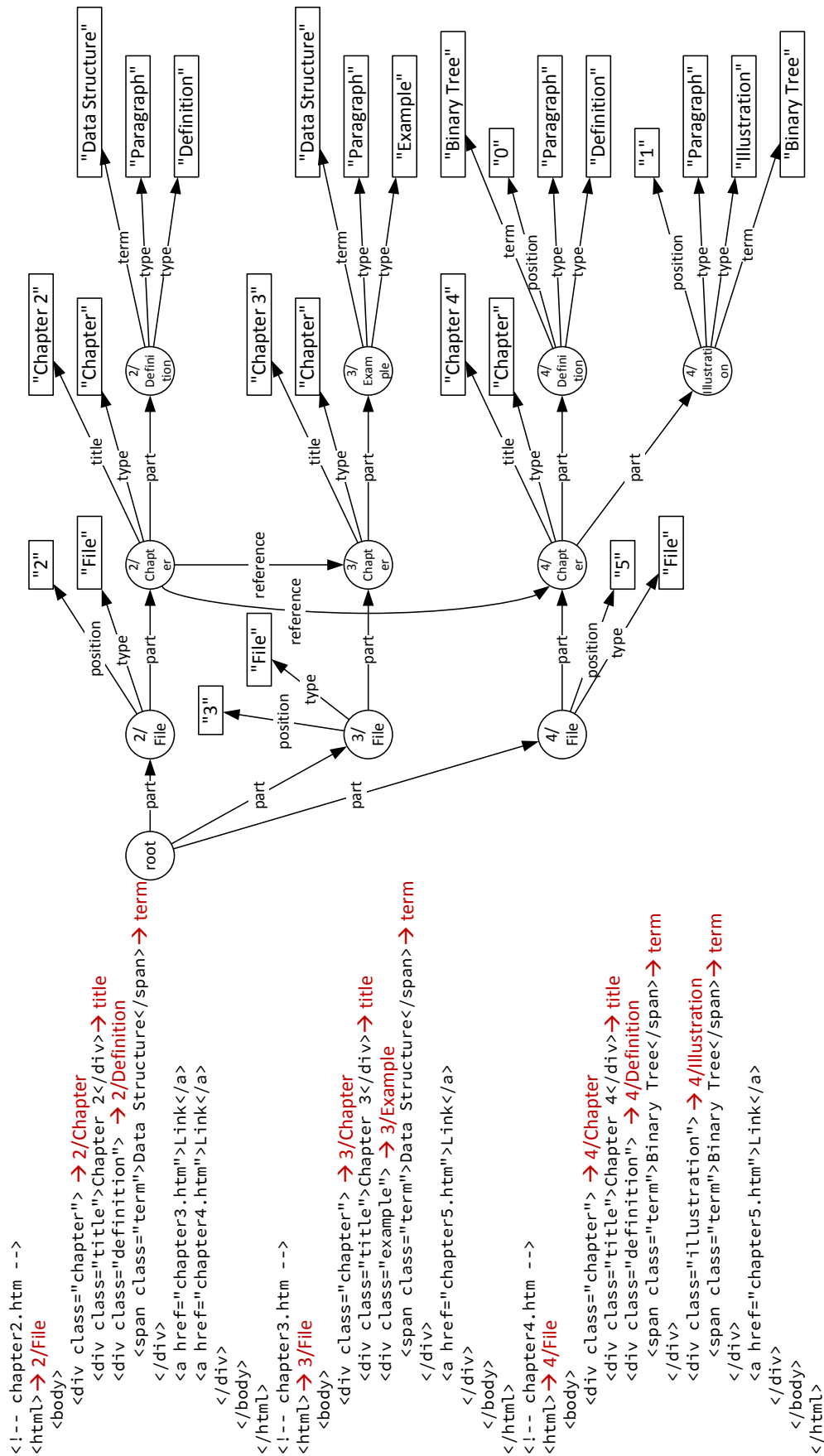


Figure 9.1: Hierarchical source document and semantic document model

then adhering to the physical order will still do no harm. If the structural order is in full or in part implied by the physical order, then observing it becomes necessary.

However, with the rules as shown in description 9.1.6, this order is disturbed whenever a reference is encountered because the referenced target is then processed before the local processing is resumed. In other words, the nodes of another file – albeit in their correct order – are inserted into the fact base in between the nodes of the current file.

For the more generic rules, this requires a small change in the implementation of the rules that process references. The rules for listing elements and for processing references are now executed with a higher priority than other rules. This leads to all nodes being inserted into the fact base in the correct order, before any other rule is executed (see below). In other words, the entire base document model is loaded into the fact base before the transformation starts.

Another approach is to proceed as before, but to include information about the source file of a node when determining its location in the document's structure. As will be shown below, for flat file formats it is necessary to keep track of the document's structure by some means. For each newly processed node, it must be determined if the node indicates a new sub-fragment in the structure, if the node belongs to the current fragment, or if the node belongs to an ancestor of the current fragment. If the file that the node belongs to is regarded in this determination, then it is possible to infer the correct document structure even if the nodes are not strictly grouped and processed by file.

However, this alternate approach requires a far more complex procedure for determining the correct fragment for a node. It is also error prone because include commands like the `LATEX \input{}` command result in different source files for nodes, but should not change their location in the document model. We therefore recommend using the first approach to avoid unnecessary complexity and potential for errors.

Description 9.1.8 (Drools Extraction Rules (Generic)). *The generic extraction rules follow a two-step approach. In the first step, all relevant elements of the source files are added to the fact base in the correct order. This is akin to creating a base document model as described in section 4.1. In the second step, a semantic document model is created based on this fact base. This is akin to using transformation rules to obtain a semantic document model from a base document model as described in section 5.1. The rules for the second step are an implementation of the transformation rules specified in description 5.1.24.*

The auxiliary function `getChildElements()` (line 13ff.) is now defined recursively, using the `getChildElementsRecursive()` convenience function (line 1ff.). It returns all elements of a DOM tree in reverse document order and is called by the element recursion rule in line 35ff. This rule now only matches against local contexts with no parent element, i.e., against the root elements of files. Combined with the rule's high priority as indicated by the `salience` keyword, this ensures that the elements of an entire file are inserted at once and in the correct order into the fact base.

In the first line after the insertion block of the element recursion rule, the current local context is modified (via the `modify` keyword) to re-insert it into the fact base. This is necessary to make sure that it is the first element of its file to be processed. Setting the `recursed` attribute to true ensures that it is not matched against the element recursion rule again. Then, a new fragment is created and inserted into the document model. This is a place holder for the file fragment created later (see below). It is inserted here to ensure that they are in the correct order in the document model, namely in the order in which the files are referenced from one to another.

The `getIndicators()` and `getId()` functions in line 17ff. and in line 24ff. remain unchanged.

A new reference insertion rule has been defined (line 53ff.). It takes on half of the functionality

of the original reference rule: the insertion of the root element of referenced files into the fact base. It is triggered with a higher priority (`salience 10`) to ensure that references are processed immediately after all elements of a file have been inserted into the fact base. The even higher priority of the element recursion rule (`salience 20`) then ensures that the newly inserted file root element is processed right away.

The first part of the file root rule (line 175ff.) is also the same as before. The file fragment is created with the same identifier as in the element recursion rule, so they are mapped onto each other. In the last line, the new fragment is now pushed on top of a stack that keeps track of the document's structure. This has become necessary because we cannot rely on the source files' structure to guide the document model's structure.

The most notable changes were made to the fragment rule in line 136ff. The first part remains the same, but when it comes to inserting the newly created fragment into the correct place in the document model's structure, the rule becomes more complex. First, a new knowledge base named "documentStructure" is accessed, which contains information about the document's structural hierarchy. It is typically derived from the `hasNarrower` role in the semantic document model that defines a hierarchy of structural types.

Then, the types of the new fragment are compared to the types of the fragment that is currently on top of the stack: if, according to the knowledge base, one of the "new" types is broader or equal (`getBroaderSelf()`) to one of the types from the stack, the top stack element is removed. This is repeated until the new types are narrower than the types from the stack or until no relationship, neither narrower nor broader nor equals, is known between them. Finally, the new fragment is added as a child to the top stack element, and is then pushed onto the stack itself.

For example, if the top fragment on the stack has the type "paragraph" and the newly created fragment has the type "section", which is known to be broader than "paragraph", then the top fragment is removed from the stack. If the new top fragment has the type "chapter", which is known to be broader than "section", the new fragment is inserted as a child of the top stack fragment.

If there is a contradiction, i.e., if a new fragment has two types, one of which is broader than the type of the stack fragment, and the other is narrower than the stack fragment, then the broader type wins out. Thus, conflicts are resolved in favour of inserting fragments higher up in the document structure. Fragments without type information are inserted at the current position.

Both the data rule (line 77ff.) and the reference rule (line 94ff.) have remained largely unchanged, except that both rules now get the current document fragment from the stack instead of from the context. In addition, the reference rule now only adds the appropriate reference relationship to the document model without modifying the fact base, since the fact base has already been updated by the reference insertion rule above.

This split is necessary because new files need to be inserted into the fact base as early as possible to ensure the correct processing order. However, at this time the appropriate structure of the document model is not yet ready. In particular, the starting fragment for the reference has not been created yet and there is not enough context information yet to create it. Therefore, the processing of references is split into two parts, the first part to update the fact base and the second to update the document model.

```

1 function List getChildElementsRecursive(Element node) {
2     List result = new ArrayList();
3     NodeList children = node.getChildNodes();
4     for (int i=children.getLength()-1;i>=0;i--)
5         if (children.item(i) instanceof Element) {
6             Element child = (Element) children.item(i);
7             result.addAll(getChildElementsRecursive(child));
8             result.add(child);

```

```

9      }
10     return result;
11 }
12
13 function List getChildElements(LocalContext lcontext) {
14     return getChildElementsRecursive(lcontext.getNode());
15 }
16
17 function String getIndicators(LocalContext lcontext) {
18     String result = lcontext.getNode().getLocalName();
19     if (lcontext.hasAttribute("class"))
20         result += "_" + lcontext.getAttributes().get("class");
21     return result;
22 }
23
24 function int getId(LocalContext lcontext, GlobalContext context) {
25     List<DataMapping> mappings = context.getMappings("id");
26     for (DataMapping mapping: mappings) {
27         Object obj = context.evaluateXPath(mapping.getSource(),
28             lcontext.getNode());
29         if (obj != null)
30             return context.getId(obj);
31     }
32     return 0;
33 }
34
35 rule "Element Recursion"
36 salience 20
37 when
38     $context: DroolsGlobalContext()
39     $lcontext: DroolsLocalContext(parent == null && recursed == false)
40 then
41     for (Node $child: getChildElements($lcontext)) {
42         LocalContext $childcontext = $context.createLocalContext();
43         $childcontext.setNode($child);
44         $childcontext.setParent($lcontext);
45         insert($childcontext);
46     }
47     modify($lcontext) { setRecurse(true); }
48     Fragment $fragment =
49         $context.createFragment(getId($lcontext, $context));
50     $context.getModel().getRoot().addChild($fragment);
51 end
52
53 rule "Reference Insertion"
54 salience 10
55 when
56     $context: DroolsGlobalContext()
57     $lcontext: DroolsLocalContext(
58         $context.knowledgebases["structure"].termGroups["reference"]
59         contains getIndicators($this))
60 then
61     List<DataMapping> $mappings = $context.getMappings("reference");
62     for (DataMapping $mapping: $mappings) {
63         String $reference = $context.evaluateXPath(
64             $mapping.getSource(), $lcontext.getNode());
65         String $filename = getFilename($reference, $context);
66         if (!$filename.equals("")) {
67             Document $doc = $context.loadDocument($filename);
68             LocalContext $rcontext =
69                 $context.createLocalContext();
70             $rcontext.setNode($doc.getDocumentElement());

```

```

71         if ($context.visit($filename)
72             insert($rcontext);
73     }
74 }
75 end
76
77 rule "Data Annotation"
78 when
79     $context: DroolsGlobalContext()
80     $lcontext: DroolsLocalContext(
81         $context.knowledgebases["structure"].termGroups["data"]
82         contains getIndicators($this))
83 then
84     List<DataMapping> $mappings = $context.getMappings("data");
85     for (DataMapping $mapping: $mappings) {
86         String $value = $context.evaluateXPath($mapping.getSource(),
87             $lcontext.getNode());
88         if (!$value.equals(""))
89             annotate($context.getStack().top(),
90                 $mapping.getTarget(), $value, $context);
91     }
92 end
93
94 rule "Reference"
95 when
96     $context: DroolsGlobalContext()
97     $lcontext: DroolsLocalContext(
98         $context.knowledgebases["structure"].termGroups["reference"]
99         contains getIndicators($this))
100 then
101     List<DataMapping> $mappings = $context.getMappings("reference");
102     for (DataMapping $mapping: $mappings) {
103         String $reference = $context.evaluateXPath(
104             $mapping.getSource(), $lcontext.getNode());
105         String $filename = getFilename($reference, $context);
106         String $anchor = getAnchor($reference, $context);
107         Node $node = null;
108         if (!$filename.equals("")) {
109             $node = $context.loadDocument($filename)
110                 .getDocumentElement();
111         }
112         if (!$anchor.equals("")) {
113             for (DataMapping $idref:
114                 $context.getMappings("idref")) {
115                 String $xpath = $idref.getSource()
116                     .replace("?", $anchor);
117                 Object $obj = $context.evaluateXPath($xpath,
118                     $lcontext.getNode());
119                 if ($obj != null) {
120                     $node = $obj;
121                     break;
122                 }
123             }
124         }
125         if ($node != null) {
126             LocalContext $rcontext = $context.createLocalContext();
127             $rcontext.setNode($node);
128             Fragment $target = $context.createFragment(
129                 getId($rcontext, $context));
130             annotate($context.getStack().top(),
131                 $mapping.getTarget(), $target, $context);
132         }
133     }

```

```

133     }
134 end
135
136 rule "Fragment"
137 when
138     $context: DroolsGlobalContext()
139     $lcontext: DroolsLocalContext(
140         $context.knowledgebases["structure"].termGroups["fragment"]
141         contains getIndicators($this))
142 then
143     Fragment $fragment =
144         $context.createFragment(getId($lcontext, $context));
145     String $ind = getIndicators($lcontext);
146     List<DataMapping> $mappings;
147     $mappings = $context.getMappings("styleFragment");
148     for (DataMapping $mapping: $mappings)
149         if ($ind.equals($mapping.getSource()))
150             $fragment.addType($mapping.getTarget());
151     String $text = $lcontext.getNode().getTextContent();
152     $mappings = $context.getMappings("keywordFragment");
153     for (DataMapping $mapping: $mappings)
154         if ($text.contains($mapping.getSource()))
155             $fragment.addType($mapping.getTarget());
156     $lcontext.setFragment($fragment);
157     BackgroundKnowledgeBase $kb = $context.getKnowledgebases()
158         .get("documentStructure");
159     boolean changed = true;
160     while (changed) {
161         changed = false;
162         for (String $newtype: $fragment.getTypes())
163             for (String $stacktype: $context.getStack().top()
164                 .getTypes())
165                 if ($kb.getBroaderSelf($stacktype)
166                     .contains($newtype)) {
167                     $context.getStack().pop();
168                     changed = true;
169                 }
170     }
171     $context.getStack().top().addChild($fragment);
172     $context.getStack().push($fragment);
173 end
174
175 rule "File Root"
176 when
177     $context: DroolsGlobalContext()
178     $lcontext: DroolsLocalContext(parent == null)
179 then
180     Fragment $fragment =
181         $context.createFragment(getId($lcontext, $context));
182     $fragment.setType("File");
183     $context.getStack().clear();
184     $context.getStack().push($fragment);
185 end

```

Remark 9.1.9. Recall from section 5.1 (Application) that along with the correct processing order of the media objects, it is also necessary to reset the stack `context.getStack()` to an earlier state when backtracking to an earlier branching point in the base document model. The correctness of the processing order is ensured by the order in which the media objects are inserted by the rules, and by the processing order of the rule engine.

Resetting the stack has to be done whenever a path from a branching point has been followed to its end, i.e., when the current media object has no successors. A branching point is any media object with more than one (unprocessed) successor. This can be achieved by introducing two functions `branch()` and `join()`. The former pushes the current stack in its entirety on top of a separate “backup stack”. The latter function restores the current stack from the top of this “backup stack”.

In terms of the rules from description 9.1.8, this necessitates two changes. The first change is to the element recursion rule, where the number of successors is determined for each element based on the background knowledge used in the reference insertion rule. Based on the number of successors, local `branch` and `join` properties are set.

```

1 rule "Element Recursion"
2 salience 20
3 when
4     $context: DroolsGlobalContext()
5     $lcontext: DroolsLocalContext(parent == null && recursed == false)
6 then
7     List<Node> $children = getChildElements($lcontext);
8     for (int $i=0;$i<$children.size();$i++) {
9         Node $child = $children.get($i);
10        LocalContext $childcontext = $context.createLocalContext();
11        $childcontext.setNode($child);
12        $childcontext.setParent($lcontext);
13        int $successorCount = 1; // default
14        if ($context.getKnowledgebases().get("structure").getTermGroups()
15            .get("reference").contains(getIndicators($childcontext)))
16            $successorCount++; // additional reference
17        if ($i == $children.size() - 1)
18            $successorCount--; // no default successor (last element)
19        if ($successorCount > 1)
20            $childcontext.setBranch(true);
21        else if ($successorCount == 0)
22            $childcontext.setJoin(true);
23        insert($childcontext);
24    }
25    modify($lcontext) { setRecursed(true); }
26    Fragment $fragment =
27        $context.createFragment(getId($lcontext, $context));
28    $context.getModel().getRoot().addChild($fragment);
29 end

```

The second change is the insertion of two rules to actually deal with the branching and joining. The rules must be inserted between the reference insertion and the data annotation rules to ensure that they are triggered at the correct time.

```

1 rule "Element Branching"
2 when
3     $context: DroolsGlobalContext()
4     $lcontext: DroolsLocalContext(branch == true)
5 then
6     $context.branch();
7 end
8
9 rule "Element Joining"
10 when
11     $context: DroolsGlobalContext()
12     $lcontext: DroolsLocalContext(join == true)
13 then

```

```

14     $context.join();
15 end

```

It is, however, not always necessary to make these control flow adjustments. In particular, these adjustments can be omitted for document with a very regular structure, i.e., for documents where different files always contain the same structural types and where references always point to fragments with identical structural types. In practice, this is the case for most of the documents we have encountered. Since the above changes notably clutter up the rule definitions while only necessary in specific cases, we have omitted them from description 9.1.8.

Example 9.1.10 (Drools Extraction Rules (3)). The generic extraction rules can be applied to documents encoded in files like the following:

```

1 <!-- index.htm -->
2 <html>
3   <body>
4     <h1>Introduction</h1>
5     <a href="chapter2.htm">Link</a>
6   </body>
7 </html>

1 <!-- chapter2.htm -->
2 <html>
3   <body>
4     <h1>Chapter 2</h1>
5     <h2 class="definition">Data Structure</h2>
6     <a href="chapter3.htm">Link</a>
7     <a href="chapter4.htm">Link</a>
8   </body>
9 </html>

1 <!-- chapter3.htm -->
2 <html>
3   <body>
4     <h1>Chapter 3</h1>
5     <h2 class="example">Data Structure</h2>
6     <a href="chapter5.htm">Link</a>
7   </body>
8 </html>

1 <!-- chapter4.htm -->
2 <html>
3   <body>
4     <h1>Chapter 4</h1>
5     <h2 class="definition">Binary Tree</h2>
6     <h2 class="illustration">Binary Tree</h2>
7     <a href="chapter5.htm">Link</a>
8   </body>
9 </html>

```

```

1 <!-- chapter5.htm -->
2 <html>
3   <body>
4     <h1>Conclusion</h1>
5   </body>
6 </html>

```

This document does not specify exactly where references start in its structure: for example, in “chapter4.htm”, the `<a>` reference could be part of the preceding illustration or it could be part of the chapter. The extraction process will assume the former, so different from the previous document model, references start in paragraphs and not in chapters.

The knowledge base “documentStructure” contains information about which structural types indicate a higher level in the document hierarchy (recall that the `skos:broader` role represents a has-broader relation, not an is-broader-than relation):

```

vdk:file      rdf:type      skos:Concept ;
              skos:prefLabel "File" .

vdk:chapter   rdf:type      skos:Concept ;
              skos:prefLabel "Chapter" ;
              skos:broader   vdk:file .

vdk:paragraph rdf:type      skos:Concept ;
              skos:prefLabel "Paragraph" ;
              skos:broader   vdk:chapter .

```

The knowledge base “structure” is adapted to the new file format as follows:

```

vdk:fragment  rdf:type      skos:Concept ;
              skos:prefLabel "h1" ;
              skos:altLabel  "h2_definition" ;
              skos:altLabel  "h2_example" ;
              skos:altLabel  "h2_illustration" .

vdk:data      rdf:type      skos:Concept ;
              skos:prefLabel "h1" ;
              skos:altLabel  "h2_definition" ;
              skos:altLabel  "h2_example" ;
              skos:altLabel  "h2_illustration" .

vdk:reference rdf:type      skos:Concept ;
              skos:prefLabel "a" .

```

The “id” and “idref” mappings remain unchanged from example 9.1.7:

source	target
.	id
//*[@id="?"]	idref

The “data” mappings are adapted as follows:

source	target
. [node-name(.)="h1"]/text()	title
. [node-name(.)="h2"]/text()	term

The “reference” mappings remain the same as in example 9.1.7:

source	target
./@href	reference

The “*styleFragment*” mappings are adapted as follows:

<i>source</i>	<i>target</i>
<i>h1</i>	<i>Chapter</i>
<i>h2_definition</i>	<i>Definition</i>
<i>h2_definition</i>	<i>Paragraph</i>
<i>h2_example</i>	<i>Example</i>
<i>h2_example</i>	<i>Paragraph</i>
<i>h2_illustration</i>	<i>Illustration</i>
<i>h2_illustration</i>	<i>Paragraph</i>

The “*keywordFragment*” mappings remain empty.

Let us apply the extraction rules from description 9.1.8 to this document, using the given background knowledge. We start with an initial fact base containing only the root element of the first file, namely the `<html>` element of “*index.htm*” which we will denote as *1/html*. Note that we will use local contexts transparently by only regarding the file elements wrapped within these contexts whenever possible. The initial fact base can be written as follows:

(*1/html*)

This element immediately triggers the element recursion rule because it matches the rule’s condition and the rule has the highest priority. Recursively inserting all child elements in reverse document order and then re-inserting the root element leads to the following fact base:

(*1/html, 1/body, 1/h1, 1/a*)

A place holder fragment for the current file is also created and added to the root of the otherwise empty document model.

The next rule that is triggered is the reference insertion rule for the *1/a* element, because it has the highest priority of all matching rules. Its node name *a* matches the preferred label of the *reference* concept in the “*structure*” knowledge base.

In the conclusion of this rule, all “*reference*” data mappings – namely the mapping from the XPath `./@href` onto a *reference* property – are resolved. Resolving the XPath for the *1/a* elements yields the file name “*chapter2.htm*”. This file is parsed into a DOM tree, and since it has not been regarded before (`$context.visit()` evaluates to true), its root element (denoted as *2/html*) is inserted into the fact base:

(*2/html, 1/html, 1/body, 1/h1, 1/a*)

The new root element immediately triggers the element recursion rule again, leading to the the creation of another place holder fragment and to the insertion of all other elements from the new file into the fact base:

(*2/html, 2/body, 2/h1, 2/h2, 2/a[1], 2/a[2], 1/html, ...*)

The first of the two references, denoted as *2/a[1]*, triggers the reference insertion rule and the root element of the file “*chapter3.htm*” is inserted into the fact base. This leads to the insertion of the other elements of “*chapter3.htm*” by the element recursion rule and to the creation of yet another place holder fragment.

(*3/html, 3/body, 3/h1, 3/h2, 3/a, 2/html, ...*)

Currently, there are two unprocessed reference elements in the fact base: `3/a` and `2/a[2]`. The former has been inserted last, so it is processed first and `5/html` from “chapter5.htm” is inserted (along with the creation of a place holder fragment), followed by the other elements from this file. Then, `2/a[2]` is processed and the elements of “chapter4.htm” are inserted into the fact base, and a final place holder fragment is created.

```
(4/html, 4/body, 4/h1, 4/h2[1], 2/h2[2], 4/a, 5/html, 5/body, 5/h1, 3/html,
...)
```

Finally, `4/a` with a link to “chapter5.htm” is processed. But since this file has already been processed, no new elements are inserted and the fact base is complete.

```
(4/html, 4/body, 4/h1, 4/h2[1], 2/h2[2], 4/a, 5/html, 5/body, 5/h1, 3/html,
3/body, 3/h1, 3/h2, 3/a, 2/html, 2/body, 2/h1, 2/h2, 2/a[1], 2/a[2], 1/html,
1/body, 1/h1, 1/a)
```

This concludes the first phase of the processing, in which the fact base is populated. The document model is still empty at this point except for its root fragment and the five place holder fragments for the five files, and so is the stack of fragments that is maintained in the global context. The five place holder fragments are arranged in the order in which they are reached through the references, namely “index.htm” first, then “chapter2.htm”, “chapter3.htm”, “chapter5.htm” and “chapter4.htm”. The semantic document model is now created in the second phase.

The first element in the fact base, `4/html` matches against the file root rule, thereby creating the first fragment of type “File” (denoted as `4/File`). It replaces its place holder in the document model and is pushed onto the (currently empty) stack:

4/File

The next element, `4/body`, does not match any rule, but `4/h1` does: it matches the fragment rule by virtue of the preferred label “h1” of the **fragment** concept in the “structure” ontology. So a new fragment, denoted as `4/Chapter`, is created. From the “styleFragment” mappings, we learn that this fragment has the type “Chapter” because of its source style “h1”. After adding the type to the fragment, we compare this type (“Chapter”) with the type of the top stack element (“File”). When we consult the “documentStructure” knowledge base, we find that “Chapter” is not a broader structural type than “File” (in fact, we find that the exact opposite is true). We therefore leave `4/File` on the stack, add `4/Chapter` as a child fragment of `4/File` in the document model, and push `4/Chapter` on top of the stack:

4/File	4/Chapter
--------	-----------

But we are not done with `4/h1`: it also matches the data annotation rule by virtue of a label of the **data** concept in the “structure” knowledge base. Now, all XPath expressions from the “data” data mappings are evaluated against the current element. Only the first expression, `.[node-name(.)="h1"]/text()` yields a result: “Chapter 4”. This is added to the current top stack element, `4/Chapter`, as an annotation. The predicate `title` is specified in the data mapping. Here it becomes clear why the order of the rules matters: if the data annotation rule had been triggered before the fragment rule, the data would have been annotated to the wrong fragment! Since JBoss Drools tries to match the rules in reverse order, last to first, the fragment rule must be written after the data annotation rule.

Next, the $4/h2[1]$ element matches first against the fragment rule, and then against the data annotation rule. This works as before, resulting in a new fragment $4/Definition$ of type “Paragraph” and “Definition” on top of the stack, with a **term** annotation “Binary Tree”.

$4/File$	$4/Chapter$	$4/Definition$	
----------	-------------	----------------	--

The next element, $4/h2[2]$, that also matches both rules is more interesting. In the fragment rule, a new fragment $4/Illustration$ of type “Paragraph” and “Illustration” is created. But now we find that a fragment of the same type “Paragraph” sits atop the stack. So before we do anything else, we remove it. $4/Illustration$ is then added as a child of $4/Chapter$ and pushed onto the stack.

$4/File$	$4/Chapter$	$4/Illustration$	
----------	-------------	------------------	--

Now, $4/a$ is processed a second time, this time by the reference rule. Its target is evaluated to “chapter5.htm” and the appropriate file is parsed into a DOM tree. Since no anchor tag, i.e., a pointer to a specific element in the file like `chapter5.htm#example`, is provided, the reference points to the root of the file. A new fragment $5/File$ is created for this location, and a **reference** relationship with this fragment is added to $4/Illustration$. Note that the targeted fragment may not yet be in the document model’s hierarchical structure, i.e., be defined as a sub-fragment of another fragment. Instead, it may be floating freely, only tied to the model by the reference from chapter 4.

Incidentally, in this case $5/File$ already exists in its proper place in the document model, namely as the place holder fragment for “chapter5.htm”. $5/html$ is processed next, triggering the file root rule. Now $5/File$ is given its proper type, the stack is emptied, and the file fragment is pushed onto it.

$5/File$	
----------	--

The other elements from “chapter5.htm” are processed in a similar manner, as are the elements from the other files afterwards.

Figure 9.2 shows part of the flat source files and the corresponding part of the semantic document model, whose implementation is extracted from the files.

Example 9.1.11 (Drools Extraction Rules (4)). The generic rules from description 9.1.8 can also be applied to the hierarchical file format from example 9.1.2.

The first phase, populating the fact base, yields the following result:

```
(4/html, 4/body, 4/div, 4/div/div[1], 4/div/div[2], 4/div/div[2]/span,
4/div/div[3], 4/div/div[3]/span, 4/div/a, 5/html, 5/body, 5/div,
5/div/div, 3/html, 3/body, 3/div, 3/div/div[1], 3/div/div[2],
3/div/div[2]/span, 3/div/a, 2/html, 2/body, 2/div, 2/div/div[1],
2/div/div[2], 2/div/div[2]/span, 2/div/a[1], 2/div/a[2], 1/html, 1/body,
1/div, 1/div/div, 1/div/a)
```

Using the background knowledge from example 9.1.7 and the “documentStructure” ontology from example 9.1.10, this leads to the same document model as in example 9.1.7.

For example, when processing the seventh element from the fact base, $4/div/div[3]$, the stack contains the following three fragments:

$4/File$	$4/Chapter$	$4/Definition$	
----------	-------------	----------------	--

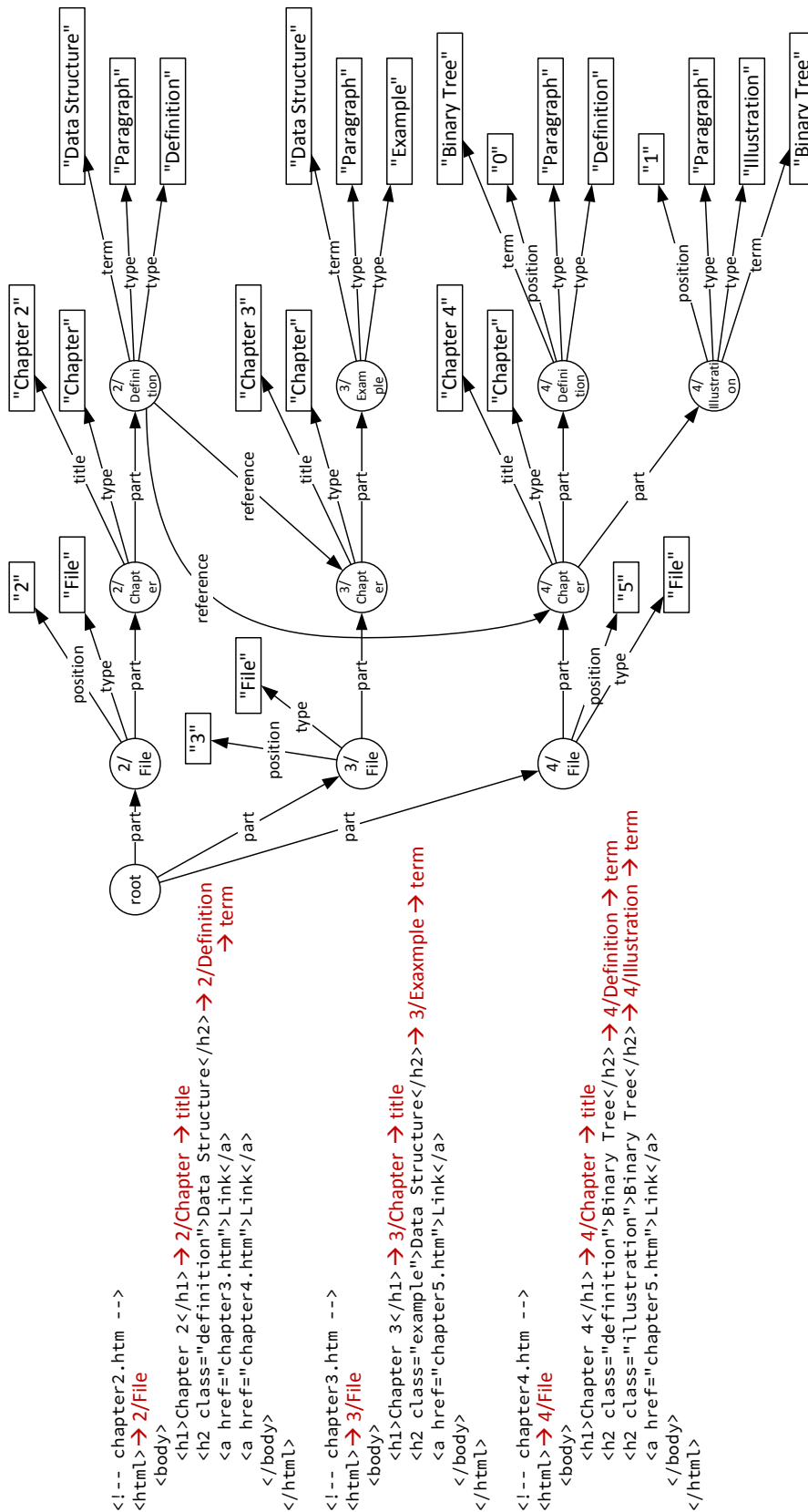


Figure 9.2: Flat source document and semantic document model

The element triggers the fragment rule, and a new fragment 4/illustration is created. Because 4/div/div[3] is a <div> element with a class attribute value of “illustration”, it matches the background knowledge entry for the fragment concept in “structure”. It also matches the div_illustration source in “styleFragment”, so the fragment is typed as both “Paragraph” and “Illustration”. Now, a “Paragraph” fragment is already on top of the stack, so that fragment is removed from the stack and replaced by 4/illustration, which is also added as a child to 4/Chapter.

The generic rules from description 9.1.8 can easily be applied to other documents by adapting the three auxiliary functions and the background knowledge. Changes and additions to the rules are necessary for special cases that cannot be formalised in the background knowledge.

The parsing specifics need to be changed for applying the rules to non-XML document formats. However, it is often easier and more efficient to preprocess non-XML formats to an XML format, because then a standard DOM parser can be used. We have successfully used this technique for both Microsoft Word documents and for L^AT_EX documents.

As stated before, include commands should be resolved in preprocessing, but they can also be resolved in the rules during the recursive insertion of elements into the fact base.

Example 9.1.12 (Drools Extraction Rules (5)). *As another example, let us regard a document similar to the one above, but specified in L^AT_EX format:*

```

1 \begin{document}
2
3   \chapter{Introduction}\label{chp:chapter_1}
4
5   Link~\ref{chp:chapter_2}
6
7
8   \chapter{Chapter 2}\label{chp:chapter_2}
9
10  \begin{definition}{Data Structure}
11  \end{definition}
12
13  Link~\ref{chp:chapter_3}
14  Link~\ref{chp:chapter_4}
15
16
17  \chapter{Chapter 3}\label{chp:chapter_3}
18
19  \begin{example}{Data Structure}
20  \end{example}
21
22  Link~\ref{chp:chapter_5}
23
24
25  \chapter{Chapter 4}\label{chp:chapter_4}
26
27  \begin{definition}{Binary Tree}
28  \end{definition}
29
30  \begin{illustration}{Binary Tree}
31  \end{illustration}
32
33  Link~\ref{chp:chapter_5}
34
35
36  \chapter{Conclusion}\label{chp:chapter_5}
37
38 \end{document}

```

The simplest way to deal with such non-XML formats is to convert them into XML in a preprocessing step as stated above. This not only allows us to use the vast array of tools and libraries available for XML, but it also prevents us from having to define new context structures (based on the `GlobalContext` and `LocalContext` interfaces) for every file format.

Using conversion software that we developed during the VERDIKT project, the \LaTeX source file can be preprocessed into a standardised XML format. The \LaTeX parser and converter is available through the `LaTeXDocumentAdapter` class.

Every \LaTeX environment is converted into an `<environment>` element, with `<param>` sub-elements for each parameter of the environment. For example, a `\begin{definition}{Data Structure}` \LaTeX environment is converted into a `<environment name="definition"><param>Data Structure</param></environment>` XML element. Similarly, \LaTeX commands are converted into `<command>` elements. Command parameters are handled in the same manner as parameters of environments.

The \LaTeX parser is not a full \TeX processor, but it is restricted in several ways. For one, it cannot expand macros, instead converting every command directly into XML. It also does not recognise every syntactical variant allowed in \TeX , such as command parameters that are not enclosed in some kind of parentheses but that are separated from the command name by whitespace, for instance. While this prevents many \LaTeX documents from being preprocessed entirely without flaw and is thus unsuitable for commercial application, it is sufficient for evaluation purposes.

The following file content is obtained when applying the preprocessor to the \LaTeX source above:

```

1 <environment name="document">
2
3   <command name="chapter">
4     <param>Introduction</param>
5   </command>
6   <command name="label">
7     <param>chp:chapter_1</param>
8   </command>
9
10  Link
11  <command name="ref">
12    <param>chp:chapter_2</param>
13  </command>
14
15
16  <command name="chapter">
17    <param>Chapter 2</param>
18  </command>
19  <command name="label">
20    <param>chp:chapter_2</param>
21  </command>
22
23  <environment name="definition">
24    <param>Data Structure</param>
25  </environment>
26
27  Link
28  <command name="ref">
29    <param>chp:chapter_3</param>
30  </command>
31  Link
32  <command name="ref">
33    <param>chp:chapter_4</param>
34  </command>
35

```

```

36
37 <command name="chapter">
38   <param>Chapter 3</param>
39 </command>
40 <command name="label">
41   <param>chp:chapter_3</param>
42 </command>
43
44 <environment name="example">
45   <param>Data Structure</param>
46 </environment>
47
48 Link
49 <command name="ref">
50   <param>chp:chapter_5</param>
51 </command>
52
53
54 <command name="chapter">
55   <param>Chapter 4</param>
56 </command>
57 <command name="label">
58   <param>chp:chapter_4</param>
59 </command>
60
61 <environment name="definition">
62   <param>Binary Tree</param>
63 </environment>
64
65 <environment name="illustration">
66   <param>Binary Tree</param>
67 </environment>
68
69 Link
70 <command name="ref">
71   <param>chp:chapter_5</param>
72 </command>
73
74
75 <command name="chapter">
76   <param>Conclusion</param>
77 </command>
78 <command name="label">
79   <param>chp:chapter_5</param>
80 </command>
81
82 </environment>

```

The knowledge base “documentStructure” can be used as in example 9.1.11 without adaptation.

The knowledge base “structure” is adapted to the new file format as follows:

```

vdk:fragment  rdf:type      skos:Concept ;
               skos:prefLabel "command.chapter" ;
               skos:altLabel  "environment.definition" ;
               skos:altLabel  "environment.example" ;
               skos:altLabel  "environment.illustration" .
vdk:data      rdf:type      skos:Concept ;
               skos:prefLabel "command.chapter" ;
               skos:altLabel  "environment.definition" ;
               skos:altLabel  "environment.example" ;
               skos:altLabel  "environment.illustration" .
vdk:reference rdf:type      skos:Concept ;
               skos:prefLabel "command.ref" .

```

The “id” mappings are adapted as follows to match the first `\label{}` command after the current node:

source	target
<code>./following::command[@name="label"][1]/param/text()</code>	<code>id</code>

The “idref” mappings are adapted in a similar manner:

source	target
<code>//*[@./following::command[@name="label"][1]/param/text()="?"]</code>	<code>idref</code>

The “data” mappings are adapted as follows:

source	target
<code>.[@name="chapter"]/param/text()</code>	<code>title</code>
<code>.[@name!="chapter"]/param/text()</code>	<code>term</code>

The “reference” mappings remain the same as in example 9.1.7:

source	target
<code>./param/text()</code>	<code>reference</code>

The “styleFragment” mappings are adapted as follows:

source	target
<code>command.chapter</code>	<code>Chapter</code>
<code>environment.definition</code>	<code>Definition</code>
<code>environment.definition</code>	<code>Paragraph</code>
<code>environment.example</code>	<code>Example</code>
<code>environment.example</code>	<code>Paragraph</code>
<code>environment.illustration</code>	<code>Illustration</code>
<code>environment.illustration</code>	<code>Paragraph</code>

The “keywordFragment” mappings remain empty.

Of the auxiliary functions, only the `getIndicators()` function needs to be adapted to take the names of commands and environments into account:

```

1 function String getIndicators(LocalContext lcontext) {
2     String result = lcontext.getNode().getLocalName();
3     if (lcontext.hasAttribute("name"))
4         result += "_" + lcontext.getAttributes().get("name");
5     return result;
6 }

```

Applying the extraction rules with the given background knowledge to the XML version of the source file, we start with a fact base that contains only the root element `<environment name="document">` of this file, which we will denote as `env/document`:

`(env/document)`

This triggers the element recursion rule, leading to the insertion of all DOM elements into the fact base:

`(env/document, cmd/chapter[1], param[1], cmd/label[1], cmd/ref[1], param[2],
cmd/chapter[2], ...)`

It also leads to the creation of a single placeholder fragment in the document model. Next, the reference insertion rule is triggered by the five reference commands `cmd/ref[1]` through `cmd/ref[5]`. This rule, however, does nothing as the references do not point to new files. The fact base is now complete.

The first processing rule to be triggered after the fact base is complete is the file root rule. A file fragment is created in the document model and replaces its placeholder there. It is also pushed onto the stack:

File	
------	--

Next, `cmd/chapter[1]` triggers the fragment rule. A `Chapter[1]` fragment is created, added as a child of the File fragment, and pushed onto the stack.

File	Chapter[1]	
------	------------	--

Then, `param[1]` triggers the data annotation rule, and `cmd/ref[1]` triggers the reference rule. Using the background knowledge, the reference `"chp:chapter2"` is resolved to a fragment `Chapter[2]` based on `cmd/chapter[2]`. When `cmd/chapter[2]` is processed next, the resulting fragment `Chapter[2]` replaces `Chapter[1]` on top of the stack.

File	Chapter[2]	
------	------------	--

Processing the remaining elements finally results in a document model similar to the one in example 9.1.10.

Additional details on different extraction specifications can be found in chapter 10 and in appendix B. They are also based on the generic rules introduced here, showing how transferable they really are.

Conclusion

In this section, we have discussed the use of Java programs, XQuery programs, and Drools rule specifications for defining the extraction logic for semantic document models from digital documents. We have shown that Java programs can quickly become too complex to be maintainable. This is also true for XQuery programs. In contrast, rules can be applied efficiently and are easily transferable across formats and domains.

While it is possible to write well-structured Java code that uses external background knowledge, thus eliminating the main drawback of this approach, it is easier to do so in a rule-based language because the rule structure enforces (or at least suggests) a methodical structuring of the extraction logic. It also makes it harder to use "implementation shortcuts" in the programming.

We therefore recommend using a rule language with external background knowledge for specifying the extraction logic. This approach moves everything that is not generic, i.e., that depends on either the document format or on the application domain, out of the extraction logic and into the background knowledge. This knowledge is then used as a parameter for the generic extraction logic. Generic rules can be applied to (almost) arbitrary documents, because they are defined on base document models instead of any specific file format, and (almost) all knowledge that is specific to a format and a domain is externalised in the background knowledge.

9.2 Inference on Document Models

There are two primary relevant inference tasks on semantic document models:

1. Deduce new types for fragments. For example, if a fragment is known to be a **Definition**, it can be deduced that it is also a **Paragraph**.
2. Deduce new topics and other role relationships for fragments
 - ▶ based on terminological relationships (generalisation or relatedness of topics). For example, if a topic t_s is relevant for a fragment f , then a topic t_g that is more general than t_s is also relevant for f .
 - ▶ based on fragment relationships (part-of). For example, if a topic t is relevant for a fragment f_{sub} , then t is also relevant for a fragment f_{super} that is a super-fragment of f_{sub} .

For many inference tasks on document models, description logics are an adequate formalism. The inference services provided by most description logic languages can be applied directly to a description logics-based implementation of a semantic document model. These inference services have one important omission, however: generalisation relationships can only be defined on (complex) concepts, not on individuals. Yet there are excellent reasons to specify generalisation relationships on individuals instead of concepts: individuals can be classified (via concept assertions), and they can have arbitrary relationships with other individuals (via role assertions).

Remark 9.2.1. *Note that using set constructors and concept subsumption does not provide a loophole for specifying generalisation relationships on individuals: Let $\mathcal{O} = (\emptyset, \emptyset, \{i_{sub}, i_{super}\}, \{\{i_{sub}\} \sqsubseteq \{i_{super}\}\})$ be an ontology. Then for every model $I_{\mathcal{O}}$ of \mathcal{O} : $I_{\mathcal{O}} \models (i_{sub} = i_{super})$, which clearly is not the intended semantics.*

One option for specifying a form of individual generalisation is to combine individuals with a specific role and to generalise over the existence of role instances.

Definition 9.2.2 (Individual Generalisation Restricted to a Specific Role). *For individuals i_{sub} , i_{super} and a role R , $\exists R.\{i_{sub}\} \sqsubseteq \exists R.\{i_{super}\}$ is a generalisation on individuals restricted to R .*

Example 9.2.3 (Individual Generalisation Restricted to a Specific Role). *Let $\mathcal{O} = (C, R, I, X)$ be an ontology with*

$$\begin{aligned}
 C &= \{\mathit{Fragment}, \mathit{Topic}\}, \\
 R &= \{\mathit{hasTopic}\}, \\
 I &= \{f, \mathit{Data Structure}, \mathit{Binary Tree}\}, \text{ and} \\
 X &= \{\exists \mathit{hasTopic}.\{\mathit{Binary Tree}\} \sqsubseteq \exists \mathit{hasTopic}.\{\mathit{Data Structure}\}, \\
 &\quad \mathit{Fragment}(f), \mathit{Topic}(\mathit{Data Structure}), \mathit{Topic}(\mathit{Binary Tree}), \\
 &\quad \mathit{hasTopic}(f, \mathit{Binary Tree})\}.
 \end{aligned}$$

Then for every model $I_{\mathcal{O}}$ of \mathcal{O} : $I_{\mathcal{O}} \models \mathit{hasTopic}(f, \text{Data Structure})$.

This can be implemented in description logics or in OWL and a reasoner such as Pellet can be used to deduce the additional assertion.

This approach is very specific, which may necessitate a large number of axioms. In the worst case, for every distinct triple of two individuals and one role one axiom is required. On the other hand, this allows for this approach to be used surgically: in some cases a generalisation is only sensible in the context of a small number of roles.

Another less specific option is to use inference rules to generate the desired assertions. This inference is based on role assertions that serve as a declaration of an intended generalisation. In particular, the `hasNarrower` role and its relatives are well suited for this task. It should be noted that for such a generalisation of individuals, an EER-like semantics replaces the usual set-semantics for description logic generalisation, so that a more specialised individual can be substituted in any place where a more general individual is used.

Description 9.2.4 (Inference Rules). *We define two rules that interpret `hasNarrower` role assertions as a generalisation on individuals:*

1. for all t_1, t_2 with $\mathit{hasNarrower}^+(t_1, t_2)$ do:

- ▶ for all assertions $C(t_1)$ do: assert $C(t_2)$,
- ▶ for all assertions $R(t_1, x)$ do: assert $R(t_2, x)$, and
- ▶ for all assertions $R(x, t_1)$ do: assert $R(x, t_2)$,

where x is an individual, C is a concept, and R is a role with $R \notin \{\mathit{sameAs}, \mathit{equals}, \mathit{hasNarrower}\}$ (*sameAs* is the OWL role that expresses individual equality).

2. for all t_1, t_2 with $\mathit{hasNarrower}(t_1, t_2) \wedge \mathit{hasNarrower}(t_2, t_1)$ do:

- ▶ assert $\mathit{equals}(t_1, t_2)$ or $\mathit{sameAs}(t_1, t_2)$ (the latter carries stronger build-in semantics).

The restriction of R in point 1 is necessary to avoid undesired side effects by successively adding new generalisations and equivalences. For example, two existing assertions $\mathit{hasNarrower}(\text{Data Structure}, \text{Binary Tree})$ and $\mathit{hasNarrower}(\text{Data Structure}, \text{List})$ should not lead to a new assertion $\mathit{hasNarrower}(\text{Binary Tree}, \text{List})$.

Example 9.2.5 (Inference Rules). *Let $\mathcal{O} = (C, R, I, X)$ be an ontology with*

$$\begin{aligned}
C &= \{ \mathit{Fragment}, \mathit{Topic} \}, \\
R &= \{ \mathit{hasTopic}, \mathit{hasNarrower} \}, \\
I &= \{ f_1, f_2, f_3, f_4, \text{Computer Science, Data Base, Data Structure, List,} \\
&\quad \text{Tree, Binary Tree, BTree} \}, \text{ and} \\
X &= \{ \mathit{Fragment}(f_1), \mathit{Fragment}(f_2), \mathit{Fragment}(f_3), \mathit{Fragment}(f_4), \\
&\quad \mathit{Topic}(\text{Computer Science}), \\
&\quad \mathit{Topic}(\text{Data Base}), \mathit{Topic}(\text{Data Structure}), \\
&\quad \mathit{Topic}(\text{List}), \mathit{Topic}(\text{Tree}), \\
&\quad \mathit{Topic}(\text{Binary Tree}), \mathit{Topic}(\text{BTree}), \\
&\quad \mathit{hasNarrower}(\text{Computer Science, Data Base}), \\
&\quad \mathit{hasNarrower}(\text{Computer Science, Data Structure}), \\
&\quad \mathit{hasNarrower}(\text{Data Base, BTree}), \\
&\quad \mathit{hasNarrower}(\text{Data Structure, List}), \\
&\quad \mathit{hasNarrower}(\text{Data Structure, Tree}), \\
&\quad \mathit{hasNarrower}(\text{Tree, Binary Tree}), \\
&\quad \mathit{hasNarrower}(\text{Tree, BTree}), \\
&\quad \mathit{hasTopic}(f_1, \text{Data Structure}), \\
&\quad \mathit{hasTopic}(f_2, \text{Computer Science}), \\
&\quad \mathit{hasTopic}(f_3, \text{Tree}), \\
&\quad \mathit{hasTopic}(f_4, \text{Data Base}) \}.
\end{aligned}$$

Then the inference rules from description 9.2.4 generate the following additional assertions

X_{inf} :

$$\begin{aligned}
X_{inf} &= \{ \mathit{hasTopic}(f_1, \text{List}), \\
&\quad \mathit{hasTopic}(f_1, \text{Tree}), \\
&\quad \mathit{hasTopic}(f_1, \text{Binary Tree}), \\
&\quad \mathit{hasTopic}(f_1, \text{BTree}), \\
&\quad \mathit{hasTopic}(f_2, \text{Data Structure}), \\
&\quad \mathit{hasTopic}(f_2, \text{Data Base}), \\
&\quad \mathit{hasTopic}(f_2, \text{List}), \\
&\quad \mathit{hasTopic}(f_2, \text{Tree}), \\
&\quad \mathit{hasTopic}(f_2, \text{Binary Tree}), \\
&\quad \mathit{hasTopic}(f_2, \text{BTree}), \\
&\quad \mathit{hasTopic}(f_3, \text{Binary Tree}), \\
&\quad \mathit{hasTopic}(f_3, \text{BTree}), \\
&\quad \mathit{hasTopic}(f_4, \text{BTree}) \}.
\end{aligned}$$

Example 9.2.6 (Inference Rules (2)). *With very little adaptation of the fact base, the inference rules from description 9.2.4 can also be applied to the has-part relationships between fragments.*

Let $\mathcal{O} = (C, R, I, X)$ be the ontology from example 9.2.5. Let $\mathcal{O}' = (C, R', I, X')$ be an extended ontology with

$$\begin{aligned}
R' &= R \cup \{ \mathit{hasPart} \}, \text{ and} \\
X' &= X \cup \{ \mathit{hasPart} \sqsubseteq \mathit{hasNarrower}, \mathit{hasPart}(f_3, f_4) \}.
\end{aligned}$$

Note, in particular, how the $\mathit{hasPart}$ role has been defined as a specialisation of the $\mathit{hasNarrower}$ role.

Then the inference rules from description 9.2.4 generate the following additional assertions

$$X'_{inf} = X_{inf} \cup \{ \mathit{hasTopic}(f_4, \text{Tree}), \mathit{hasTopic}(f_4, \text{Binary Tree}) \}.$$

The inference rules stated above are not a full substitute for concept subsumption on individuals, however, because they do not allow for complex modelling. For example, generalisations

like `Person` \sqcup `Organisation` \sqsubseteq `Creator` cannot be expressed with the `hasNarrower` role alone. While it might be possible to introduce new roles and new inference rules to capture some of the necessary semantic complexity, we will leave this for future work.

It is often desirable to restrict inference tasks to some of the ontologies relevant for a semantic document model. For example, the conclusions drawn from `hasNarrower` role assertions differ across domains because the semantics of this role are wide and can be interpreted differently in different settings. While it may be sensible to use the `hasNarrower` role for inference on the terminological ontology of a document model as shown in the examples above, it is usually less sensible to use it on the structural ontology. In the terminological ontology, the `hasNarrower` role is often used to model *is-a* relations, while in the structural ontology it is used to model *has-part* relations. The former use is compatible with a generalisation semantics, while the latter is not.

Inference on Large Ontologies

Executing inference tasks on large and complex document models or on very large and complex background knowledge requires a lot of processing power. While the specific complexity varies with the complexity of the description logic language used [BCM⁺03], description logic reasoning is generally exponential in the size of the knowledge base. Attempts to push some of the reasoning itself into a database [Har05, Dok06, ECTOO09] and using efficient query techniques cannot address this fundamental problem. We posit that reducing instead the number of entities to reason on, even at the cost of limiting the expressive power of inference services, can be a viable compromise.

It is possible to make a horizontal cut through a knowledge base, keeping the TBox and all assertions from the ABox in a (main memory based) ontology and moving the rest of the ABox into a database. Since the actual reasoning semantics are kept in the TBox, the database simply acts as a data supplier and the whole open world/closed world issue does not apply. There is, however, little advantage in only keeping the list of individuals (in OWL: a list of URIs) in a database and using database identifiers in the TBox, because it does not limit the overall number of entities. The situation improves if some of the assertions from the ABox can be moved into the database, and thus be excluded from any reasoning. Good candidates are role assertions with literal objects and assertions on concepts and roles that are not part of any other complex axioms. Examples include roles like `rdfs:label`, `skos:prefLabel`, `skos:altLabel`, and `vdk:id`, which are used frequently but are only a hindrance to reasoners.

Another possibility depends on precise knowledge of the ontology schema and the application domain. If it is possible to determine all the entities required for an inference task, then these entities can be taken out of the complete database (vertical cut) and the reasoning can be executed on this smaller subset. Any newly inferred assertions can then be inserted into the original database. The decreased cost of the reasoning can offset the cost of extracting the required data. For example, calculating the transitive closure of the `hasNarrower` role on topics only requires all individuals of type `Topic` and all `hasNarrower` assertions. With proper database indices, retrieving them and running the reasoning task externally can be done with considerable efficiency.

We will revisit this topic briefly in section 11.5.

9.3 Views on a Document Model

As indicated earlier, obtaining a semantic document model of a document is rarely its own reward. Instead, it is usually a stepping stone toward some other goal, like verifying content consistency criteria on a document, like making structural and content-related comparisons between documents, or like creating an inverted structural document index that not only contains the position of terms, but also their affiliation in the document hierarchy. For each of these applications, the data of a semantic document model needs to be transformed into a new form. Borrowing a term from the domain of data bases, this form can be understood as a (usually materialised) *view*.

Yet not only different types of views for different applications are required, because in some instances multiple different views are needed for one and the same application. For document verification, for example, the hierarchical graph of the semantic document model is mapped onto the flat graph of a temporal model. This mapping is not unique, however, but can lead to a different result for each structural level of the document model. This in turn can lead to different verification results as shown in examples 9.3.1 and 9.3.2. So not only is it necessary to find the correct formula for a verification criterion, but for each formula the appropriate view has to be determined.

Example 9.3.1 (Verification Model from Semantic Document Model (1): Chapter Level). *Recall the semantic document model D from example 4.2.15 on page 83. Then the \mathcal{ALCCTL} temporal model from example 3.5.17 (page 55) is a view on D on the level of chapters, i.e., each chapter-fragment from D is represented by a state in the temporal model. Note that the view does not contain information about an illustration, only about examples. Using the axiom $\text{Illustration} \sqsubseteq \text{Example}$, it can be inferred that the illustration of binary trees in chapter 4 is also an example. See below for more information on how to specify which data is represented in the view, and which data is not. A version of this \mathcal{ALCCTL} model that is extended by the specialisation relationship between binary trees and data structures is shown in figure 9.3.*

Now imagine a criterion that specifies that after every definition, there must be an example with the same topic in the next state, no matter which path the reader follows (i.e., in all successor states). Formally: $\text{Definition} \sqsubseteq \text{AXExample}$. Note that this criterion is stricter than the similar criterion in example 3.5.15. In this view, the criterion is not satisfied. Even if binary trees are recognised as special data structures, so that the example in s_4 can be used to satisfy the definition in s_2 , there is no example for the definition in s_4 ! The appropriate example is in the same state and is therefore not recognised, and there is no example in s_5 .

Example 9.3.2 (Verification Model from Semantic Document Model (1): Paragraph Level). *Again, recall the semantic document model D from example 4.2.15.*

Let $S = \{s_1, s_2, s_3, s_{41}, s_{42}, s_5\}$ be a set of states.

Let $R = \{(s_1, s_2), (s_2, s_3), (s_2, s_{41}), (s_{41}, s_{42}), (s_{42}, s_5), (s_3, s_5), (s_5, s_5)\}$ be a successor relation on S that represents the links between them.

Let $\Delta^I = \{\text{Data Structure}, \text{Binary Tree}\}$.

Let $I = \{(s_1, (\Delta^I, ()^{I(s_1)})), (s_2, (\Delta^I, ()^{I(s_2)})), (s_3, (\Delta^I, ()^{I(s_3)})), (s_{41}, (\Delta^I, ()^{I(s_{41})})), (s_{42}, (\Delta^I, ()^{I(s_{42})})), (s_5, (\Delta^I, ()^{I(s_5)}))\}$ be a function that can be used to represent the topics of definitions and examples in each state.

Let $\text{Definition}^{I(s_2)} = \{\text{Data Structure}\}$, $\text{Example}^{I(s_3)} = \{\text{Data Structure}\}$, $\text{Definition}^{I(s_{41})} = \{\text{Binary Tree}, \text{Data Structure}\}$, and $\text{Example}^{I(s_{42})} = \{\text{Binary Tree}, \text{Data Structure}\}$.

Then the \mathcal{ALCCTL} temporal model $M = (S, R, I)$ is a view on D on the level of paragraphs, i.e., each paragraph-fragment from D is represented by a state in the temporal model. Note that

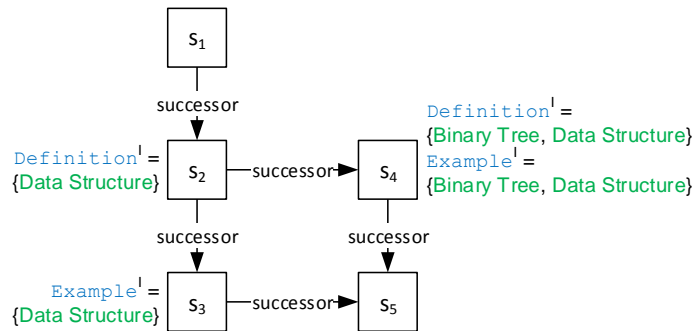


Figure 9.3: Chapter-level view on a semantic document model

whenever a chapter does not have a paragraph sub-fragment, the chapter itself is represented by a state in the temporal model. This model is shown in figure 9.4.

If we apply the criterion from example 9.3.1 ($\text{Definition} \sqsubseteq \text{AXExample}$) to this view, it will be satisfied provided that binary trees are recognised as special data structures. The finer granularity of the structure is the crucial difference here.

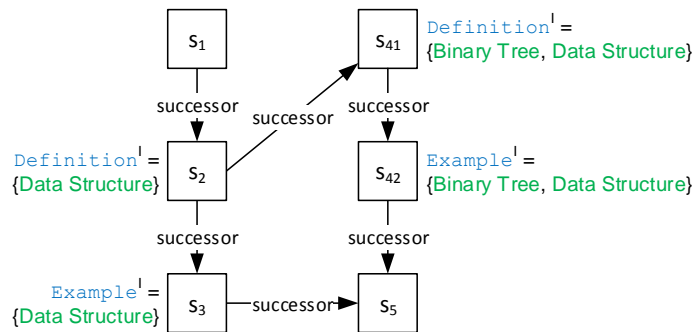


Figure 9.4: Paragraph-level view on a semantic document model

The required structural level depends on the criterion, however. It is not sufficient to simply choose the most detailed structure, i.e., the lowest level in the hierarchy. This is illustrated in example 9.3.3.

Example 9.3.3 (Verification Model from Semantic Document Model (2)). *Let D be a semantic document model similar to the familiar model from example 4.2.15. In chapters 2 and 4 there are learning objectives defined for data structures and for binary trees, respectively. Now imagine a criterion that requires that all learning objectives for each chapter must be covered by an appropriate definition. Formally: $\text{LearningObjective} \sqsubseteq \text{Definition}$.*

A verification model with a fine granularity will not satisfy the criterion, because chapter 4 is split into two paragraphs and the second paragraph does not contain a definition. A coarser granularity is necessary here to retain the chapter cohesion.

The easiest way to specify views on semantic document models is by defining a number of queries on the model to collect the required data points, and a function to combine these sets of data. Since XQuery evaluation is NExpTime-hard in general [Koc06], while SPARQL evaluation is “only” PSpace-complete, we recommend using SPARQL for specifying the queries. In fact, for queries that do not use the UNION and FILTER constructs, SPARQL evaluation is linear. Evaluation of queries that do not use the OPTIONAL construct is NP-complete. [PAG09]

However, SPARQL misses one crucial capability: the ability to formulate recursive queries. This is necessary for determining for example the topics of all sub-fragments of a particular fragment. In some cases, transitive roles can be used to compensate for this limitation. Whenever the distance from the start of the recursion is relevant, however, transitive roles cannot be used. The distance is relevant when limiting the retrieval to a certain maximum distance, or when calculating some kind of cost function. We will discuss some options for SPARQL recursion below.

For state-transition systems, at least three queries are necessary to gather all required information: one to determine the states, one to determine the successor relationships, and one to determine the starting state. For models in temporal (propositional) logic, an additional query is required for each proposition, to determine its interpretations. For models in temporal description logic, additional queries are required for each concept and for each role to determine their interpretations.

Example 9.3.4 (View Definition: \mathcal{ALCCTL} Model). *Let D again be the semantic document model from example 4.2.15. Consider the following set of SPARQL queries:*

1. *states*

```

1 SELECT ?d
2 WHERE {
3     ?s rdf:type vdk:Chapter .
4     ?s vdk:id ?d
5 }
```

2. *starting state*

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$state" .
4     ?s vdk:id ?d .
5     OPTIONAL { ?p ref:successor ?s }
6     FILTER (!bound($p))
7 }
```

3. *successor relation*

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$state" .
4     ?s ref:partTransitive ?p .
5     {
6         s ref:reference ?r
7     } UNION {
8         ?p ref:reference ?r
9     } .
10 }
```

```

11     ?r ref:partTransitive ?t
12 } UNION {
13     ?t ref:partTransitive ?r
14 } .
15 ?t rdf:type vdk:Chapter .
16 ?t vdk:id ?d
17 }

```

4. concepts

(a) Topic

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$state" .
4     ?s ref:partTransitive ?p .
5     {
6         ?s dc:subject ?d
7     } UNION {
8         ?p dc:subject ?d
9     }
10 }

```

(b) Fragment

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$state" .
4     ?s ref:partTransitive ?p .
5     {
6         ?s vdk:id ?d
7     } UNION {
8         ?p vdk:id ?d
9     }
10 }

```

5. roles

(a) hasTopic

i. base

```

1 SELECT ?d
2 WHERE {
3     {
4         ?s vdk:id "$state" .
5         ?s vdk:id ?d
6     } UNION {
7         ?s vdk:id "$state" .
8         ?s ref:partTransitive ?p .
9         ?p vdk:id ?d
10    }
11 }

```


ii. left hand side

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$base" .
4     ?s vdk:id ?d
5 }

```

iii. right hand side

```

1 SELECT ?d
2 WHERE {
3     ?s vdk:id "$base" .
4     ?s dc:subject ?d
5 }

```

Item 1 lists the query that determines the states of the temporal model. It simply matches any fragment of type “chapter” and returns its identifier.

The query in item 2 determines the starting state from this set. The “\$state” expression is iteratively replaced with the ids obtained in the first query, thus matching all fragments that represent states in the temporal model. The query returns only the id of fragments with no predecessor.

Item 3 shows the successor query. It iterates over all states (again by successively replacing the “\$state” expression with each id) and collects all outgoing references from the state fragment and all its sub-fragments (first UNION clause). It then finds the sub- or super-fragment of the referenced fragment (second UNION clause) that has the correct type for a state and returns its id.

Item 4 contains the listings for two concepts: the **Topic** concept query (item 4a) finds all sub-fragments of a state fragment and returns their topics, including the topics of the state fragment itself. The **Fragment** concept query (item 4b) works in a similar manner, only returning the identifiers of the fragments instead of their topics.

A role specification is shown in item 5a. It consists of three queries: the first query determines the base (item 5(a)i). The other two queries are evaluated relative to this base. Here, the base query simply finds all sub-fragments of the state fragment. The second query in item 5(a)ii determines the left hand side of the role (in this case: the id of the fragment identified by the “\$base” expression), and the third query (item 5(a)iii) determines the right hand side. For each base, all results of the left hand side query are put in a role relationship with all results of the right hand side query.

The combiner function is a Java program that puts the pieces together and creates an *ALCCTL* model in an appropriate formal description language. Applied to the document model *D*, this results in the following verification model:

```

1 <model>
2   <state name="ID0001" startingState="yes">
3     <successor name="ID0002"/>
4     <concept name="Fragment">
5       <interpretation value="ID0001"/>
6     </concept>
7   </state>
8   <state name="ID0002" startingState="no">
9     <successor name="ID0003"/>
10    <successor name="ID0004"/>
11    <concept name="Topic">

```

```

12         <interpretation value="Data Structure"/>
13     </concept>
14     <concept name="Fragment">
15         <interpretation value="ID0002"/>
16         <interpretation value="ID00021"/>
17     </concept>
18     <role name="hasTopic">
19         <interpretation left="ID00021" right="Data Structure"/>
20     </role>
21 </state>
22 <state name="ID0003" startingState="no">
23     <successor name="ID0005"/>
24     <concept name="Topic">
25         <interpretation value="Data Structure"/>
26     </concept>
27     <concept name="Fragment">
28         <interpretation value="ID0003"/>
29         <interpretation value="ID00031"/>
30     </concept>
31     <role name="hasTopic">
32         <interpretation left="ID00031" right="Data Structure"/>
33     </role>
34 </state>
35 <state name="ID0004" startingState="no">
36     <successor name="ID0005"/>
37     <concept name="Topic">
38         <interpretation value="Binary Tree"/>
39     </concept>
40     <concept name="Fragment">
41         <interpretation value="ID0004"/>
42         <interpretation value="ID00041"/>
43         <interpretation value="ID00042"/>
44     </concept>
45     <role name="hasTopic">
46         <interpretation left="ID00041" right="Binary Tree"/>
47         <interpretation left="ID00042" right="Binary Tree"/>
48     </role>
49 </state>
50 <state name="ID0005" startingState="no">
51     <concept name="Fragment">
52         <interpretation value="ID0005"/>
53     </concept>
54 </state>
55 </model>

```

The SPARQL queries above can be encapsulated in a *VMSpecification* object in the implementation (cf. section 7.4).

Another advantage of having multiple views on a single document model is that different inference services can be applied to each view. For example, one view can incorporate topic specialisations by adding all generalisations of a topic to any fragment that deals with that topic, while another view is more strict and only contains the original topics.

SPARQL Recursion

As mentioned above, recursion is an important feature that SPARQL does not support “out of the box” in version 1.0. On the other hand, SPARQL 1.1, which does support recursion through so-called property paths, is not widely supported yet [HS12]. There exist approaches that use a

language extension to address this limitation [Ope], but they require a SPARQL query evaluator that understands these extensions. If the maximum recursion depth is limited, it is also possible to achieve this goal via a re-formulation of the query that is entirely conforming to the SPARQL specification. If the maximum recursion depth is not known, and if none of the existing language extensions is available, then using transitive roles might still be an option.

Description 9.3.5 (SPARQL Recursion by Reformulation). *SPARQL patterns of the custom form $\{ ?a ?b[min:max] ?c . ?c ?d ?e \}$, where min and max represent the minimum and maximum recursion depths, respectively, can be reformulated in standard SPARQL using cascading UNION expressions that make the recursion explicit.*

For example, the query $\{ ?s \text{ ref:part}[0:2] ?p . ?p \text{ vdk:id} ?d \}$ can be re-formulated as

```

1 {
2   ?s vdk:id ?d
3 } UNION {
4   ?s ref:part ?p1 .
5   ?p1 vdk:id ?d
6 } UNION {
7   ?s ref:part ?p2 .
8   ?p2 ref:part ?p3 .
9   ?p3 vdk:id ?d
10 }
```

Using this syntax, the query for the Topic concept in example 9.3.4 can be simply written as

```

1 SELECT ?d
2 WHERE {
3   ?s vdk:id "$state" .
4   { ?s ref:part[0:3] ?p . ?p dc:subject ?d }
5 }
```

SPARQL Optimisation

There exist several approaches to optimising SPARQL query evaluation in the literature.

ρ DF is an attempt to minimise the semantics of RDF by removing unnecessary predicates and keywords, while preserving the primary semantics. A sound and complete deductive system for ρ DF has been proposed, and it was shown that entailment of ground terms in the defined language fragment has a worst case complexity bound of $O(n \log n)$. [MnPG07]

[VRL⁺10] optimises SPARQL queries with a focus on group patterns with a single shared variable, so-called *star patterns*. A cost function for query execution plans has been developed, and a simulated annealing randomized algorithm is used to find the global optimum.

A mapping of SPARQL onto Datalog is proposed in [Sch07], which can be used as a basis for further optimisation.

Multiple approaches attempt to map SPARQL onto SQL. [Cyg05] provides a partial mapping onto the relational algebra, and [Har05] shows an early approach for evaluating SPARQL queries using database systems. Another early attempt uses caching [Dok06].

An efficient translation of SPARQL into SQL that covers all SPARQL language expressions is proposed in [ECTO09]. It creates an abstract query model to obtain a flat SQL query instead of cascading queries as many other approaches do.

Join-Order Optimisation [SMK97]

SPARQL evaluation on a relational database requires a large number of joins, especially if a triple-table based database schema is employed. It is therefore important to optimise the order of these join operations so that their overall execution becomes as efficient as possible. [SMK97] have conducted a survey over various optimisation techniques and found that both randomised and genetic algorithms provide the best combination result quality and runtime performance. These approaches regard the entire solution space, instead of relying on heuristics like minimum selectivity [SKS06].

Randomised algorithms randomly traverse the solution space, moving only between solutions that differ in a single step. They terminate after a predefined number of steps or when they become stuck, and return the best solution found during the traversal. Genetic algorithms require little specific knowledge about a problem and can be applied to a wide range of optimisation problems. Starting with a random subset of the solution space, a genetic algorithm retains the best elements of the current set and makes random changes to some, to produce a new set. This is repeated for a fixed number of iterations. The best element of the final set is returned.

A genetic algorithm requires a literal encoding schema for a solution, a cost function, a selection operation, a crossover operation, and a mutation operation. The encoding schema is used by the operators. For join-order optimisation, it can be based on a numbering of relations in the join, so that a solution is encoded as a sequence of numbers in the join order as described by the solution. For example, if the relations numbered 1 and 2 are joined first, then the encoding sequence starts with “12”.

The selection operation selects for good solutions based on the cost function (in the evolutionary sense). Better (“fitter”) solutions are retained with a higher probability than less adequate solutions.

The crossover operation combines existing solutions to obtain a new one, with the goal that the new solution should be better than its ancestors (again, based on the cost function). One way to attempt this for join-order optimisation is to select a sub-sequence of join operations from each parent, where each sub-sequence is of equal length and affects the same relations (albeit in a different order). These sub-sequences are then inserted into one of the other parents to obtain new solutions.

The mutation operation introduces new and random elements into an existing solution. In this case, randomly swapping the order of two join operations in a solution is sufficient.

The genetic algorithm produces an adequate solution in a short time.

There are several randomised algorithms worth considering for join-order optimisation [SMK97].

Iterative improvement begins by selecting a starting point at random. It then selects neighbours at random until one is found that is better than the current solution. If a better candidate is found within a fixed number of tries, this neighbour becomes the new current solution and its neighbours are examined in turn. Otherwise the current solution is declared a (local) minimum. This process is repeated several times for different starting points, and the best local minimum is returned. The approach produces an adequate solution in a short time.

Simulated annealing works in similar way, but also accepts new solutions that are worse than the current one. This acceptance depends on a continually decreasing probability. The algorithm therefore has the potential to “escape” local minima. It produces similar results to two-phase optimisation (see below), but has a much higher runtime cost.

Two-phase optimisation uses iterative improvement to find a number of local minima, then uses simulated annealing on the best of these minima. It produces the best solution in adequate time.

Toured simulated annealing selects several starting points based on a heuristic, then starts

simulated annealing in each. It produces results similar to two-phase optimisation.

Random sampling takes a truly random sample from the solution space and selects the optimum from this sample. This algorithm is based on the analysis result that a large number of solutions in the solution space is close to the optimum, which should then also be the case for the sample set. The algorithm is a good choice when adequate solutions need to be found very fast.

Conclusion

In this chapter, we have examined several practical options of obtaining a semantic document model from a document. We have developed a set of extraction rules for this purpose, based on the abstract transformation rules shown in chapter 5. We have shown that these rules can be easily adapted with new background knowledge to other document types and domains.

We have also regarded inference procedures for document models, and have proposed a method for implementing generalisation semantics on individuals in description logics.

The possibilities and the implementation of obtaining other models from a semantic document model by specifying and materialising views on the document model were discussed. Finally, we gave a brief overview on optimisation options for SPARQL query evaluation.

Chapter 10

Use Cases

In this chapter, we will discuss several use cases that we applied our implemented approach to. Further details on some of the documents, the background knowledge used, and the extraction rules can be found in appendix B. Details on how much (if at all) the extraction rules had to be adapted to the new domain can be found in section 11.4.

10.1 Document Verification

Verification of content consistency criteria for documents was the primary goal of the VERDIKT research project [FWJS08]. Consistency criteria can specify both the relative order along a reading path within a document and the coherence of the content of different parts of a document.

For example, the criterion “*After every definition, there must be an example with the same topic*” specifies a relative order (first the definition, then the example), it specifies the content types (definition and example), and it specifies the coherence of the content (same topic).

We applied the approach proposed in this thesis to documents from different domains, with the aim of using formal verification techniques on non-formal documents.

10.1.1 Realistic Technical Documentation Use Case

The first domain is that of technical documentation. We created several documents, implemented in HTML, for fictional devices that were nonetheless modelled after real-world documentations. [SWJF09, SWF11]

Figures 10.1 and 10.2 show outlines of documentations for digital cameras, called VDK 1501 and VDK 1109, respectively. Both documentations have been modelled after actual documentations for consumer-type cameras. Figure 10.3 shows the outline of a documentation for a digital satellite receiver called VDK 1108. The latter is based on an actual document in PDF format of 80 pages, grouped into 24 chapters. It has been converted to HTML for easier technical access, and any names and other identifying features have been changed to protect the identity of the original manufacturer.

It has been our observation that many technical documentations have a similar structure: a linear primary reading path, some cross references, and a few hubs with (sometimes bidirectional) references to most of the other parts of the document (like the table of contents or an index). In general, the number of references is low, which is reflected in a relatively low node-degree (generally below 10) in the RDF/OWL implementation of the semantic document model. The out-degree is dominated by statements with literal values.

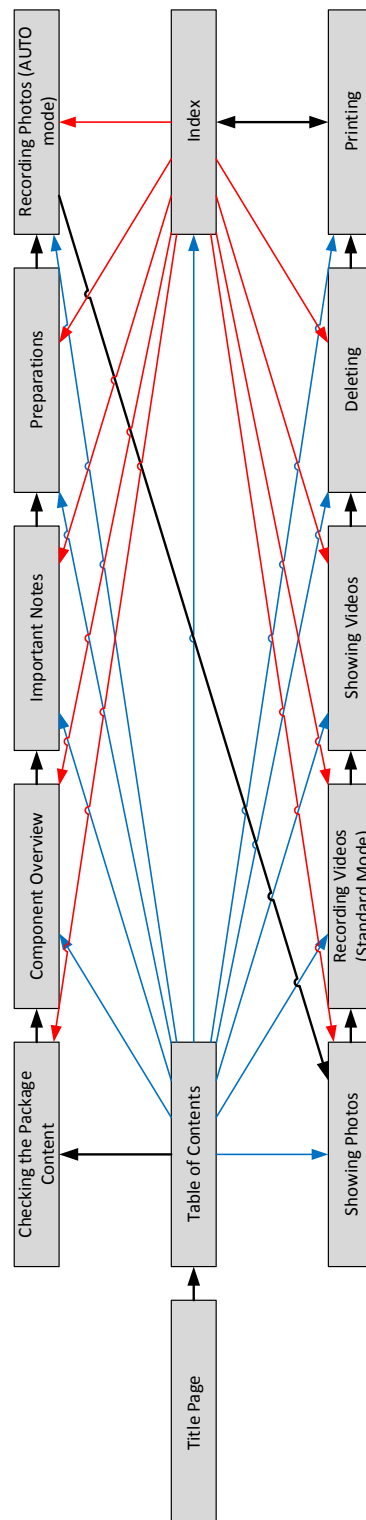


Figure 10.1: VDK 1501 document. The primary reading path is shown in black, references from the *Table of Contents* are shown in blue, and references from the *Index* are shown in red.

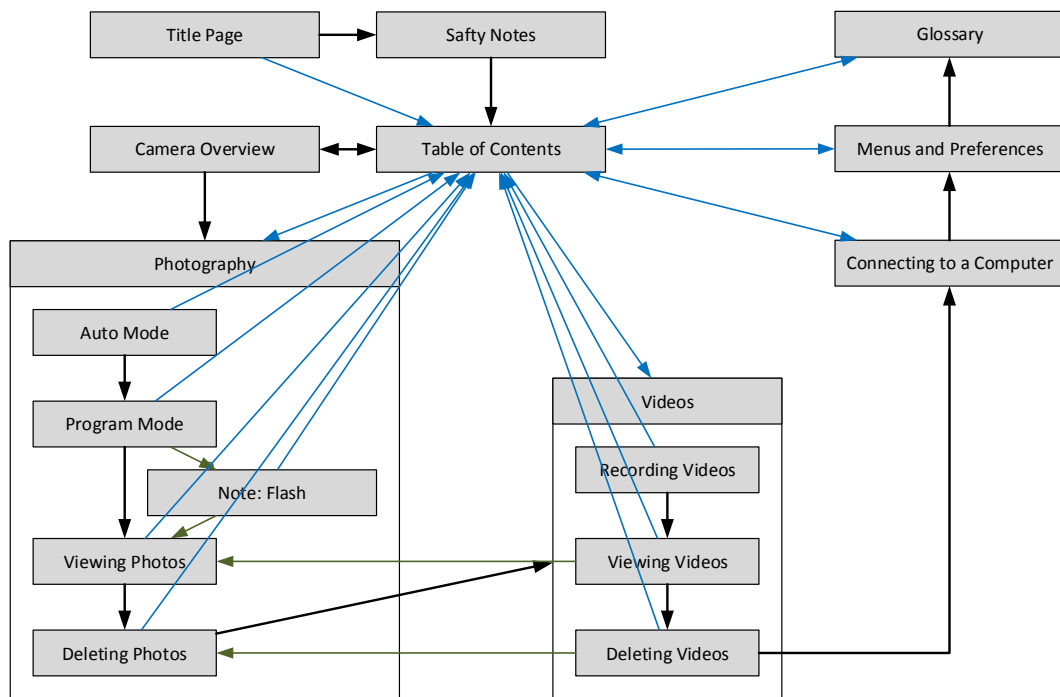


Figure 10.2: VDK 1109 document. The primary reading path is shown in black, references to and from the *Table of Contents* are shown in blue, and cross references between sections are shown in green.

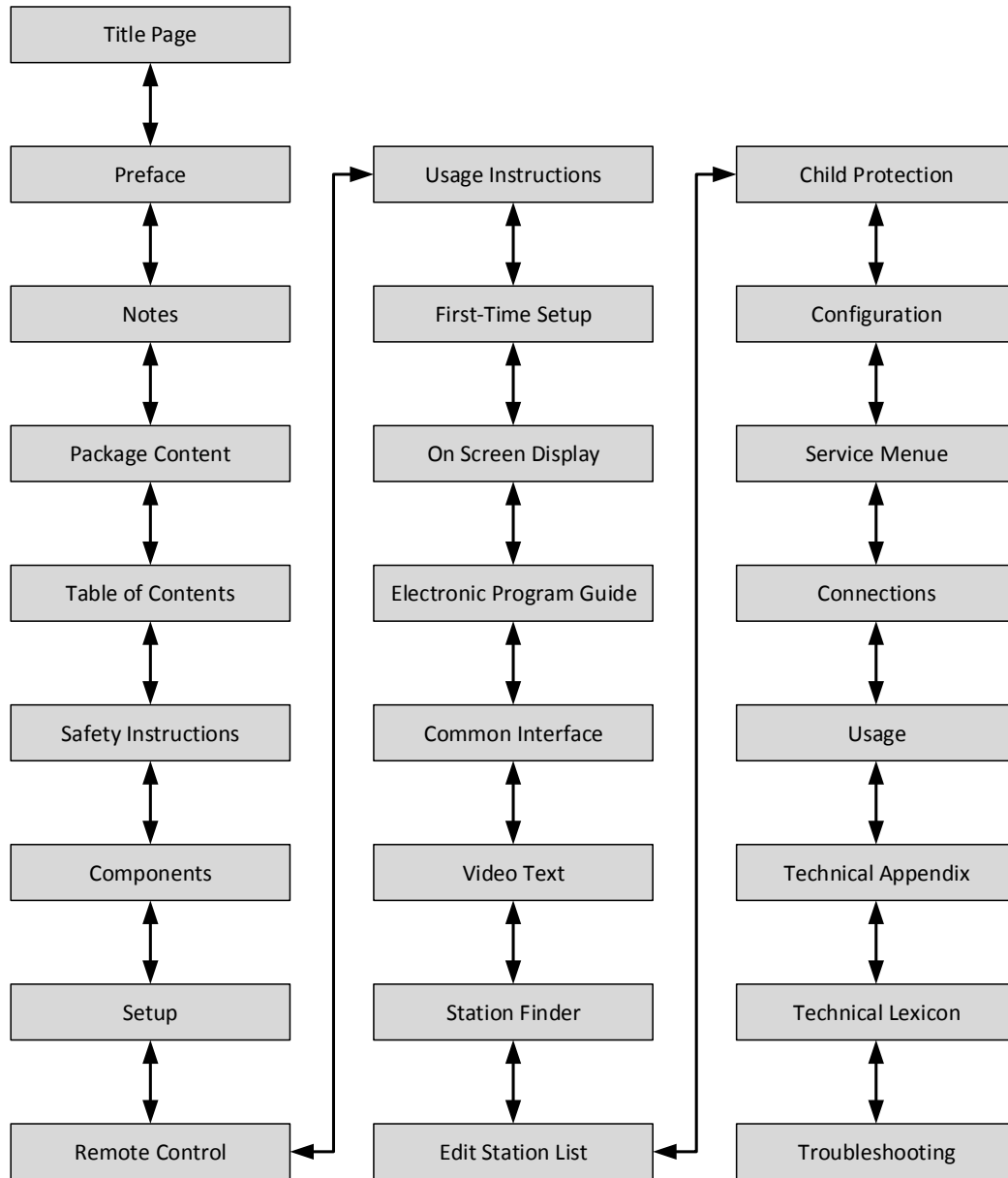


Figure 10.3: VDK 1108 document. The primary reading path is shown in black. Every section except the *Title Page* has a bidirectional reference to and from the *Table of Contents* (not shown).

For obtaining the semantic document models of these documents, we used the default rules as defined in description 9.1.8 and background knowledge about the formatting, about keywords (such as “side note”), about relevant terms (such as “memory card” or “lens”), about equivalent terms (such as “standby” and “stand-by”), and about abbreviations (such as “HDMI”, “VHS”, or “YUV”).

A single view on the level of chapters was created for each document. The views are *ALCCTL* models containing several concepts and roles.

On the first document (VDK 1501), we checked the following six criteria:

1. Every component named in the *Overview* must be referred to later, but before the *Index*.
2. There must be no reading path that skips the *Important Notes*.
3. Every term in the *Index* must be described somewhere before in the document.
4. Every term described somewhere in the document must be listed in the *Index*.
5. All images must be available in either JPEG or PNG format.
6. The chapter *Checking the Package Content* must follow immediately after the *Table of Contents*.

On the second and third document (VDK 1108 and VDK 1109), we checked the following nine criteria:

1. Every abbreviation has to be explained later in the *Glossary*.
2. Every technical term has to be explained later in the *Glossary*.
3. The topic of side notes is related to the topic of their referring content units.
4. No warning may exist only inside of an excursion/side note.
5. There must not be any path that skips even a single warning.
6. All components listed in the *Overview* must be discussed before the *Glossary*.
7. Sections with related topics must reference each other.
8. The targets of all cross references must have at least one topic in common with their referrers.
9. The targets of all cross references must have at least one topic that is related to at least one topic of their referrers.

Unsurprisingly, most of the criteria were not satisfied.

The software developed during the VERDIKT project provides an end user with an error report about which criteria failed, and why they failed. Formally, the report contains counterexamples for the *ALCCTL* formulae.

Including location data in the semantic document model and in the *ALCCTL* view allows us to provide a link for each counterexample that, when clicked, leads the user to the section of the original document where the error is located. In some cases it is even possible to highlight the error. This location data is annotated to every fragment in the semantic document model, and to every state in the *ALCCTL* model (not shown in previous examples). It consists of an XPath

expression that identifies the location in the source file represented by a media object in the base document model. [SWF11]

The documents in this domain have an extra benefit: they can be used to showcase some of the principles of our research to the general public. Many people have had negative experiences with sub-par technical documentations and can therefore relate to the use case. This expectation was confirmed at a university open house presentation.

10.1.2 Real E-Learning Use Case

The second domain we regarded is the e-learning domain [SJWF09]. We examined over 100 e-learning documents from the WWR project [WWR04], encoded in a specialised XML format called ML3 [TLV03]. Figure 10.4 shows a glimpse of one of these documents (in German).

These documents have a very complex structure. There are many cross references, in particular links to related topics and to side notes. In addition, there are three so-called “dimensions”: the target *medium*, the target *audience* and the level of difficulty, also called *intensity*. Possible target media are *print* (for offline review), *screen* (for online review) and *presentation* (slides for use in a lecture). Target audiences are *student* and *teacher*. Intensities are *basic* (which includes only the most basic material), *advanced* (which includes more advanced topics), and *expert* (which includes the most complex topics). All content is classified along all dimensions, with multiple classifications possible. For example, one document fragment may be classified as suitable for advanced and expert students, to be perused on- and offline. Another fragment may be intended for teachers of all levels to be viewed online, because it contains notes for a lecture and a video clip that is not suitable for printing.

The RDF/OWL implementations of the semantic document models for these e-learning documents reflect this complexity. While the documents vary widely in size (between 300 and 10.000 statements per document, and between 100 and 1.000 individual nodes per document), the node degree remains almost uniformly high (averaging between 7 and 12). While they also contain many literal-valued statements, the e-learning documents clearly have a far more complex structure than the technical documentations.

In the verification process, the best way to handle the complexity caused by the different dimensions is to create a separate view for each combination, by filtering the appropriate fragments from the document model. This can be easily done in the SPARQL queries used to specify such views. It results in 18 distinct views to which verification criteria can be applied, not counting additional views for different structural levels.

A different approach to creating multiple verification models is to create multiple formulae. For example, the formula for the criterion “after every definition, there is an example with the same topic in each of its direct successors” is $\text{Definition} \sqsubseteq \text{AXExample}$. To ensure that the intensity is consistent, i.e., that both the definition and the example have the same intensity, the following formula can be used (analogue for medium and audience):

$$\text{AG}((\text{Definition} \sqsubseteq \text{AXExample}) \wedge (\text{Intensity} \equiv \text{AXIntensity})).$$

However, this significantly increases the complexity of the formulae, which are already too complex for laypersons to handle. Worse, it is only possible in special cases. For different temporal operators, like AF, EX or EF instead of AX, it cannot be guaranteed that the state containing the example is the same state that also contains the appropriate intensity, i.e., EFExample and EFIntensity may not be evaluated in the same state. \mathcal{ALCCTL} does not allow two description logic subsumptions on a single pair of states.

We applied several verification criteria to these documents, among them the following:

Verbundprojekt im Rahmen des **bmwif**-Programms „Neue Medien in der Hochschullehre“

WWF
wissenswerkstatt
rechensysteme

selbststudium vertiefung
vortragsfolien aufbau basis

printversion
about

Formale Grundlagen von Schaltnetzen

LE 1: I. Grundlegende Begriffe und Notationen

Lernziele

- Boolesches Wort, Boolesche Sequenz
- Verhalten und Struktur von Schaltnetzen
- Notationen

Zusammenfassung
Übung

II. Boolesche Funktionen und Schaltfunktionen

LE 2: Verhaltensbeschreibung

Lernziele

- Boolesche Funktionen
- Dualzahlen
- Basiskomponenten von Schaltnetzen

Zusammenfassung
Übung

LE 3: Strukturbeschreibung

Lernziele

- Strukturbeschreibung durch Graphen
- Schaltnetzstruktur
- Basiskonventionen zur Beschreibung der Schaltnetzstruktur

Zusammenfassung
Übung

LE 4: Erweiterte Konzepte

Lernziele

- Analyse
- Von Booleschen Werten zu Booleschen Wörtern
- Schaltungsoptimierung
- Modularer Aufbau

Zusammenfassung
Übung

III. Boolesche Algebra

LE 5: Termdarstellung

Lernziele

- Grundlegende Definitionen
- Definition Boolescher Term
- Termdarstellung - Wertetabelle
- Operatoren und partielle Ordnung

Zusammenfassung
Übung

Gesetze der Booleschen Algebra

1 2 3 4 5 6 7 8 9 10 le
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 pe

Boolesche Funktionen und Schaltfunktionen · Strukturbeschreibung · Schaltnetzstruktur

Definition Schaltnetzstruktur

→ Lernziele

Abbildung 13: Graph mit explizit gerichteten Kanten

Eine **Schaltnetzstruktur** ist ein (von seinen Eingängen zu seinen Ausgängen) gerichteter **azyklischer Graph** (Kantenfolgen über die **Knoten** sind nicht geschlossen), dessen **Knoten** mit Funktionsnamen markiert sind.

Def. "Schaltnetzstruktur"

Abbildung 14: Die gestrichelte Linie definiert einen Zyklus und damit eine verbotene Struktur

Eine wichtige Struktureinschränkung ist das Verbot, Ausgänge auf Eingänge von Komponenten zurückzuführen, also eine geschlossene Kantenfolge über Komponentenknoten zu erzeugen. Ein Graph, bei dem dies nicht zugelassen ist heißt azyklisch.

"Schaltnetze sind zykliefrei"
"azyklischer Graph"

"Knoten" "Strukturbeschreibung" "Schaltfunktion"

Figure 10.4: Sample of a WWR e-learning document (German)

1. After every definition, there is an example with the same topic.
2. Every learning objective for a chapter is covered in the same chapter.
3. All relevant terms must also be the topics of a fragment.

While not all of these criteria were satisfied, the results were noticeably better than for the technical documentations. Without terminological background knowledge, the number of formula violations increases sharply because different dictions of the same term are no longer recognised.

Note that a formula violation does not necessarily mean that there is an error in the document. There could instead also be an error or an omission in the background knowledge, there could be an error in the document model, there could be an imprecision in the formula, or there could simply be an exception to the rule. Formula violations merely give an indication to a human author or editor that there *might* be an error in the document.

We also used the e-learning documents to obtain background knowledge from them, as discussed in section 6.2.

10.1.3 Real Lecture Notes Use Case

As an additional domain, we investigated university lecture notes, written in \LaTeX . Their handling was challenging on a technical level, since \LaTeX is a very complex format. Otherwise, they proved to be very similar to the e-learning documents, which may not be entirely surprising.

We used a preprocessor to obtain XML code from the source files, so that we were able to use the same processing workflow as in the other domains. Since existing \LaTeX to XML converters focus on (X)HTML and on reproducing the original visual output, thus losing structural information like “section” commands, we created our own parser. Its functionality is reduced to converting commands and environments, while it ignores mathematical formulae and other things that are not relevant for our purpose.

To increase its precision, in particular regarding the correct recognition of parameters, we compiled a list of \LaTeX commands and parameters that were frequently used in the lecture notes. This list was also used as background knowledge on structural indicators.

The verification criteria and verification results were similar to those of the e-learning documents.

10.2 Process Verification

Next to document verification, we decided to try and use the same techniques for a different purpose: the verification of consistency criteria on processes. As we have explored in section 4.2.2, processes can be seen and modelled in a way that is very similar to documents. We indeed managed to show that the same approach that we developed for documents can be applied to processes with very little adaptation.

We obtained a collection of documents that describe software engineering processes from the German Aerospace Center (DLR). They specify one “entity” (a super-process, if you will), consisting of 18 individual processes. These documents do not contain a formal specification of the processes, however, but rather a semi-formalised specification written in Microsoft Word, using lists, tables, and formatted text. Each process is described in a single file, an obfuscated excerpt can be seen in figure 10.5. Because of the documents’ confidentiality, formalised in a non-disclosure agreement, we cannot show the actual content here.

Entity Name & Entity Path	
Entity Name:	E.Name of Sub-Entity (E.xxx)
Entity Path:	E.Name of Main Entity (E.xxx) - E.Name of Sub-Entity (E.xxx)

Process Identification	
Process Name	Process Name
Process ID	QMH-EOC-E.xxx-E.xxx-EP##
Process Manager	

Purpose of Process

Text ...

Product In

- Process input 1
- Process input 2
-

Process

- Action 1
- Action 2
- ...

Product Out

- Process output 1 (Product 1: ID, Name)
- Process output 2 (Product 2: ID, Name)
-

Figure 10.5: DLR document layout

After writing a preprocessor that transformed the Word documents into suitable XML files, we were able to employ the methodology developed for documents. We used background knowledge about the documents' structure, formatting and keywords, and we used the extraction rules defined above to obtain a formal process model from these files. A small part of this model is shown in figure 10.6.

Note that the preprocessor uses the Word documents in `docx` format. A previous version based on the Visual Studio Tools for Office (VSTO) that used the documents in the older `doc` format proved to be unsuitable due to the inefficiency and instability of the VSTO.

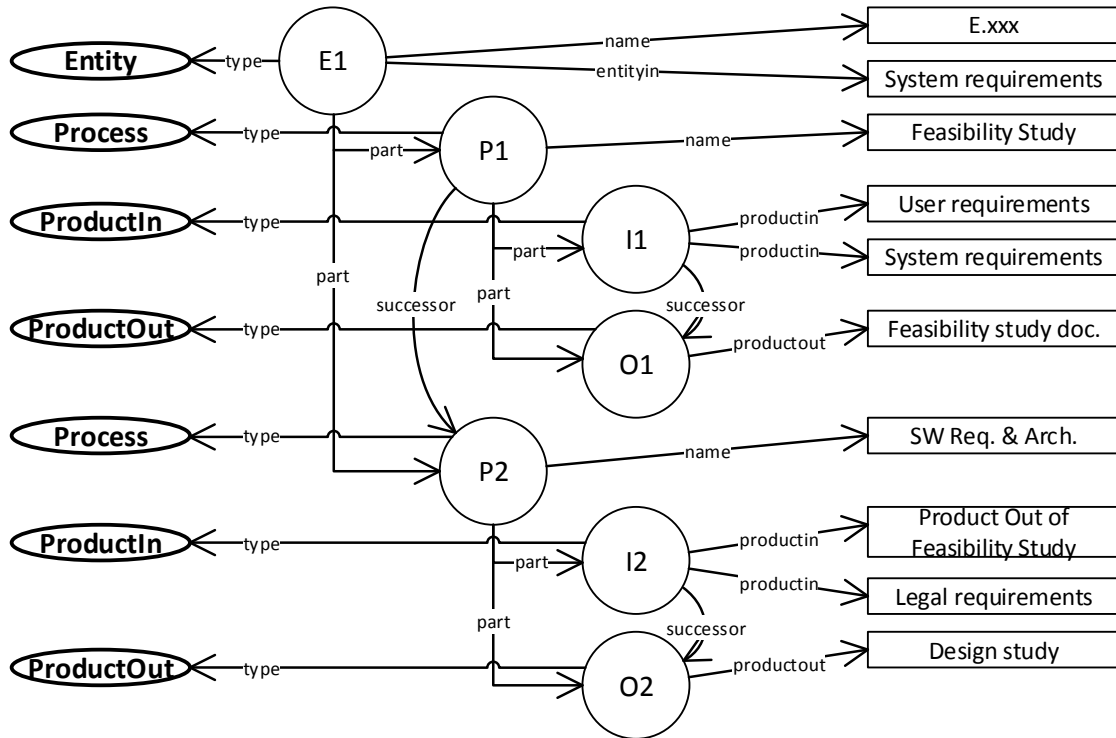


Figure 10.6: DLR process model (excerpt)

We found that the concept of views is particularly useful for processes, because it enables us to check process structures without necessarily knowing every detail of every sub-process. This is illustrated in figure 10.7, which shows two views on the same set of processes. The first view shows each process separately, and the second view shows the processes aggregated by entity.

One of the criteria we used, namely that “every manager is part of the active personnel” (see below), uses background knowledge about current employment records that is integrated directly into the view in the form of concept interpretations.

The RDF/OWL implementation of the semantic process model consists of approximately 2000 statements and 200 distinct RDF nodes for the entire set of processes. It has an average node degree of 10, which is high in relation to the other models we encountered. This characteristic is reinforced by the low number of literal statements, meaning that the process structure is remarkably complex.

In process verification, criteria can often be classified as those dealing with the *control flow*, such as “there must be a test before deployment”, and those dealing with the *data flow*, such as

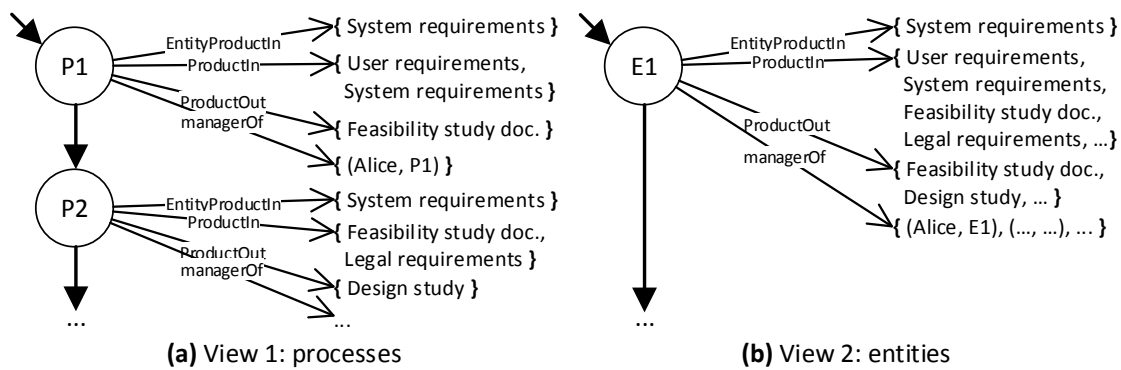


Figure 10.7: Views on the DLR process model

“all data needed by any process step must be created by an earlier process step”. The latter is only rarely supported by verification systems [TvdAS09]. However, we were able to pursue both types of criteria. The following criteria were developed in consultation with domain experts from the DLR:

1. All “product in” are either produced by at least one entity process (“product out”) or are part of the “entity product in”.
2. All “product out” are either consumed by at least one entity process (“product in”) or are part of the “entity product out”.
3. Every process has a manager.
4. Every manager is part of the active personnel.
5. No process has different managers in different locations.
6. There is an entry in the document change record about the current process version.
7. No process is reviewed by its own manager.
8. No process is approved by its own manager.
9. No process is released by its own manager.
10. Every process is prepared by its own manager.
11. Every process is prepared by someone.
12. Every process is reviewed by someone.
13. Every process is approved by someone.
14. Every process is released by someone.

Only the first two criteria were violated, speaking either for the high quality of the process specification or for the omission of an important criterion (or both).

The efficiency and runtime performance of the entire process, from reading the document models to verifying the criteria, is suitable for direct interaction with the system instead of overnight processing.

\mathcal{ALCCTL} Formula Parametrisation

The criterion that “no process has different managers in different locations” requires a statement about two distinct entities, i.e., an \mathcal{ALCCTL} formula with two variables. This is, however, not supported by the semantics of \mathcal{ALCCTL} . The corresponding formula $AG(AG\forall\text{managerOf.Process} \sqsubseteq \forall\text{managerOf.Process})$ ensures that the managers on the left and right hand side of the \sqsubseteq are the same, but there is no guarantee that the processes are also the same. We solved this problem by introducing the parametrisation of \mathcal{ALCCTL} formulae:

$$AG(AG\forall\text{managerOf}\{ \$Process \} \sqsubseteq \forall\text{managerOf}\{ \$Process \})$$

The parameters are resolved in three steps:

1. collect the names of all parameters;
2. for every \mathcal{ALCCTL} concept named like one of the parameters: expand the values of its interpretation to a set of new concepts, with one concept for each value; and
3. replace a formula with a parameter that refers to a concept with a set of formulae, where each formula uses a different one of these new concepts instead of the parameter or the original concept.

This is done successively until all parameters have been replaced. Note that resolving the parameters not only changes the formula and generates new formulae, but it also changes the verification model by adding new concept interpretations. Also note that the parameters can only be resolved against an existing \mathcal{ALCCTL} model. Resolving them against different models will yield different results.

Description 10.2.1 (Resolving \mathcal{ALCCTL} Formula Parameters). *The following pseudo-code shows the algorithm for resolving parameters in an \mathcal{ALCCTL} formula against an \mathcal{ALCCTL} model.*

```

1 public List<String> resolveParameters(String formula, ALCCTLModel model) {
2     Set<String> parameters =
3         /* get the names of all parameters from the formula */
4     List<String> formulae = new ArrayList<>();
5     for (String parameter: parameters) {
6         Set<String> values =
7             /* get all concept interpretations from all states
8              for the concept named as the parameter */
9         for (State state: model.getStates())
10            for (String value: values)
11                /* add a new concept interpretation to the state,
12                 with "Param_" + value' as the concept name,
13                 and 'value' as its sole value */
14            List<String> newFormulae = new ArrayList<>();
15            for (String formula: formulae)
16                for (String value: values)
17                    /* add a new formula to newFormulae where every
18                     usage of 'parameter' is replaced by
19                     'Param_' + value' */

```

```

20     formulae = newFormulae;
21 }
22 return formulae;
23 }

```

Example 10.2.2 (Resolving \mathcal{ALCCTL} Formula Parameters). *Recall the formula f from above:*

$$\text{AG}(\text{AG}\forall\text{managerOf}\{ \$Process \} \sqsubseteq \forall\text{managerOf}\{ \$Process \})$$

Let $M = (S, R, I)$ be an \mathcal{ALCCTL} temporal model, with $S = \{s_1, s_2, s_3\}$, $R = \{(s_1, s_2), (s_2, s_3), (s_1, s_3)\}$, $\Delta^I = \{P1, P2, P3\}$, $I = \{(s_1, (\Delta^I, ()^{I(s_1)})), (s_2, (\Delta^I, ()^{I(s_2)})), (s_3, (\Delta^I, ()^{I(s_3)}))\}$, $\text{Process}^{I(s_1)} = \{P1\}$, $\text{Process}^{I(s_2)} = \{P2\}$, and $\text{Process}^{I(s_3)} = \{P3\}$.

Resolving f against M yields the set of formulae $f' = \{ \text{AG}(\text{AG}\forall\text{managerOf}\text{Param_P1} \sqsubseteq \forall\text{managerOf}\text{Param_P1}), \text{AG}(\text{AG}\forall\text{managerOf}\text{Param_P2} \sqsubseteq \forall\text{managerOf}\text{Param_P2}), \text{AG}(\text{AG}\forall\text{managerOf}\text{Param_P3} \sqsubseteq \forall\text{managerOf}\text{Param_P3}) \}$. M is extended with $\text{Param_P1}^{I(s_1)} = \{P1\}$, $\text{Param_P2}^{I(s_2)} = \{P2\}$, and $\text{Param_P3}^{I(s_3)} = \{P3\}$.

This approach can increase the number of formulae (and thus the complexity of model checking) significantly: the factor of this increase is in $O(|P| \cdot |\Delta^I|)$, where $|P|$ is the number of parameters and $|\Delta^I|$ is the size of the interpretation domain. However, it also increases the expressive power of an \mathcal{ALCCTL} formula, which can make the additional cost worthwhile in some circumstances.

Note that usually $|P| = 1$ and that only a small subset of Δ^I is required.

10.3 Other Use Cases

To prove the concept, we tested other domains and document formats. In the e-learning domain, we also applied the approach to documents in LMML format [SF99], with results similar to those for ML3 documents. For technical documentations, we also successfully tested the DocBook [Wal09] and DITA [DEAJ10] formats [Pre08], using real-world documentations for various Linux systems and components. We checked the criterion “Every technical term has to be explained later in the *Glossary*” and found it largely unsatisfied.

In addition, we used formal process specifications written in BPMN [All10] and other formats, courtesy of the German National Process Library [Ahr12], to show the applicability of our approach to formal specifications of different domains. We even successfully obtained a formal process model from a purely graphical process description in Microsoft Visio format. Additional details can be found in section 11.2 and in appendix B.

Conclusion

In this chapter, we have shown the formal verification of consistency criteria on documents from different domains as a use-case for our approach. We have also shown that the techniques can be transferred to the validation of processes.

Furthermore, we have extended the expressive power of \mathcal{ALCCTL} in such a way that for a known model, it is possible to specify formulae over more than a single distinct variable. We have given an upper bound for the increase of complexity of model checking such formulae.

Part IV
Evaluation

Chapter 11

Quantitative Evaluation

In this chapter, we will evaluate the implementation of the proposed approach according to quantitative criteria. The evaluation system on which all measurements were made is detailed in description 11.0.1. All measurements were made five times in direct succession. The results presented here are the average values of the final three measurements to minimise the influence of external factors like system caching.

Description 11.0.1 (Configuration of the Evaluation System). *The evaluation system has been configured with the following hard- and software components:*

- ▶ *Intel(R) Core(TM) i7 Q720, 1.6 GHz*
- ▶ *8 GB RAM*
- ▶ *256 GB SSD (system, applications)*
- ▶ *500 GB SATA-II HDD 7200rpm (database, data)*
- ▶ *Microsoft Windows 8 64-bit*
- ▶ *Microsoft SQL Server 2012 64-bit*
- ▶ *Java 1.7*

11.1 Generated Documents

In this section, we will describe how the processing times of various processing steps scale in relation to increased document sizes, increased number of processing rules, or increased amount of background knowledge.

We will use the following abbreviations when referring to different processing steps:

R.Comp.	compilation of the processing rules
L.XML	loading the XML files into main memory
Prep.	preprocessing
R.Exec.	executing the processing rules to obtain a document model
V:XML	generating an XML-based view on the document model
V: <i>ALCCTL</i>	generating a view for <i>ALCCTL</i> verification
V:BGK	generating a view containing background knowledge
Verif.	running the <i>ALCCTL</i> verification

Note that the processing rules only need to be re-compiled (R.Comp.) whenever the rule set changes, which usually happens only rarely. Most changes to the background knowledge do not require changes to the rule set, and thus no re-compilation.

First, we regarded how the processing times scale with the size of the source document. We used a set of three processing rules, and background knowledge with ten different indicators each for fragment elements and for data elements. The source documents contained increasing numbers of XML elements, but only ten different element names to match the indicators in the background knowledge. We created documents of increasing size and measured the processing times for each document, as shown in figure 11.1.

Since the rule set remains unchanged, the rule compilation time stays virtually the same. Loading the XML files scales roughly in linear time, as is to be expected. The same holds true for creating the XML-based view from the document model. Executing the rule set to obtain the document model from the source document, however, requires time that grows polynomially in the number of XML nodes (about $|N|^{1.35}$, where N is the number of nodes; $|N|^{1.6}$ on a different test machine). Since the Drools rule engine is based on the RETE algorithm which in general scales polynomially in the size of the data set, this too is not unexpected. The absolute values (about 16 seconds for processing 1,000 nodes) are still in the realm of usefulness. Similar results were obtained on a set of generated HTML documents in a separate experiment.

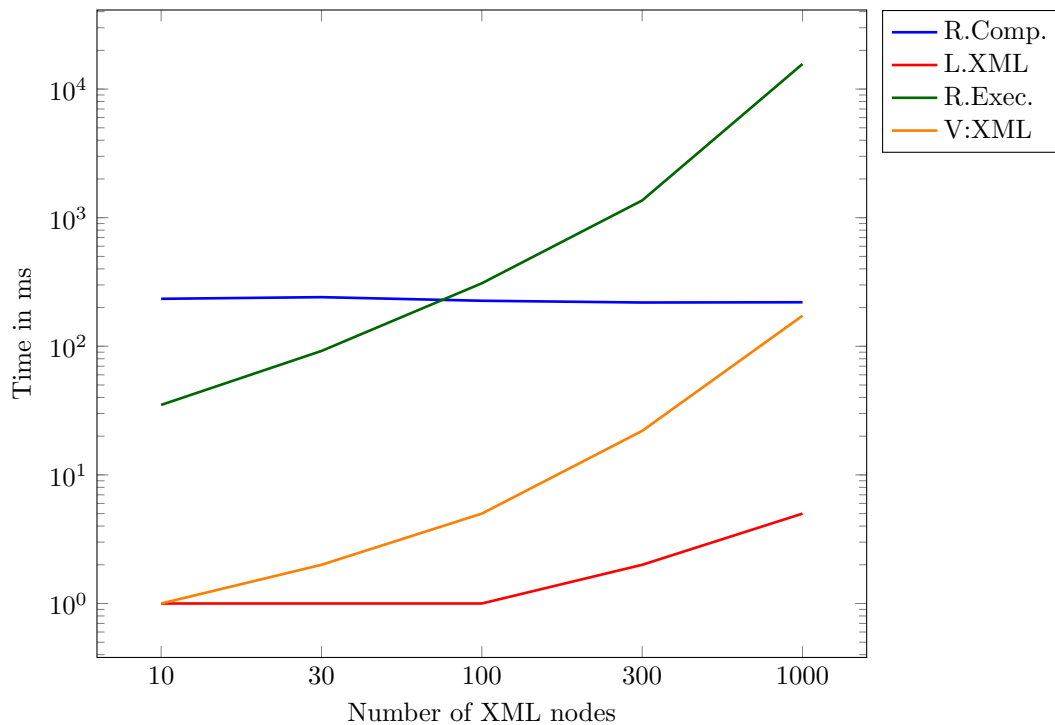


Figure 11.1: Time measurements for the processing time of documents with a varying number of XML nodes

Next, we investigated how the processing times scale with the number of processing rules. To this end, we removed the background knowledge that contained the indicators for fragments and

data elements, and wrote this information directly into the rules. One rule was required for the element recursion, and one rule each for every fragment indicator and for every data indicator. Ten different elements in the source file thus lead to $1 + 2 \cdot 10 = 21$ rules. We created rule sets for increasing numbers of *different* elements in the source documents, while the total number of elements remained constant (1,000).

Here, the compile times for the rules sets (as shown in figure 11.2) grow polynomially with the number of rules. Since rule sets only rarely need to be compiled, this is not overly worrisome. Disregarding some caching effects, the times for loading the XML files and for creating the XML views are constant, as expected for documents of constant size. The rule execution times start at about the same value where they were for a document of the same size in figure 11.1, and grow in a linear manner because the time required to find a matching rule increases with the number of rules.

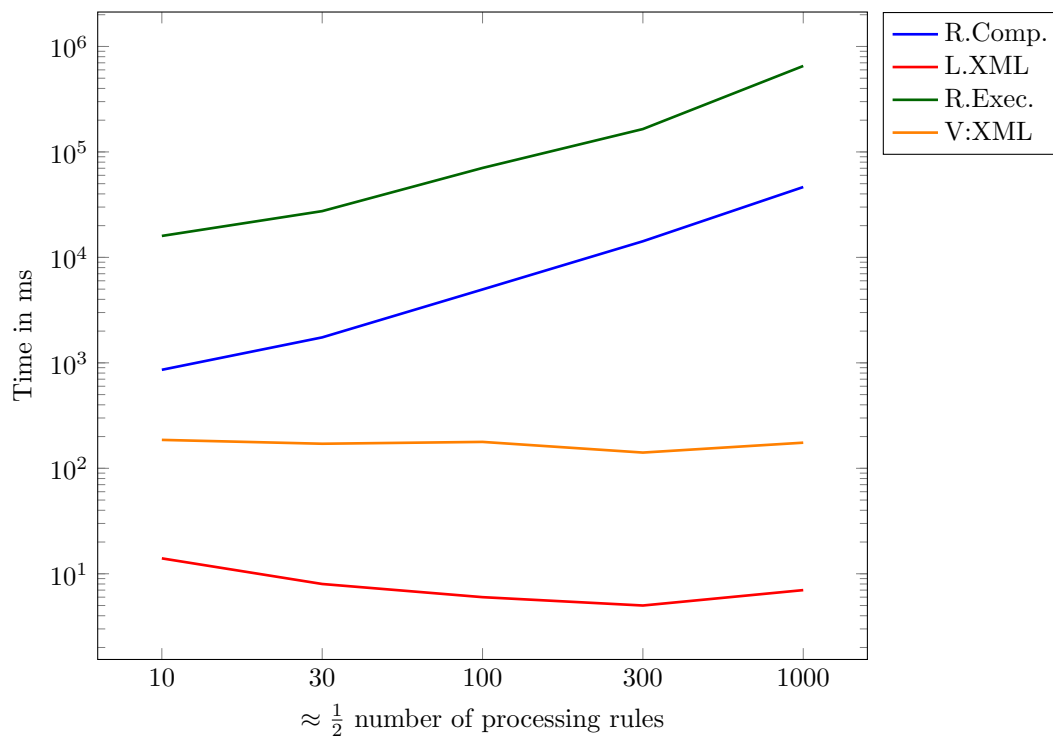


Figure 11.2: Time measurements for the processing time of documents with varying number of processing rules

Finally, we restored the rule set to the original set of three, but we kept the increasing number of different elements in the source files (as well as the total of 1,000 elements), moving the growing number of indicators into the background knowledge. The results are shown in figure 11.3.

Using background knowledge instead of fixed rules scales very similar to increasing the number of rules. Since the expressive power remains completely identical, this result was expected. In real use cases, there are usually no more than 20 indicators, so even a runtime of about 10 minutes for 1,000 indicators does not have any practical impact.

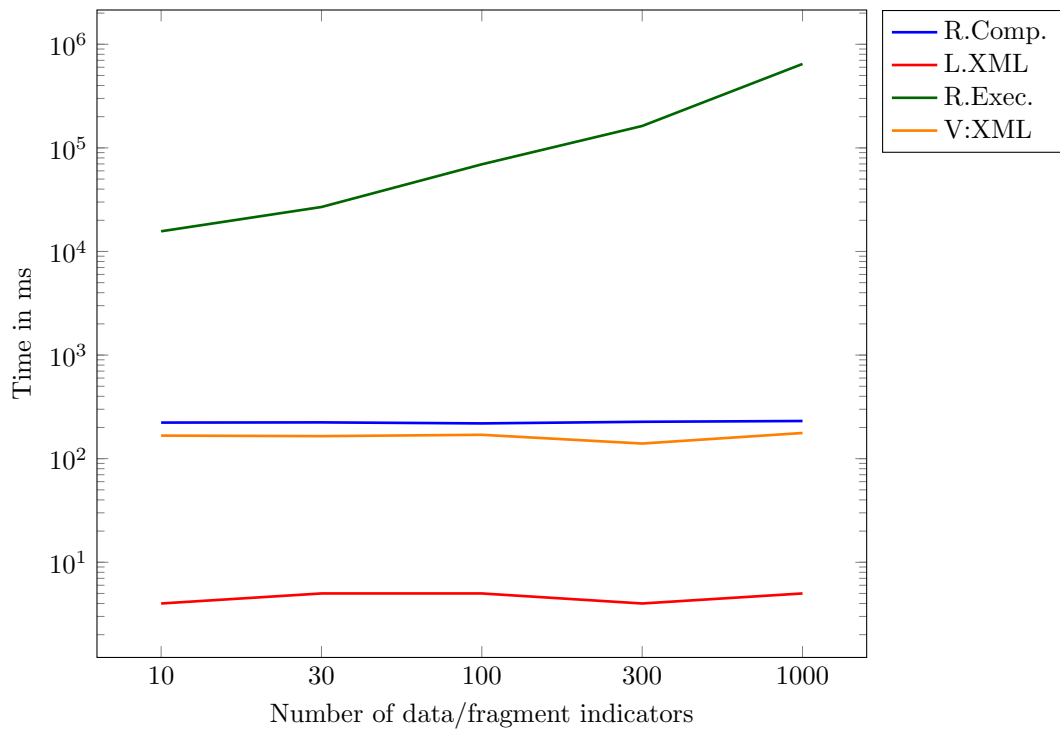


Figure 11.3: Time measurements for the processing time of documents with varying number of data and fragment indicators in the background knowledge

11.2 Real Documents from Different Domains

In the following sections we will list the size (according to different measures) of all documents and of all document models that were regarded in the evaluation, followed by the processing times of obtaining document model from the documents, and of obtaining (and materialising) views on these models. We will analyse what causes the different characterising properties of the documents and document models, as well as the extraction times.

In the tables in this section, we will use the following abbreviations with respect to a single XML-based document or a single RDF-based document model:

$ N $	total number of XML/RDF nodes
$ T $	total number of XML text nodes
$\min(A)$	minimum number of attributes found in an XML node
$\max(A)$	maximum number of attributes found in an XML node
$\text{avg}(A)$	average number of attributes found in an XML node
$\text{mean}(A)$	mean number of attributes found in an XML node
$\min(C)$	minimum number of child nodes found in an XML node
$\max(C)$	maximum number of child nodes found in an XML node
$\text{avg}(C)$	average number of child nodes found in an XML node
$\text{mean}(C)$	mean number of child nodes found in an XML node
$ S $	total number of RDF statements
$ R $	total number of <i>distinct</i> RDF resources
$ L $	total number of <i>distinct</i> RDF literals
$\min(D)$	minimum node degree found in any RDF node
$\max(D)$	maximum node degree found in any RDF node
$\text{avg}(D)$	average node degree found in any RDF node
$\min(I)$	minimum node in-degree found in any RDF node
$\max(I)$	maximum node in-degree found in any RDF node
$\text{avg}(I)$	average node in-degree found in any RDF node
$\min(O)$	minimum node out-degree found in any RDF node
$\max(O)$	maximum node out-degree found in any RDF node
$\text{avg}(O)$	average node out-degree found in any RDF node
$ F $	total number of fragments in the document model

Since $\min(A)$, $\min(C)$, $\min(D)$, $\min(I)$, and $\min(O)$ always equal zero in the documents and document models regarded here, they have been omitted from the tables below. For the XML documents, the existence of empty nodes or nodes with no attributes is unsurprising because of the size of the documents. For the RDF-based document models, a minimal node-degree of zero is caused by classes or properties that are defined in the schema, but that are not used in the model. In addition, every root node has an in-degree of zero, and every literal node has an out-degree of zero. The corresponding \max values have also been omitted, because they carry little relevant information.

We will use the following abbreviations with respect to all documents or to all document models:

sum	the sum total of all values
avg	the average of all values
mean	the mean of all values
\min	the minimum of all values
\max	the maximum of all values
md.	the mean deviation of all values
msd.	the mean square deviation of all values

For example, in table 11.1, the value for $sum:avg(A)$ in row one, column four is the sum over the average number of attributes in each XML document. The value for $avg:mean(C)$ in row two, column six is the average over all maximum numbers of child nodes. Finally, the value for $md:|N|$ in row six, column one is the mean deviation for the number of nodes in each document.

Remark 11.2.1. *The average of a set of values is obtained by calculating the sum of all values divided by the number of values.*

The mean of a set of values is obtained by putting all values into an ordered list and by selecting the centre value from this list.

11.2.1 E-Learning Documents

In the e-learning domain we regarded a total of 125 documents in the ML3 format. Table 11.1 shows the sizes of the XML documents. The large value of $|N|$ is caused by the large corpus with many documents. Since the document sizes vary widely, with some very small and some very large documents (cf. min and $max |N|$), the *mean deviation* is also very large. There is a large difference between the *average* and the *mean* values of $|T|$, because there is a small number of documents with many text nodes (as supported by the large value of max). The number of attributes per node is small, because often text nodes are used instead of attributes, while attributes mostly carry meta-information (such as IDs, target group, or difficulty). Many of these values are passed down to child elements, which therefore need fewer attributes themselves. The *mean* number of child nodes is dominated by the large number of leaf nodes which have no children, resulting in a mean value of zero.

Table 11.2 shows the sizes of the RDF implementations of the document models. The large *deviation* in the number of statements follows directly from the large *deviation* in the number of nodes in the source documents. Yet both the *sum* total and the *average* values of $|R|$ are smaller than the corresponding values of $|N|$, because not every XML element is represented as a document fragment. On the other hand, the *sum* and *average* values of $|L|$ are larger than the values of $|T|$: this is due to attributes that are represented as literals as well (in addition to text nodes). In addition, many literals are newly created for the document model and carry meta-information such as IDs, ordering information, or references to the source document. Note, however, that duplicate uses of a literal do not count towards $|L|$. The node degree is dominated by outgoing edges, which are mostly statements with a literal value. This is consistent with the *average* in- and out-degree, which is roughly in the same proportion as $|R|$ and $|L|$, namely about 1:3.

The processing times are shown in table 11.3. Since there is only one rule set to compile, the compilation times are all equal, with no deviation. A small number of very large documents exists, which causes a relatively large max value for the XML loading times.

The preprocessing step is complex because include-commands need to be resolved and inherited attributes need to be made explicit to make the subsequent processing easier. This results in a relatively long preprocessing time. The rule processing takes a similar amount of time. It is obviously the main part of obtaining a document model from a source document.

Yet all other processing times are dwarfed by the time required for creating the \mathcal{ALCCTL} view. This is caused by a complex document model (cf. table 11.2), on which $3 + 5 \cdot |S_{\mathcal{ALCCTL}}|$ (where $|S_{\mathcal{ALCCTL}}|$ is the number of states in the resulting \mathcal{ALCCTL} model) complex and recursive SPARQL queries need to be evaluated to build the \mathcal{ALCCTL} model. In contrast, only three non-recursive SPARQL queries are necessary for the background knowledge view. But a large and diverse \mathcal{ALCCTL} model is required to verify the large number of criteria for the ML3 documents.

An average of about six seconds (mean: five seconds) for a complete processing cycle for one document is adequate for live verification, especially considering that the rule compilation

	$ N $	$ T $	$avg(A)$	$mean(A)$	$avg(C)$	$mean(C)$
<i>sum</i>	93,695.00	41,695.00	125.32	93.00	124.30	0.00
<i>avg</i>	749.00	333.00	1.00	0.00	0.99	0.00
<i>mean</i>	241.00	29.00	0.88	1.00	1.00	0.00
<i>min</i>	41.00	3.00	0.32	0.00	0.98	0.00
<i>max</i>	7,841.00	4,809.00	2.79	4.00	1.00	0.00
<i>md.</i>	862.39	501.84	0.32	0.71	0.00	0.00
<i>msd.</i>	1,408.14	848.20	0.45	0.88	0.00	0.00

Table 11.1: E-learning documents: XML document sizes (across 125 documents).

	$ S $	$ R $	$ L $	$avg(D)$	$avg(I)$	$avg(O)$	$ F $
<i>sum</i>	349,617.00	57,296.00	157,316.00	963.06	232.42	730.64	54,687.00
<i>avg</i>	2,796.00	458.00	1,258.00	7.70	1.86	5.85	437.00
<i>mean</i>	2,359.00	418.00	1,102.00	7.61	1.90	5.70	395.00
<i>min</i>	307.00	68.00	155.00	5.96	1.44	4.51	50.00
<i>max</i>	10,507.00	1,370.00	4,056.00	10.58	1.97	8.61	1,349.00
<i>md.</i>	1,529.57	227.39	652.71	0.51	0.07	0.46	226.12
<i>msd.</i>	1,972.74	287.82	834.59	0.73	0.10	0.66	286.57

Table 11.2: E-learning documents: document model sizes (across 125 document models).

and the generation of the XML view is not necessary in every cycle, and the generation of the background knowledge view is mainly useful when the document in question is *not* otherwise part of the verification process (cf. section 6.2). This reduces the total average for a processing cycle to five seconds (mean: three seconds).

11.2.2 Technical Documentation

In the technical documentation domain we regarded seven documents in DocBook format. Table 11.4 shows the sizes of the XML documents. Note that we removed from each of the source files the document type definition (DTD) declaration that referred to an external DTD available online. This allows for a better comparison of the XML load times. Since this corpus is smaller than the e-learning corpus, the total *sum* of $|N|$ is smaller as well. Yet the *average* and *mean* value of $|N|$ are both considerably larger, because most documents are considerably larger as well. The ratio of XML nodes versus text nodes is very balanced, in contrast to the ML3 documents, where $|N| > |T|$. The reason is that the document structure here is far less complex, with more actual content per structural element. At the same time, the *average* and *mean* values for the number of attributes are lower than in the ML3 documents, because there are fewer

	R.Comp.	L.XML	Prep.	R.Exec.	V:XML	V:ALCCTL	V:BGK	Verif.	Total
<i>sum</i>	1.404	0.484	10.609	11.668	3.304	569.184	3.737	7.006	607.397
<i>avg</i>	1.404	0.004	0.085	0.093	0.026	4.553	0.030	0.056	6.252
<i>mean</i>	1.404	0.001	0.048	0.080	0.017	3.039	0.027	0.022	4.638
<i>min</i>	1.404	0.001	0.006	0.029	0.003	0.151	0.023	0.001	1.617
<i>max</i>	1.404	0.084	0.853	0.306	0.204	27.563	0.067	0.563	31.045
<i>md.</i>	0.000	0.004	0.069	0.040	0.019	3.631	0.005	0.056	3.825
<i>msd.</i>	0.000	0.012	0.105	0.054	0.031	5.153	0.008	0.092	5.456

Table 11.3: E-learning documents: processing times in seconds (for 125 documents/document models).

	$ N $	$ T $	$avg(A)$	$mean(A)$	$avg(C)$	$mean(C)$
<i>sum</i>	31,016.00	32,923.00	1.06	0.00	6.99	0.00
<i>avg</i>	4,430.00	4,703.00	0.15	0.00	1.00	0.00
<i>mean</i>	3,604.00	3,040.00	0.15	0.00	1.00	0.00
<i>min</i>	220.00	186.00	0.13	0.00	1.00	0.00
<i>max</i>	10,596.00	13,580.00	0.20	0.00	1.00	0.00
<i>md.</i>	3,473.84	4,064.61	0.02	0.00	0.00	0.00
<i>msd.</i>	3,971.45	4,666.07	0.02	0.00	0.00	0.00

Table 11.4: Technical documentation documents: XML document sizes (across seven documents).

	$ S $	$ R $	$ L $	$avg(D)$	$avg(I)$	$avg(O)$	$ F $
<i>sum</i>	37,904.00	8,558.00	11,161.00	43.37	13.82	29.55	8,374.00
<i>avg</i>	5,414.00	1,222.00	1,594.00	6.20	1.97	4.22	1,196.00
<i>mean</i>	4,227.00	1,005.00	1,232.00	6.20	1.99	4.21	985.00
<i>min</i>	410.00	110.00	141.00	5.50	1.75	3.73	96.00
<i>max</i>	13,266.00	3,039.00	3,540.00	6.84	2.21	4.86	2,954.00
<i>md.</i>	4,213.27	920.08	1,227.92	0.36	0.12	0.25	909.18
<i>msd.</i>	4,641.94	1,028.81	1,341.01	0.45	0.15	0.34	1,011.56

Table 11.5: Technical documentation: document model sizes (across seven document models).

metadata than in the e-learning documents (e.g., no target group or difficulty information).

Table 11.5 shows the sizes of the RDF implementations of the document models. The *average* proportion of $|R|$ to $|N|$ is similar to the e-learning domain: the document structure is less complex (as noted above), therefore there are fewer document fragments in total, but there is a similar number of document fragments per XML node (as supported by the respective values for $|F|$). On the other hand, the proportion of the *average* values of $|R|$ to $|L|$ is greater than in ML3: as noted above, the technical documentations contain fewer metadata, which requires fewer literals to represent them. The *average* node-degree is slightly lower when compared with the ML3 documents. This is caused by the out-degree, because while the in-degree is similar to above, the out-degree is lower. This, in turn, is caused by the lower number of literals per RDF node.

The processing times are shown in table 11.6. Again, the R.Comp values are all equal: as with ML3, there is only one rule set to compile. The rule execution times are even more prominent than before, because the rule processing is the main part of obtaining a document model from a source document. The average rule execution time is larger than above, reflecting the larger average document size. But the sum total of the execution time is also higher than for the e-learning documents or for the NPB corpus (see below), even though the document corpus itself is smaller. This is caused by the more complex XPath expressions used in the background knowledge. Replacing them with less complex (albeit less effective) expressions significantly reduces the processing time. The time requirements for creating the \mathcal{ALCCTL} views is considerably smaller than for the ML3 documents, because there are fewer verification criteria, and therefore a far simpler \mathcal{ALCCTL} model ($3+3 \cdot |S_{\mathcal{ALCCTL}}|$ SPARQL queries are necessary here). The average total processing time minus rule compilation of less than half a minute (10 seconds mean) is still (barely) adequate for live processing.

11.2.3 Process Descriptions (NPB)

In the process description domain we regarded four documents obtained from the National Process Library (NPB) in three different formats (BPMN, Visio, and a proprietary XML format).

	R.Comp.	L.XML	R.Exec.	V:XML	V:ALCCTL	V:BGK	Verif.	Total
<i>sum</i>	0.330	1.475	180.353	0.868	2.640	0.240	0.003	185.909
<i>avg</i>	0.330	0.211	25.765	0.124	0.377	0.034	0.000	26.841
<i>mean</i>	0.330	0.078	10.170	0.066	0.248	0.034	0.000	10.926
<i>min</i>	0.330	0.052	0.219	0.005	0.032	0.028	0.000	0.666
<i>max</i>	0.330	0.633	83.738	0.422	0.923	0.043	0.001	86.090
<i>md.</i>	0.000	0.169	26.646	0.110	0.319	0.004	0.000	27.247
<i>msd.</i>	0.000	0.200	31.005	0.137	0.364	0.005	0.000	31.712

Table 11.6: Technical documentation documents: processing times in seconds (for seven documents/document models).

	$ N $	$ T $	$avg(A)$	$mean(A)$	$avg(C)$	$mean(C)$
<i>sum</i>	36,365.00	19,402.00	4.85	4.00	3.99	1.00
<i>avg</i>	9,091.00	4,850.00	1.21	1.00	1.00	0.00
<i>mean</i>	15,186.00	9,054.00	2.25	2.00	1.00	0.00
<i>min</i>	199.00	0.00	0.15	0.00	0.99	0.00
<i>max</i>	16,234.00	10,160.00	2.25	2.00	1.00	1.00
<i>md.</i>	6,618.75	4,756.50	1.03	1.00	0.00	0.38
<i>msd.</i>	6,821.26	4,773.01	1.03	1.00	0.00	0.43

Table 11.7: Process description documents (NPB): XML document sizes (across four documents).

The Visio document is depicted in figure B.1 in appendix B.3. Table 11.7 shows the sizes of the XML documents. The average value of $|N|$ is very large, which is caused by the Visio document that contains a lot of graphical data in addition to the pure process description, and by the documents in proprietary XML format that are very verbose. The document in BPMN format is considerably smaller, as indicated by the *min* and *mean deviation* values. The proportion of $|N|$ to $|T|$ is similar to that of the ML3 documents, which have a similarly complex structure.

Table 11.8 shows the sizes of the RDF implementations of the document models. The *average* node degrees (D , O , and I) are similar to the technical documentations, as is the lower ratio of $|L|$ to $|R|$. However, the *average* values of $|S|$, $|R|$, and $|L|$ are far smaller than in either the ML3 documents or the technical documentations: the structure of the described processes is considerably simpler than the structure of the documents. This can also be seen in the relatively low *average* number of process fragments.

The processing times are shown in table 11.9. Even though there are three different rule sets for the documents now, they are all very similar with only a few deviations from the default rule set, resulting in a low *mean deviation* for R.Comp. In contrast, the *mean deviation* for the rule execution is relatively large. This is caused by the widely diverging document sizes (as evidenced by the *mean deviations* of $|N|$), upon which the rule execution time largely depends.

11.2.4 Process Descriptions (DLR)

In the process description domain we regarded another 20 documents obtained from the German Aerospace Center (DLR) in Word format. Table 11.10 shows the sizes of the XML documents. Each document follows a similar standardised pattern, thus the deviations w.r.t. the document size are small. All text is contained in attributes, as reflected by the *average* number of attributes and the absence of any text nodes.

Table 11.11 shows the sizes of the RDF implementations of the document models. Since there is only a single process model, the *mean deviation* on $|S|$ is zero. Both $|S|$ and $|L|$ are relatively large, while $|R|$ is small, because there are many attributes (such as topics, descriptions, or

	S	R	L	avg(D)	avg(I)	avg(O)	F
<i>sum</i>	631.00	203.00	322.00	15.85	4.13	11.72	95.00
<i>avg</i>	157.00	50.00	80.00	3.96	1.03	2.93	23.00
<i>mean</i>	147.00	62.00	74.00	5.05	1.50	3.55	21.00
<i>min</i>	30.00	29.00	25.00	1.17	0.14	1.03	3.00
<i>max</i>	319.00	74.00	150.00	7.10	1.95	5.15	51.00
<i>md.</i>	80.63	17.25	34.75	2.11	0.69	1.42	13.63
<i>msd.</i>	103.63	18.05	44.75	2.28	0.73	1.56	17.28

Table 11.8: Process description documents (NPB): document model sizes (across four document models).

	R.Comp.	L.XML	R.Exec.	V:XML	Total
<i>sum</i>	0.973	0.517	1.958	0.014	3.462
<i>avg</i>	0.324	0.129	0.490	0.003	0.946
<i>mean</i>	0.357	0.071	0.484	0.002	0.915
<i>min</i>	0.256	0.013	0.154	0.001	0.424
<i>max</i>	0.359	0.387	0.923	0.009	1.679
<i>md.</i>	0.045	0.129	0.217	0.003	0.394
<i>msd.</i>	0.048	0.150	0.278	0.003	0.480

Table 11.9: Process description documents (NPB): processing times in seconds (for four documents/document models).

associated persons) per fragment, also shown in the high *average* out-degree. The (average) $|F|$ is larger than for the NPB models: while there is only a single process model here, it is also more complex than the other process models.

The processing times are shown in table 11.12. The preprocessing only combines the separate Word documents into a single file and is therefore rather fast and only executed once for all documents (hence the equality of the sum and average values). The rule execution time is very large because of two contributing factors: first, the large number of XML nodes that are all processed at once, and second the complex XPath expressions in the background knowledge. To a lesser degree, we already observed this effect earlier. The time required for the \mathcal{ALCCTL} view is small because of the small process model (compared to the document models above) and because the SPARQL queries required are all very simple, non-recursive queries (even though $3 + 19 \cdot |S_{\mathcal{ALCCTL}}|$ queries are necessary).

The total processing time is longer than two minutes, and thus unsuitable for live processing. It is, however, possible to employ the entire verification process as a recurring background task.

	N	T	avg(A)	mean(A)	avg(C)	mean(C)
<i>sum</i>	6,791.00	0.00	58.86	42.00	19.94	0.00
<i>avg</i>	339.00	0.00	2.94	2.00	1.00	0.00
<i>mean</i>	322.00	0.00	2.99	2.00	1.00	0.00
<i>min</i>	176.00	0.00	2.60	2.00	0.99	0.00
<i>max</i>	682.00	0.00	3.22	3.00	1.00	0.00
<i>md.</i>	82.72	0.00	0.13	0.18	0.00	0.00
<i>msd.</i>	115.64	0.00	0.17	0.30	0.00	0.00

Table 11.10: Process description documents (DLR): XML document sizes (across 20 documents).

	S	R	L	avg(D)	avg(I)	avg(O)	F
<i>sum</i>	1,613.00	126.00	1,010.00	14.33	1.52	12.80	97.00
<i>avg</i>	1,613.00	126.00	1,010.00	14.33	1.52	12.80	97.00
<i>mean</i>	1,613.00	126.00	1,010.00	14.33	1.52	12.80	97.00
<i>min</i>	1,613.00	126.00	1,010.00	14.33	1.52	12.80	97.00
<i>max</i>	1,613.00	126.00	1,010.00	14.33	1.52	12.80	97.00
<i>md.</i>	0.00	0.00	0.00	0.00	0.00	0.00	0.00
<i>msd.</i>	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Table 11.11: Process description documents (DLR): document model sizes (for one document model).

	R.Comp.	L.XML	Prep.	R.Exec.	V:XML	V:ALCCTL	V:BGK	Verif.	Total
<i>sum</i>	0.302	0.343	0.031	240.675	0.022	0.326	0.085	0.108	241.892
<i>avg</i>	0.302	0.017	0.031	240.675	0.022	0.326	0.085	0.008	241.465
<i>mean</i>	0.302	0.015	0.031	240.675	0.022	0.326	0.085	0.005	241.460
<i>min</i>	0.302	0.008	0.031	240.675	0.022	0.326	0.085	0.003	241.452
<i>max</i>	0.302	0.059	0.031	240.675	0.022	0.326	0.085	0.040	241.540
<i>md.</i>	0.000	0.007	0.000	0.000	0.000	0.000	0.000	0.006	0.013
<i>msd.</i>	0.000	0.011	0.000	0.000	0.000	0.000	0.000	0.009	0.021

Table 11.12: Process description documents (DLR): processing times in seconds (for 20 documents/one document model).

Conclusion

Figure 11.4 shows an overview of the total processing times for documents from the four domains. Unsurprisingly, the high similarity of the rule sets leads to very similar rule compilation times for all domains. Loading times for the XML files are generally low across the domains. Only the ML3 and the DLR documents require preprocessing steps, where the one for ML3 is considerably more complex because it must aggregate and pass down various attributes, as well as create different branches for different content intensity levels. The rule execution times vary widely, based on the different numbers of documents, different document sizes, and different complexity of the XPath expressions in the background knowledge. The time for creating the XML views is dominated by the size and number of the documents, while the time for creating the *ALCCTL* views is mostly determined by the number and complexity of the SPARQL queries that define the views. Creating the background knowledge view takes time proportional to the total document size. The time for the verification depends mostly on the number of documents and on the number of formulae.

The average processing times for document from the four domains are presented in figure 11.5. The average compilation time for the NPB rules is slightly lower than for the other rules, even though their general structure is the same. The loading times for ML3 files are generally lower than for the other domains, because the average document size there is considerably smaller. For the average preprocessing time, this smaller size is counteracted by the more complex preprocessing procedure, resulting in a similar average as for the DLR documents. As noted and explained above, rule execution times for the technical documentations and, even more so, for the DLR documents are rather high. When creating the XML views, the smaller document size of the ML3 documents compared to the NPB documents is offset by the far less complex structure of the NPB models. As stated above, times for creating the *ALCCTL* views are dependent on their specification, in particular on the complexity of the SPARQL queries involved.

The processing time is generally dominated by the number of objects that are processed

and by the complexity of the XPath expressions used in the background knowledge. Therefore, assuming fixed document sizes, the best way to minimise processing times is to use XPath expressions that are as simple as possible, or to use a more efficient XPath evaluator. We used the Xerces XPath evaluator, which to the best of our knowledge is one of the fastest evaluators available for Java.

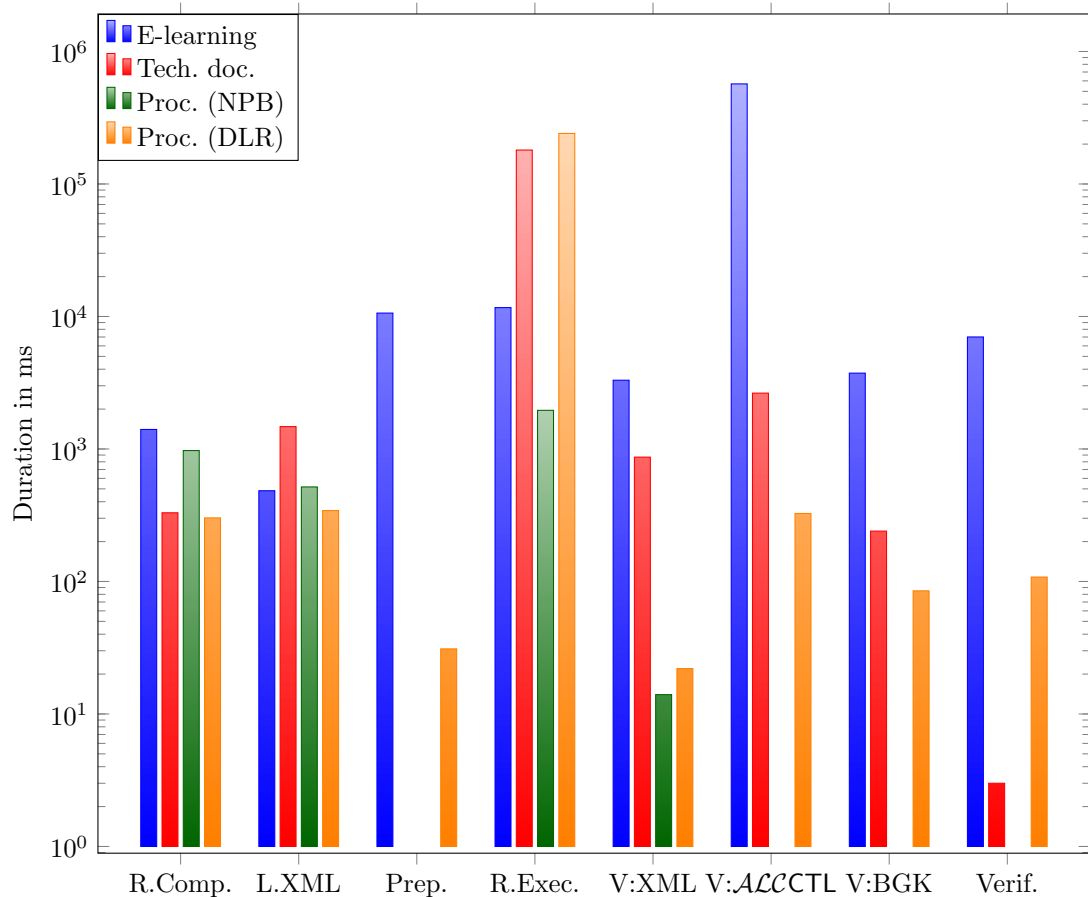


Figure 11.4: Time measurements for the *total* processing time of documents from different domains

To summarise, we have shown that it is possible to create and use semantic document models in a reasonable amount of time for real documents in different domains and formats and even for process descriptions.

11.3 Comparison with Alternative Methods

We will now compare the efficiency of our implementation with alternative methods. In particular, we will compare the Drools implementation of the transformation rules with an XQuery implementation, and the SPARQL implementation of the view instantiation with an XQuery

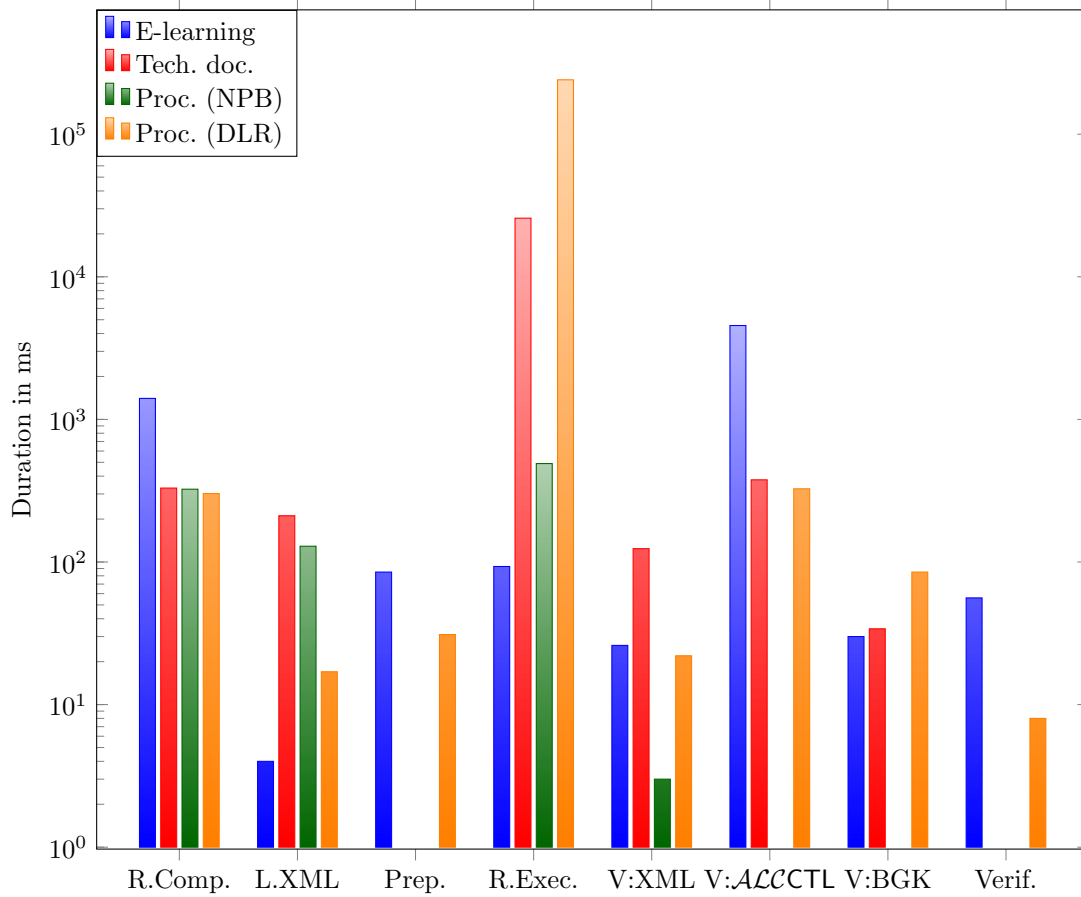


Figure 11.5: Time measurements for the *average* processing time of documents from different domains

implementation as well. Note that for the latter, we transform the RDF/OWL-based implementation of the semantic document model into a suitable XML representation first. This evaluation complements the discussion of alternate methods from chapter 9. Rule-based inference as an alternative to inference by an OWL reasoner for processing document models will be discussed in section 11.5.

For the transformation rules, we expect the XQuery implementation to be slightly more efficient than the Drools implementation, because the RETE algorithm, on which Drools is based, must check all objects against all rules, while an XQuery program can move more purposefully through the data by filtering the objects that must be processed. This supposition is confirmed by the data, as shown in figures 11.6 and 11.7. The first figure shows the results for a set of generated HTML documents scaled by document size (between 100 and 1,000 files per document, shown on the left hand side) and by number of references (averaging between 4 and 64 references per file with 500 files per documents, shown on the right hand side). It can be seen that the processing times scale similarly for both methods, with XQuery in the lead performance-wise with a constant factor of about 2.5. The second figure shows the results for a set of DocBook documents. Here, the relative performance results differ widely, with factors between 5 and 250 depending on the complexity of the source document, but the greater efficiency of the XQuery implementation is evident.

However, this advantage is bought at the cost of flexibility: XQuery is harder to write domain-independently, and it is harder to integrate background knowledge. For applications where this trade-off is worthwhile, we recommend using a specialised implementation in either XQuery or Java (which, in our experiments, performed even better than XQuery), instead of relying on the more general, yet less efficient, Drools-based approach. Yet many things that are easy to achieve in Drools or directly in Java are hard to do in XQuery because the language simply is not meant for them, such as removing duplicates from a list, formatting string values, or prompting the user for confirmation.

For the view instantiation, we expect a slightly different outcome. XQuery even without recursion is NExpTime-hard, while SPARQL (which does not support recursion) is only PSpace-complete. We therefore expect the SPARQL implementation to handily beat the XQuery program. Figure 11.8 shows the processing times for document models based on the set of generated HTML documents (see above). Contrary to our initial assumption, it can be seen that for the large, but very simple (i.e., with a very shallow XML hierarchy) HTML documents, the SPARQL implementation cannot outperform the XQuery implementation – the performance is very similar, with XQuery actually a bit faster in most cases. For the more complex DocBook documents however, this picture changes (cf. figure 11.9). With two exceptions, SPARQL now actually performs somewhat better than XQuery. For the very complex PyGTKTutorial document model, SPARQL even performs significantly better than the XQuery implementation. We conclude that for simple document models, the theoretical complexity of XQuery has little effect because no complex path expressions or similar operations are required. Additional experiments with e-learning documents in ML3 format confirm these observations.

An interesting future experiment is the use of a graph database such as Neo4j as an alternative to the Jena-based RDF/OWL implementation of document models, including Jena's SPARQL implementation. However, most graph databases are optimised for typical graph-related tasks like shortest path or minimum spanning tree, but not for the subgraph matching task required for SPARQL evaluation. This is evident in the operations that are supported by many of these databases – or, more to the point, in the operations that are *not* supported. Therefore, considerable development effort is required to implement our document model framework on top of such a graph database.

There exists however a bridge between the Sesame RDF framework and the Neo4j graph data-

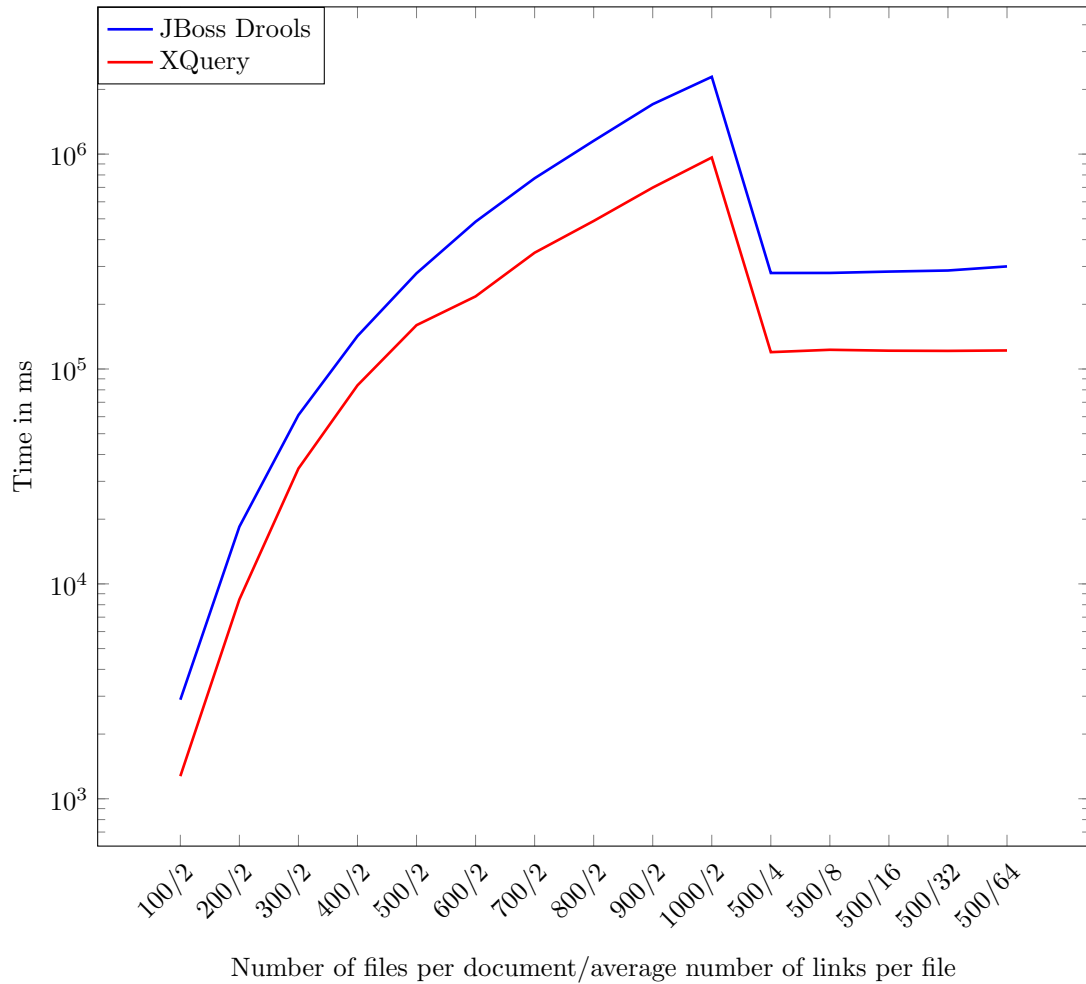


Figure 11.6: Comparison of processing times for JBoss Drools and XQuery: obtaining a semantic document model from an HTML document

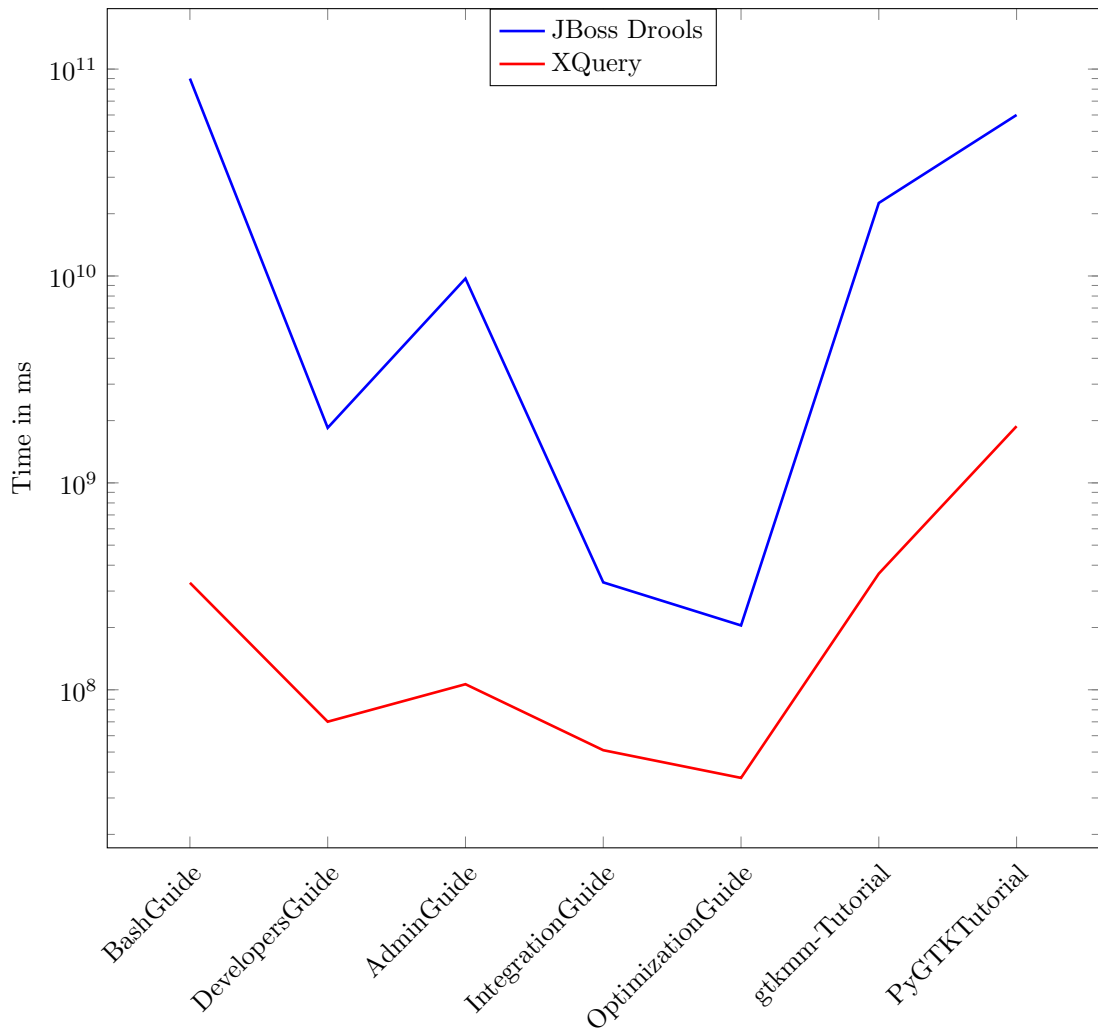


Figure 11.7: Comparison of processing times for JBoss Drools and XQuery: obtaining a semantic document model from a DocBook document

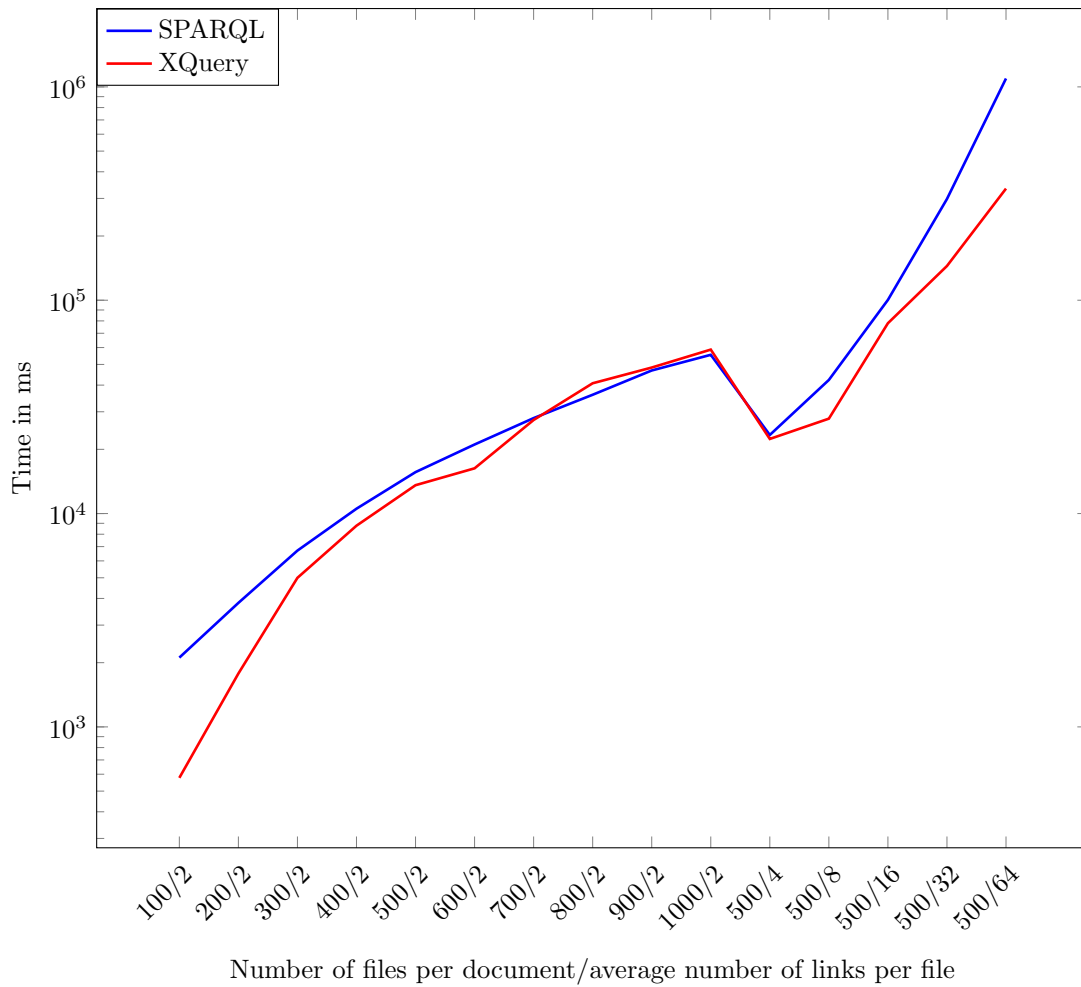


Figure 11.8: Comparison of processing times for SPARQL and XQuery: obtaining a verification model from a semantic document model (HTML)

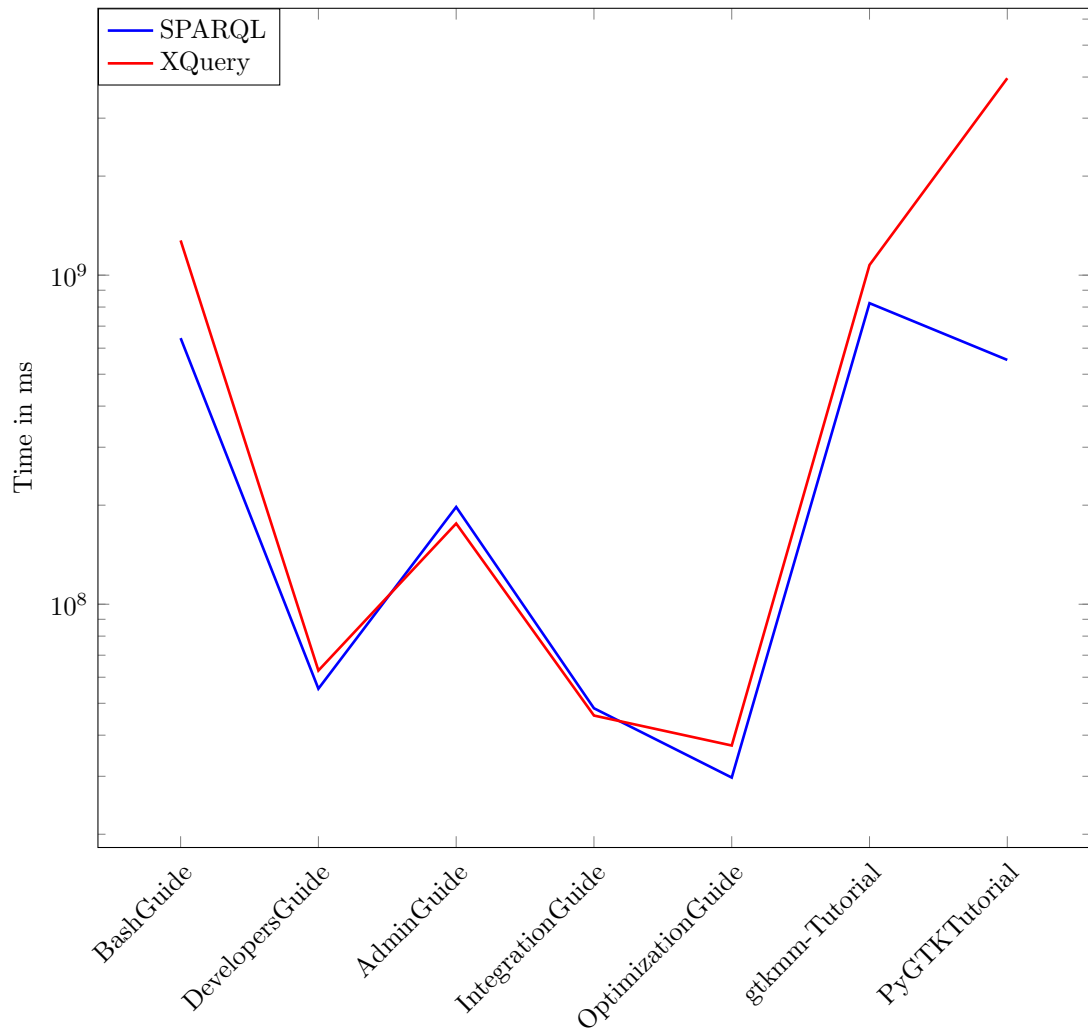


Figure 11.9: Comparison of processing times for SPARQL and XQuery: obtaining a verification model from a semantic document model (DocBook)

	DocBook	BPMN	Visio	Custom XML	DLR
Mandatory adaptations	2	2	2	2	2
Simplifications/removals	4	3	4	5	5
Changes/additions	2	2	2	1	4
Extensions/additions	0	0	1	0	0
Weighted sum	10	9	15	9	15

Table 11.13: Adaptations to the processing rules for different domains.

base, which allows SPARQL queries to be executed on the database. It would be an interesting future project to find out if this configuration can be used to further enhance the performance of our framework.

11.4 Adaptability of the Processing Rules

We will now attempt to quantify and assess the adaptability of the processing rules, i.e., how many changes need to be made to the default rule set before it can be used in a new domain. The rule set for the ML3 documents was created in [Stel10]. It is not yet based on the default rule base, and will therefore be ignored in this comparison.

Table 11.13 shows a listing of the various types of changes that were made to the default rule set. The **mandatory adaptations** are those that are defined in section 9.1.3, namely the two functions `getChildElements(LocalContext): List` and `getIndicators(LocalContext)`. These adaptations are made for every domain.

Simplifications/removals are changes where an existing rule is either removed entirely, or some part of it is deleted because the new domain or the new document format only requires less complex processing. For example, if there are no keywords in the background knowledge for a domain, then the corresponding part of the processing rules can be removed as well.

Changes/additions are small adaptations of the rules where simple things are changed or added, usually in a single line of code. For example, the text that is checked for keywords may not only be retrieved from XML text nodes, but also from specific attributes.

Extensions/additions are more complex adaptations, like a completely new rule or comprehensive changes to an existing rule. Such an extensive change was necessary for processing the references (i.e., the control flow) in the Visio process description: references are represented as graphical arrows between other graphical components, which in turn represent process steps. So the control flow had to be established by finding the graphical components at the start and at the end of an arrow, and then by connecting the corresponding process steps.

The weights for the weighted sum (bottom line) were chosen so that the resulting number roughly corresponds to the number of work-minutes required to make the adaptations (based on our own implementation times). The first two rows (mandatory adaptations and simplifications/removals) are weighted as 1, the third row (changes/additions) is weighted as 2, and the fourth row is weighted as 5.

Note how the bulk of the adaptations are removals, which are only made for reasons of efficiency. In fact, the rules could be used just as effectively – but not as efficiently – without making these removals. Combined with the mandatory adaptations, the simplifications pose over 70 % of the number of rule changes, and over 50 % of the total time required to make the changes.

This shows how little actual work has to be put into changing the processing rules, with a maximum of about 15 minutes for a completely new domain. These numbers are based on the assumption, however, that the domain and the new document format are already known to the

person making the adaptations, i.e., that the changes are made by domain experts.

11.5 Inference on Large Ontologies

We will now have a look at different inference engines that are publicly available and evaluate their suitability for inference tasks on large ontologies, such as large document models or large background knowledge bases. The Hermit (version 1.3.5) and Fact++ (version 1.5.2) reasoners have both proven to be inadequate, since they both produced results that while sound were incomplete. The reasoner integrated into the Jena framework (version 2.6.2) produced complete results, but missed a (deliberate) inconsistency in the data. Additionally, the Jena reasoner performs very poorly, requiring far more time than any other reasoner we regarded.

The Pellet reasoner (version 2.3) has shown itself to be both efficient and adequate (in the sense of “sound and complete”). We will test it against a set of inference rules that cover concept and role generalisation. The rules are executed by the JBoss Drools rule engine. Note that these rules only represent a fraction of the description logics semantics supported by Pellet. On the other hand, Pellet is a dedicated OWL/DL reasoner, while Drools is a general purpose rule engine.

We start with a basic schema of 10 concepts in a linear generalisation relationships, i.e., the first concept is a sub-concept of the second, the second concept is a sub-concept of the third, and so forth. First, we test how the inference times scale in the number of individuals. To this end, we created five test sets with 20, 40, 60, . . . , 180 or 200 individuals asserted for each concept, respectively. The results are shown in figure 11.10.

At first, the Pellet reasoner clearly outperforms the Drools engine. Both time requirements rise sharply, which is unsurprising because description logics reasoning is NP-hard. It is noteworthy, however, that the custom inference rules slowly close the performance gap and finally overtake the Pellet reasoner. For 10,000 individuals (1,000 per concept, not shown in the diagram), the Pellet reasoner requires more than twice as long as the inference rules, with five minutes versus 2 minutes, respectively. This comes, however, at the cost of fewer inferred facts and no consistency checking.

Our next test set asserts a fixed number of 10 individuals per concept, but scales the number of concepts: for each of the 10 original concepts, there are 20, 40, 60, . . . , 180 or 200 super-concepts introduced into the ontology, respectively. Figure 11.11 shows the measured times. This time the inference rules, which focus on just the concept generalisation that is scaled in this scenario, are completely outclassed by Pellet.

It can also be seen that Pellet performs better for a large number of concepts than for a large number of individuals. This is to be expected since the number of inference tasks as dictated by the full description logics semantics for individuals and roles is much higher than the number of tasks for concepts alone.

The results show that executing a specific set of inference rules with drools is well suited for simple or very specific inference tasks such as those from section 9.2, but Pellet should be used in all other cases when actual description logics semantics are required.

In-memory reasoning works well for relatively small ontologies. When dealing with huge knowledge bases such as very large document corpora or ontologies obtained from Wikipedia, however, such techniques reach their limits. We will now investigate how a vertical-cut technique that keeps the TBox in-memory, but moves the bulk of the knowledge – the ABox – into a conventional relational database, compares to a pure in-memory technique and a technique that works entirely on the database. For the hybrid vertical-cut technique (which we will now call

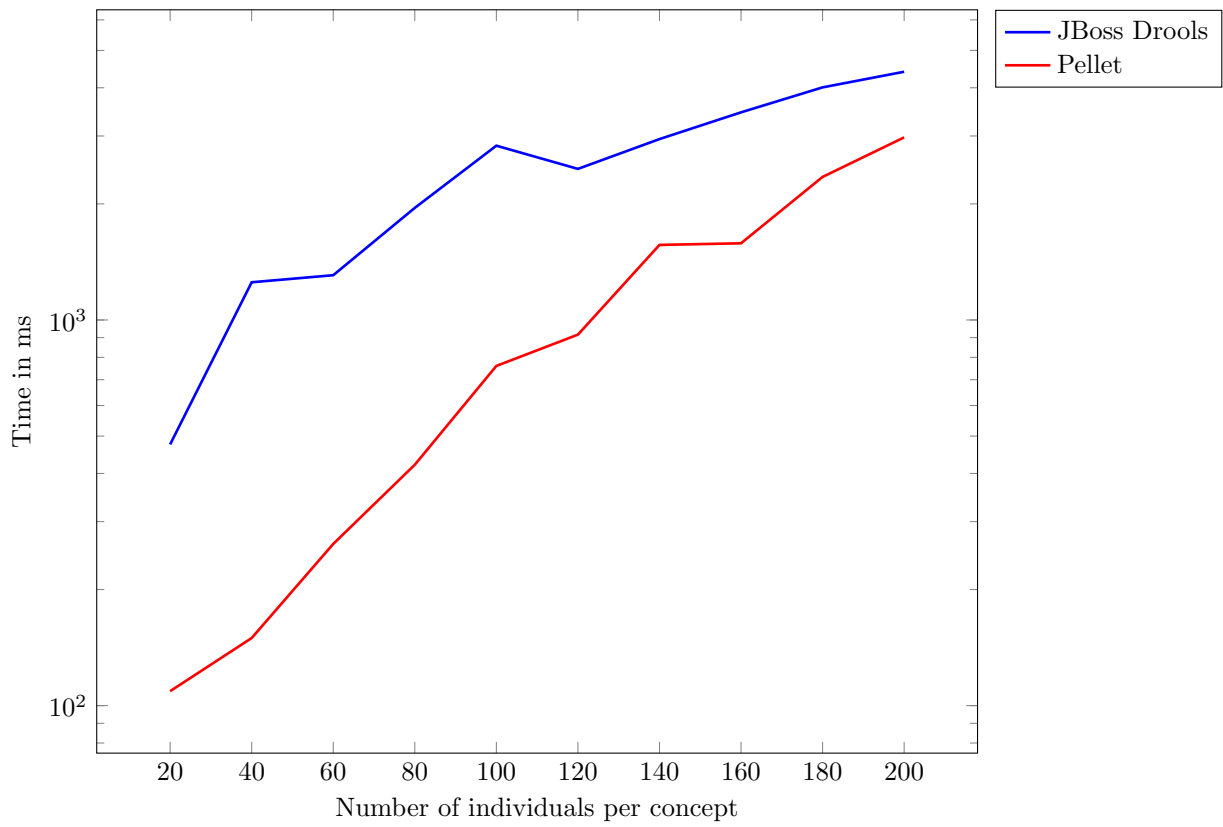


Figure 11.10: Ontology inference scaled by number of individuals

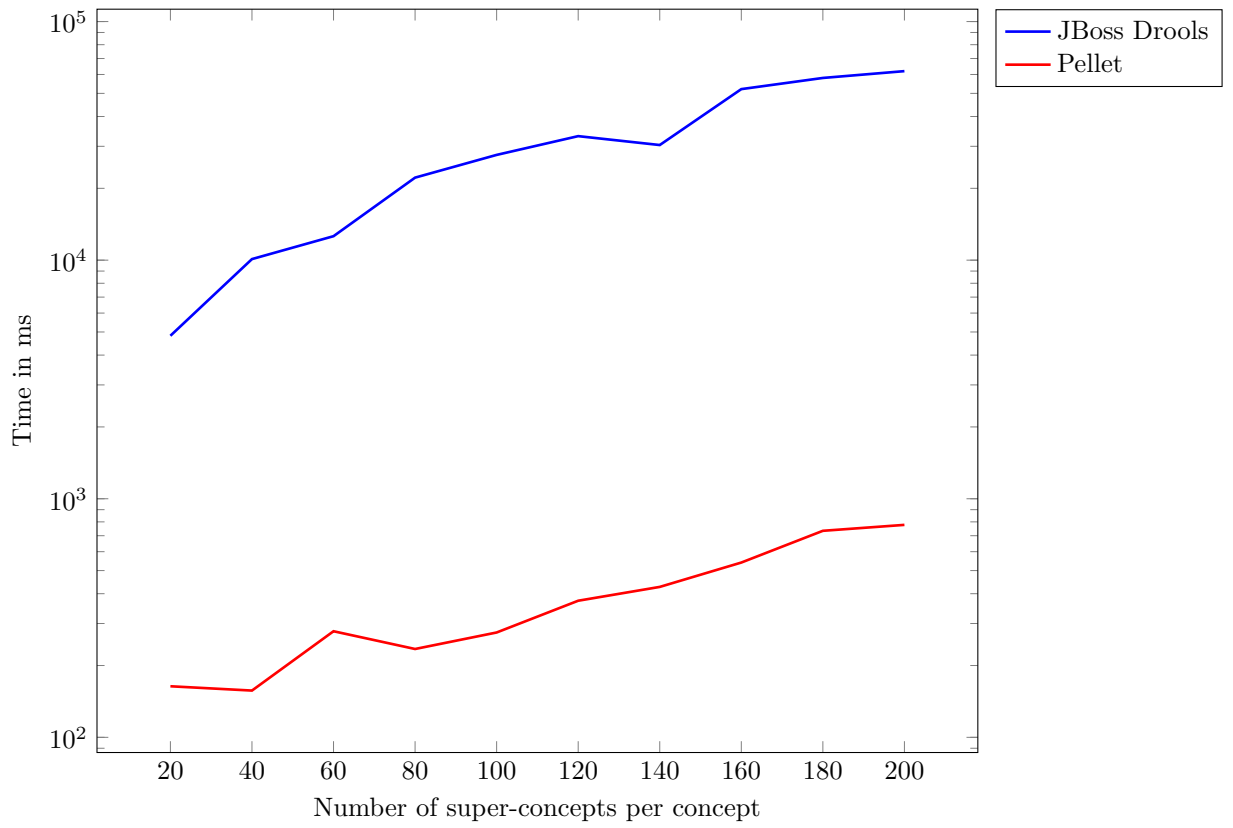


Figure 11.11: Ontology inference scaled by number of concepts

“Mem./DB”), we used two Jena ontology models: one for the TBox, and one for the ABox. The former is a memory-model, while the latter is a database-backed model. For the in-memory technique (which we will now call “Memory”), a single memory-model is used. And for the database technique (which we will now call “Database”), a single database-backed model is used.

We investigate the three techniques using two sets of generated ontologies that scale in the number of individuals and in the number of concepts, respectively. When the number of concepts is scaled, each concept has a constant average of two concept assertions for individuals and two sub-concept axioms. When the average number of individuals per concept is scaled, there is a fixed number of 128 concepts with a constant average of two sub-concept axioms. Sub-concept axioms are created randomly, but in a way that resembles many typical ontologies: a hierarchy of concepts is created, where most axioms are asserted between one layer and the next, but some axioms ignore the hierarchy and connect arbitrary concepts. The Wikipedia category hierarchy is structured in a similar manner, as are most ontologies that we used as background knowledge.

The times for reading the ontology files serialised in Turtle and loading them into the different Jena models follow our expectations, with the Database approach the slowest (between one and 1,000 seconds, depending on the size of the ontology) and the Memory approach the fastest (between 10 and 1,000 milliseconds). The Mem./DB approach falls in between the other two (between 100 milliseconds and 500 seconds).

The times required for concept and individual retrieval are more interesting. For each ontology, we first obtained a list of all concepts, and then for each concept, we obtained the list of individuals belonging to the concept. The time requirements can be seen in figures 11.12 and 11.13.

The Database approach requires the most time when scaling the number of concepts, while Memory and Mem./DB are almost en par. The Memory technique needs more time to find classes because its single Jena model contains more statements than the small Mem./DB TBox. But the Mem./DB technique needs more time to retrieve the individuals from the database, despite its index support. However, further tests show that for even larger numbers of concepts, the Mem./DB approach actually becomes more efficient than the Memory approach.

The most obvious effect when scaling the number of individuals is how badly the Memory technique scales. Its huge time requirements are caused by its need to search through a huge amount of data, as all statements of the very large ontology are lumped together in a single Jena model. Since Jena does not use index structures for memory models, this is time consuming even in-memory. The hybrid approach has the double advantage of a small TBox ontology and index-supported ABox retrieval, clearly beating both other strategies in this retrieval scenario. Note that this advantage vanishes for other retrieval tasks that require many joins on the ABox data. It is interesting to note that the index-support of the database eventually beats the in-memory advantage, as evidenced by the largest ontology.

While there are several possible optimisation for the Memory approach, such as splitting the ontology into multiple Jena models (which, however, comes with its own set of caveats because it necessitates joins) or introducing index structures, our data has clearly shown that for large ontologies a vertical cut through a knowledge base is a promising approach. It can effectively combine the speed advantages of in-memory processing with the storage and retrieval advantages of conventional relational databases.

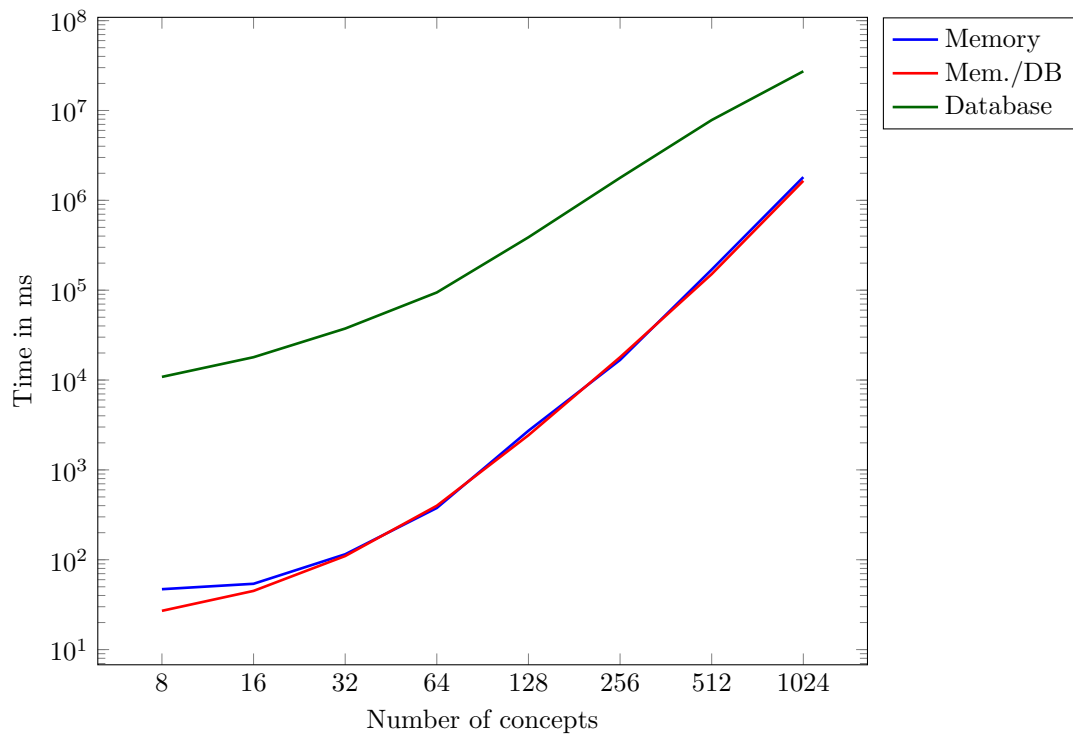


Figure 11.12: Huge ontology retrieval times, scaled by number of concepts.

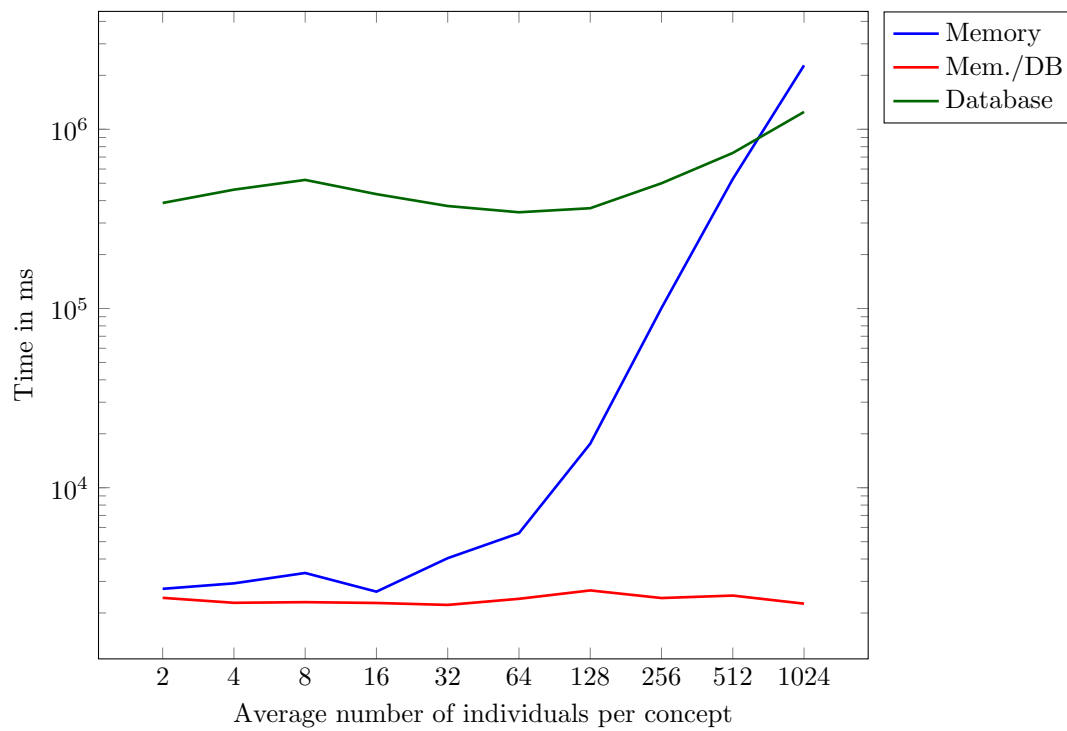


Figure 11.13: Huge ontology retrieval times, scaled by number of individuals per concept

Conclusion

In this chapter we have shown that our proposed approach works in a reasonable amount of time on very different documents. We have also shown that the processing rules can indeed be easily adapted, without the necessity of huge changes when applying them to a new domain or a new document format. Finally, we have presented different options for calculating inference tasks on large ontologies.

Chapter 12

Qualitative Evaluation

In this chapter, we will investigate how the quality of semantic models of documents can be measured, and how the quality of background knowledge can be measured. We will then evaluate the relationship between the respective qualities of the two. Finally, we will discuss the effectiveness of our approach for various types of documents. First, however, we ask ourselves how expressive the rules used for the model generation are.

12.1 Expressive Power of Transformation Rules and Background Knowledge

The premise of a transformation rule (cf. section 5.1), similar to the rule head in many rule languages including JBoss Drools, can only access local information, i.e., information available in the media object that the rule is matched to. For example, a premise cannot (directly) contain conditions about a successor or a predecessor of a media object. There are, however, ways to resolve this. One option is to preprocess the base document model and make the relevant information about successors of predecessors available in each media object. Another option is the use of state variables that are set in other rules, when the succeeding or preceding media objects are processed. Finally, one might state these conditions using path expressions (such as XPath expressions). While neither transformation rules nor Drools rules formally support path expressions in the premise, at least for Drools it is possible to use them through some programming trickery¹. Yet, these options make rules both harder to write and harder to understand, so we recommend avoiding it whenever possible.

In general, [SSD09] has shown for Constraint Handling Rules (CHR) that they are Turing-complete. The Drools rule language is a super-set of CHR and is thus also Turing-complete. The question remains, however, if transformation rules as defined in section 5.1 are Turing-complete as well, or if they are missing some necessary expressive power.

Proposition 12.1.1 (Expressive Power of Transformation Rules). *Transformation rules as defined in section 5.1 are Turing-complete.*

Proof of Proposition 12.1.1.

¹Expressions like `field["name"]` in a Drools rule head are interpreted as a call to a Java `Map`. By implementing the `Map` interface and overriding the `getValue(String name)` method, the `name` can be interpreted as an XPath expression and evaluated against some XML document.

We will show the Turing-completeness of transformation rules by implementing Conway's *Game of Life* [Gar70], which is known to be Turing-complete [Ren]. The following set of five rules emulates the Game of Life:

```

1 ( c(m) '=' 'init' )
2 ↪
3 ( EG.cells = new boolean[] [] )
4
5 ( md(m, "neighbours") '⊆' {0,1} )
6 ↪
7 ( EG.cells[md(m, "x")][md(m, "y")] = false )
8
9 ( md(m, "neighbours") '⊆' {2,3} )
10 ↪
11 ( EG.cells[md(m, "x")][md(m, "y")] = true )
12
13 ( md(m, "neighbours") '⊆' {4,5,6,7,8} )
14 ↪
15 ( EG.cells[md(m, "x")][md(m, "y")] = false )
16
17 ( c(m) '=' 'generation' )
18 ↪
19 (
20     display(EG.cells),
21     MediaObject o = new MediaObject(),
22     o.setText('generation'),
23     append o to the base document model,
24     for each (x,y) in EG.cells
25         int n = count living neighbours of EG.cells[x][y],
26         if (EG.cells[x][y] == true or n == 3)
27             o = new MediaObject(),
28             o.setMetadata('neighbours', n),
29             o.setMetadata('x', x),
30             o.setMetadata('y', y),
31             append o to the base document model
32     EG.cells = new boolean[] []
33 )

```

A valid input for these rules is for example the base document model $B_{blinker} = (M, F, m_1, c, s, f)$ and its associated metadata $A_{B_{blinker}} = (L, V, d)$, which together represent the "blinker" shape in the Game of Life.

$$\begin{aligned}
 M &= \{m_0, m_1, m_2, m_3, m_4, m_5, m_6\} = T, \\
 F &= \emptyset, \\
 c &= \{(m_0, \text{"init"}), (m_6, \text{"generation"})\}, \\
 s &= \{(m_0, m_1), (m_1, m_2), (m_2, m_3), (m_3, m_4), (m_4, m_5), (m_5, m_6)\}, \\
 f &= \emptyset, \\
 L &= \{\text{neighbours}, x, y\}, \\
 V &= \{0, 1, 2, \dots\}, \text{ and} \\
 d &= \{(m_1, \text{neighbours}, 1), (m_1, x, 1), (m_1, y, 0), \\
 &\quad (m_2, \text{neighbours}, 2), (m_2, x, 1), (m_2, y, 1), \\
 &\quad (m_3, \text{neighbours}, 1), (m_3, x, 1), (m_3, y, 2), \\
 &\quad (m_4, \text{neighbours}, 3), (m_4, x, 0), (m_4, y, 1), \\
 &\quad (m_5, \text{neighbours}, 3), (m_5, x, 2), (m_5, y, 1)\}.
 \end{aligned}$$

Using a suitable interpreter for the transformation rules, applied to $B_{blinker}$ they result in a continuously alternating output of a vertical bar or a horizontal bar, respectively. An equivalent Drools implementation and sample output can be found in appendix B.6. \square

This shows that in principle the transformation rules can be used to generate any document model that could be obtained through any other programming paradigm.

12.2 Quality of the Document Models

The first question when assessing the quality of document models is how this quality can be measured, i.e., if it is possible to define a norm $\|\cdot\| : \mathcal{DM} \rightarrow \mathbb{R}$, where \mathcal{DM} is the set of all document models. For document models, their adequacy (correctness and completeness) can only be measured with respect to a reference model that is “known” to be both correct and complete. We will call a document model adequate if it contains all and only these assertions that were intended to be in the model by the author of the extraction rules and background knowledge. Such a reference model will likely be constructed manually. In the sequel, we will refer to a given reference model as R .

We will now introduce and assess two different norms for the quality of document models. The first norm, $\|D\|_{steps}^R$ is defined as the minimal number of atomic steps necessary to transform a document model D into the reference model R . Atomic steps are the insertion, movement and deletion of fragments (i.e., the operations $+$, \pm , $-$ as defined in definition 4.2.7 on page 80), and the insertion, deletion and change of annotations (i.e., the operations $+_a$, $-_a$, $\#_a$ as defined in definition 4.2.18 on page 84).

The second norm, $\|D\|_{f-measure}^R$ is defined as the F-measure of a document model D w.r.t. a reference model R . The F-measure is defined as $2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$, where $\text{precision} = \frac{|\{\text{correct assertions}\}|}{|\{\text{correct assertions}\}| + |\{\text{incorrect assertions}\}|}$ and $\text{recall} = \frac{|\{\text{correct assertions}\}|}{|\{\text{correct assertions}\}| + |\{\text{missing assertions}\}|}$.

The F-measure is easy to calculate, requiring only to count the number of assertions in R ($|\{\text{correct assertions}\}|$), to count the number of assertions that are in R but not in D ($|\{\text{missing assertions}\}|$), and to count the number of assertions that are in D but not in R ($|\{\text{incorrect assertions}\}|$). $\|\cdot\|_{steps}^R$ is harder to calculate because of the atomic move operation for fragments: it is not only necessary to detect incorrect or missing assertions, but also if an incorrect sub-fragment assertion “fits” a missing one, i.e., if an incorrectly placed sub-fragment can be moved to another location and thereby solve two errors at once. But precisely this operation is what makes the steps-norm more meaningful, because the F-measure reveals less about how costly repairing the document model would be. If, for example, a sub-section had been erroneously placed into the wrong chapter, then this can be repaired with a single move operation, while the F-measure counts one missing assertion (the missing sub-section) and one incorrect assertion (the wrong placement of the sub-section), thus counting some errors twice.

Yet at the same time it can be argued that such structural errors should indeed be weighted more heavily than, say, a missed term. Combined with the higher cost of calculating $\|\cdot\|_{steps}^R$ and the advantage that the F-measure is limited to the range $[0, 1]$, we generally recommend using $\|\cdot\|_{f-measure}^R$.

The norms for semantic document models can be transferred without adaptation to semantic process models.

We have manually created reference document/process models R for one e-learning document written in ML3 and for one technical documentation written in DocBook, as well as for a process description written in Visio (NPB) and a process description written in Word (DLR).

The results for the ML3 document are easy to describe: of the 259 assertions in the reference model, the document model D_{ML3} obtained automatically through the processing rules and the background knowledge faithfully reproduced every single one and did not introduce any erroneous ones, resulting in $\|D_{ML3}\|_{f-measure}^R = 1$ and $\|D_{ML3}\|_{steps}^R = 0$.

The results for the other documents are not quite as perfect, but still quite good. The technical documentation, where we chose the “Gnome Integration Guide”, has a reference model with 215 assertions. In the document model $D_{DocBook}$ that was obtained automatically, 18 assertions were missing and 18 assertions were incorrect, resulting in $\|D_{DocBook}\|_{f-measure}^R = 0.9227$. But $\|D_{DocBook}\|_{steps}^R = 18$ better captures the underlying cause, which is that 18 fragments need to be moved to a different location.

The reason for these errors lies in the non-strict hierarchy of fragment types in the source document: there is no strict hierarchy were, for example, chapters are always parent fragments of sections, and sections are always parent fragments of sub-sections. Instead, sections can be parents of other sections, which leads to errors because the processing rules as defined in chapter 9 rely on a strict hierarchy to detect if one fragment is a sub-fragment of another. This is, however, easy to solve by adding a new rule that is triggered not when a new fragment starts, but when a fragment ends (cf. section 9.1.3). In the case of XML, the rule would match closing tags instead of opening tags. The conclusion of this new rule would then be responsible for maintaining the stack of fragments $\mathcal{E}_G.stack$ (cf. sections 5.1 and 9.1.3), in particular for popping the current fragment off the stack when the fragment’s end is reached in the source document. We will leave the straight-forward implementation to the imagination of the reader.

The reference process model for the Visio model consists of 133 assertions. One of these was missed, and one erroneous assertion was introduced in the process model P_{Visio} , leading to $\|P_{Visio}\|_{f-measure}^R = 0.9925$. But $\|P_{Visio}\|_{steps}^R = 2$ shows that this time, the error cannot be fixed with a single move (in fact, one deletion and one insertion are required). The cause of the error in this case is that the direction of an arrow in the Visio illustration was detected incorrectly, leading to a reference assertion from fragment A to fragment B instead of the other way around. Visio assigns to each connecting element (such as lines or arrows) a start and an end position, which are used by the processing rules to detect the direction of the arrow. However, it is possible to define an arrow that has the tip at the wrong end, i.e., that points to its starting point instead of its end point. This is what happened here. While this error can be fixed by having the processing rules check for the actual arrowhead (requiring more extensive background knowledge), perhaps the better solution in this case is to change the Visio document.

The reference process model for the Word-based process description is considerably larger than the others, containing 1341 assertions, six of which were missed when extracting P_{Word} , and one incorrect assertion was added. This results in $\|P_{Word}\|_{f-measure}^R = 0.9970$. The seven atomic operations needed to repair the model lead to $\|P_{Word}\|_{steps}^R = 7$. At one point, the keyword “process manager” was used to describe the tasks of a process manager, instead of (as defined in the background knowledge) introducing the name of an actual process manager. Since there exists background knowledge about the current personnel, this can be detected automatically and is, indeed, detected by one of the consistency criteria in the verification step. Six assertions were missed because of misspelt keywords or improper formatting. While some of them could be fixed by making the background more forgiving of small errors like mistypes, we believe that it is better to find these errors via the verification process and to correct them in the source document.

Note that due to the semi-formal nature of the process description document the correctness of many assertions are judgement calls, as there generally are no absolute truth values for many assertions. We have followed the same mindset for the evaluation as we did when collecting the background knowledge for the processing rules: it is better to collect too much data than too little. Therefore, the document model contains as much information as could reasonably be obtained, and if an assertion was not clearly false (as in the instance of the process manager) we regarded it as correct.

In figure 12.1 we show an overview of the qualitative results for the document models. To make its presentation more compatible with that of $\|\cdot\|_{f\text{-measure}}^R$, we normalised $\|\cdot\|_{steps}^R$ to the interval $[0, 1]$.

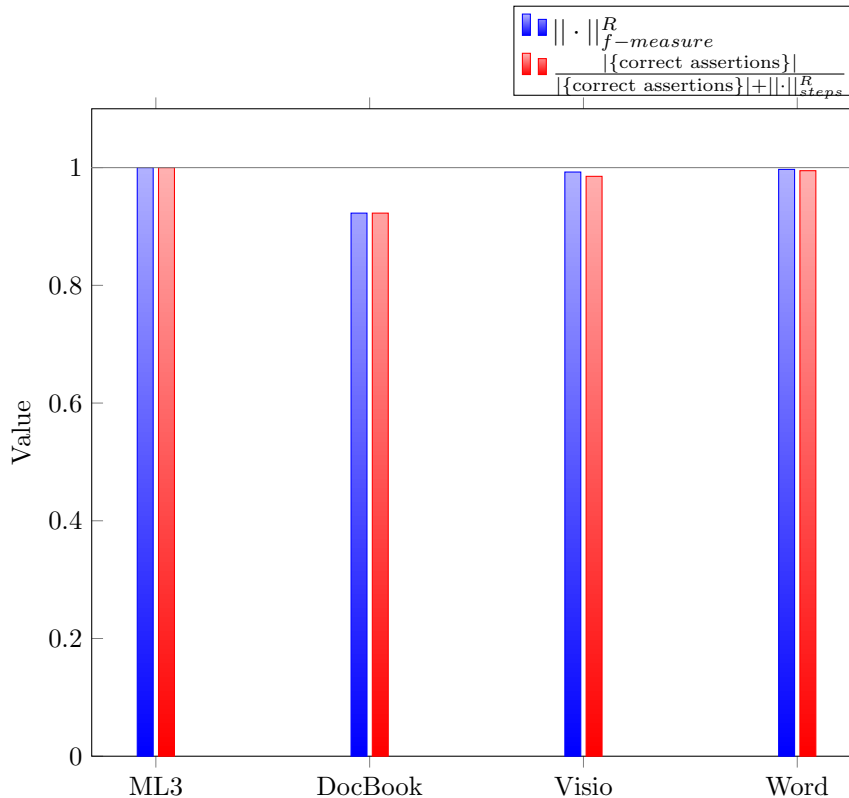


Figure 12.1: Quality of the extracted semantic document/process models

Despite the good results, it is obvious that the quality of the document model largely depends on the quality of the background knowledge. If the background knowledge used in the extraction had been of lower quality, the quality of the extracted document models would have suffered as well. Anything that is missing from the background knowledge cannot be found in the document model. We will investigate this relationship further after a discussion on how the quality of background knowledge can be measured in the first place.

12.3 Quality of the Background Knowledge

For assessing the quality of background knowledge, the same question as for the document models arises: how can it be measured? Can we define a norm for it? It is certainly possible to define a norm along the same lines as $\|\cdot\|_{f\text{-measure}}^R$. However, a reference knowledge base is necessary for a norm. But background knowledge usually reflects a best-effort approach, meaning that the best knowledge base that is available is already the one in use, and there is no secret better version that could serve as a reference. Instead, a usable approach is to have the existing knowledge bases

checked by domain experts and to count the errors they find (potentially normalised against the total number of assertions in the background knowledge).

Yet another option is to measure the quality of the knowledge bases indirectly, by finding and counting the errors in the document models that are caused by errors or omissions in the background knowledge. Ignoring all errors in the document model that are not caused by the background knowledge, the norms defined in the previous section can then be used to assess the quality of the background knowledge. This approach has the advantage that it is usually easier to spot errors in a document model (which resembles a concrete document) than in a highly abstract and formalised knowledge representation.

In our use cases, we corrected and extended the background knowledge until it did not cause any errors in the extracted document models (cf. section 12.2).

Volatility of the Background Knowledge

Background knowledge often is not constant, but changes over time (it certainly changes across domains, as we have already seen). For highly standardised formats such as ML3, DocBook or BPMN, this volatility is very low and changes are well documented.

Document formats that are only semi- or unofficially standardised or that receive frequent updates obviously change more often, requiring more or less extensive updates of the background knowledge. When Microsoft replaced the binary formats for its office suite with XML-based formats, large parts of existing knowledge bases suddenly became obsolete. Yet such sweeping changes are rare. More often the templates used in documents are changed, so that the knowledge base about formatting indicators must be updated. If updating the background knowledge is done in parallel with changing the templates, then the effort necessary is minimal. This is possible when the templates and the knowledge base are controlled by the same entity.

In other cases, the volatility can vary drastically, depending on the size and composition of the author group of a document. For example, small groups of authors that collaborate on a research paper or on a set of lecture notes often create their own informal standards that remain reasonably constant. When regarding different papers written by different authors, the differences are likely far more pronounced, even if they use a common template with a small number of formatting options or \LaTeX commands as the lowest common denominator. For websites or documents without a common template or at least common design guidelines, the background knowledge likely differs widely, and changes often.

12.4 Relationship between Qualities

Finally, we will now investigate the relationship between the quality of the background knowledge and the quality of the document models.

To this end, we selected a document, changed the background knowledge by removing or adding information, obtained a document model using the new background knowledge, and compared the new document model with the original model R . In particular, we selected one of the technical documentations in DocBook format and then proceeded as follows: for each combination of one or two assertions from the knowledge base about structure or from the knowledge base about structure, respectively, we removed the selected assertions from the background knowledge, created the document model, and calculated $\|\cdot\|_{f\text{-}measure}^R$. In addition, we successively added one, two, and three invalid (but plausible) assertions to the knowledge base about structure or to the knowledge base about structure, respectively, created the document model, and calculated $\|\cdot\|_{f\text{-}measure}^R$.

Figure 12.2 shows the number of missing and incorrect assertions for various sets of modified background knowledge. The set RS1 contains all modified versions of the knowledge base where one assertion about the structure, e.g., which XML element indicates a specific structural type, has been removed. RS2 contains all knowledge bases where two such assertions were removed. RD1 and RD2 are similar sets, but refer to assertions about data, e.g., which XML attribute indicates a specific data annotation. The sets AS and AD contain all knowledge bases where one, two or three assertions about structure or data, respectively, have been added. The original knowledge base contains 21 data assertions and 36 structural assertions, and the original document model R that serves as point of reference here contains 306 assertions.

The values for missing assertions for the two sets AS and AD are all zero, because the additional knowledge only leads to new assertions in the document model. Vice versa, however, removing assertions from the background knowledge can lead to incorrect attributions of data, and thus to incorrect assertions in the document model, and not just to missing assertions.

The most notable feature of the graph are the two large values for the maximum number of missing assertions in both RS1 and RS2. Such catastrophic results occur when structural indicators for central fragment types like sections or paragraphs are removed. On the other hand, in cases like this, the cause for the error is both easily determined and easily fixed. The other maximum values are far less pronounced, indicating that the approach is – at least for some file formats – reasonably robust even against extreme omissions in the background knowledge. This is corroborated by the very low average and mean values, and has been confirmed for a random sample of other file formats. The values for the AD set are slightly higher because some of the assertions added to the background knowledge have a particularly high selection yield.

The average, mean, and minimal values for the F-measure for the various sets of background knowledge can be seen in figure 12.3. The average and especially the mean values are generally very close to the optimum of 1.0. The lower values for RS1 and RS2 reflect the effect of missing important fragments, as discussed above. This effect is very distinctly shown in the minimum values and is based on the low recall values.

These figures show that, apart from a few central structural omissions, a small amount of error in the background knowledge only has a limited effect on the quality of the document model. Large errors like the omission of central structural types are easy to deal with. More importantly, however, they are unlikely to occur in the first place for knowledge bases created by domain experts, barring accidents or sabotage.

Note, however, that no amount of robustness can save the approach if there simply is too little background knowledge, or if the background knowledge is of insufficient quality. Because of the different selectivity of different assertions, we cannot give any general figures on the amount of assertions required, or exactly how many errors the knowledge base may contain before the quality of the document model deteriorates too much.

12.5 Applicability to Different Types of Documents

As we have already seen, our approach can be effectively applied to hyperdocuments, to web-documents, and to various types of office documents. We will now discuss if the approach can also be used on other types of documents (cf. section 4.1.2).

The simplest text-centred types of documents are plain text documents. They usually have little inherent structure and are therefore not very good candidates for structural processing.

The technical accessibility of PDF documents is more complex than for many other document formats, but in principle they can be treated just like any other hypertext. Yet a more sensible approach is to regard the source documents from which the PDF files were generated.

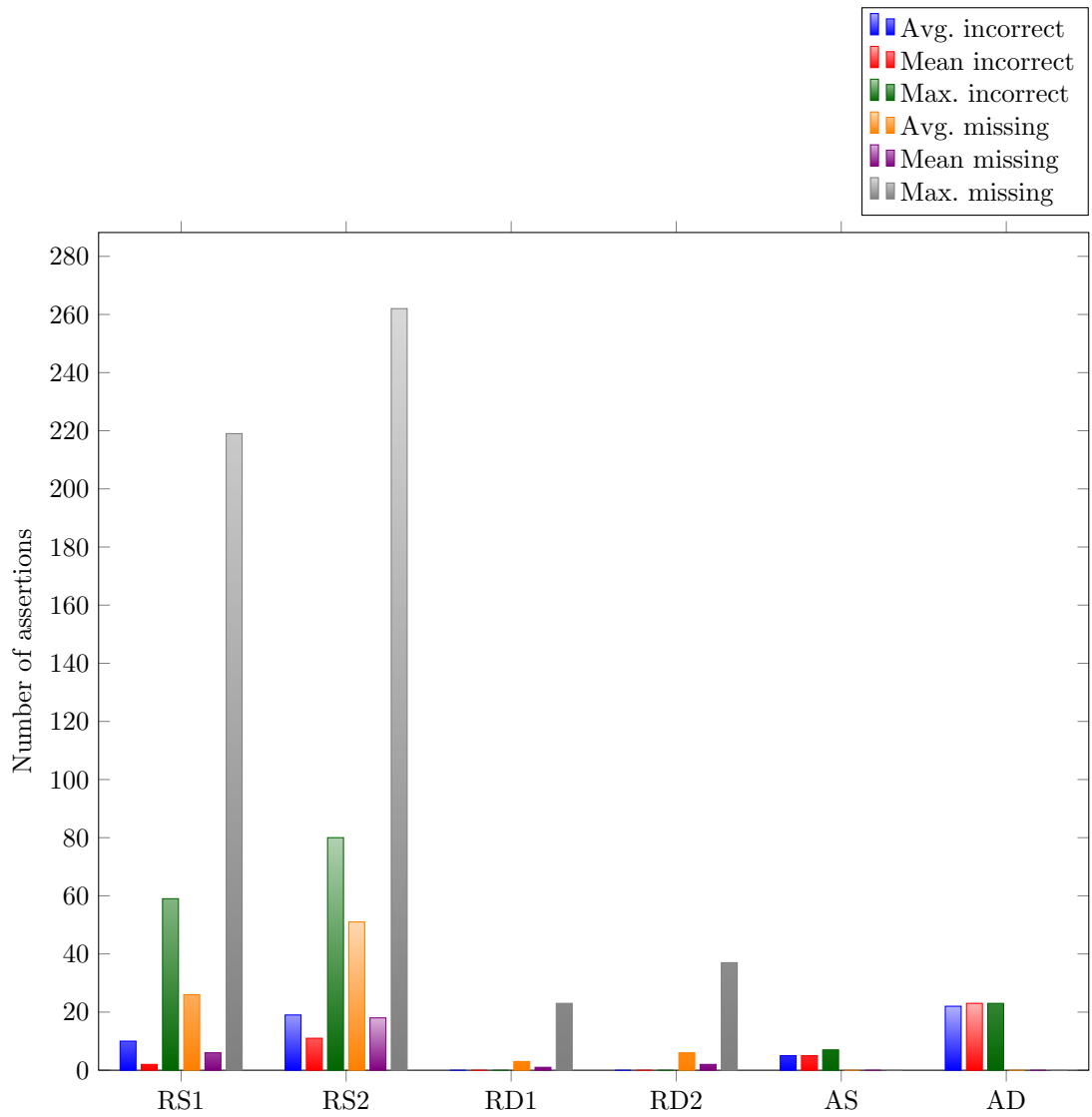


Figure 12.2: Incorrect and missing assertions in document models with modified background knowledge

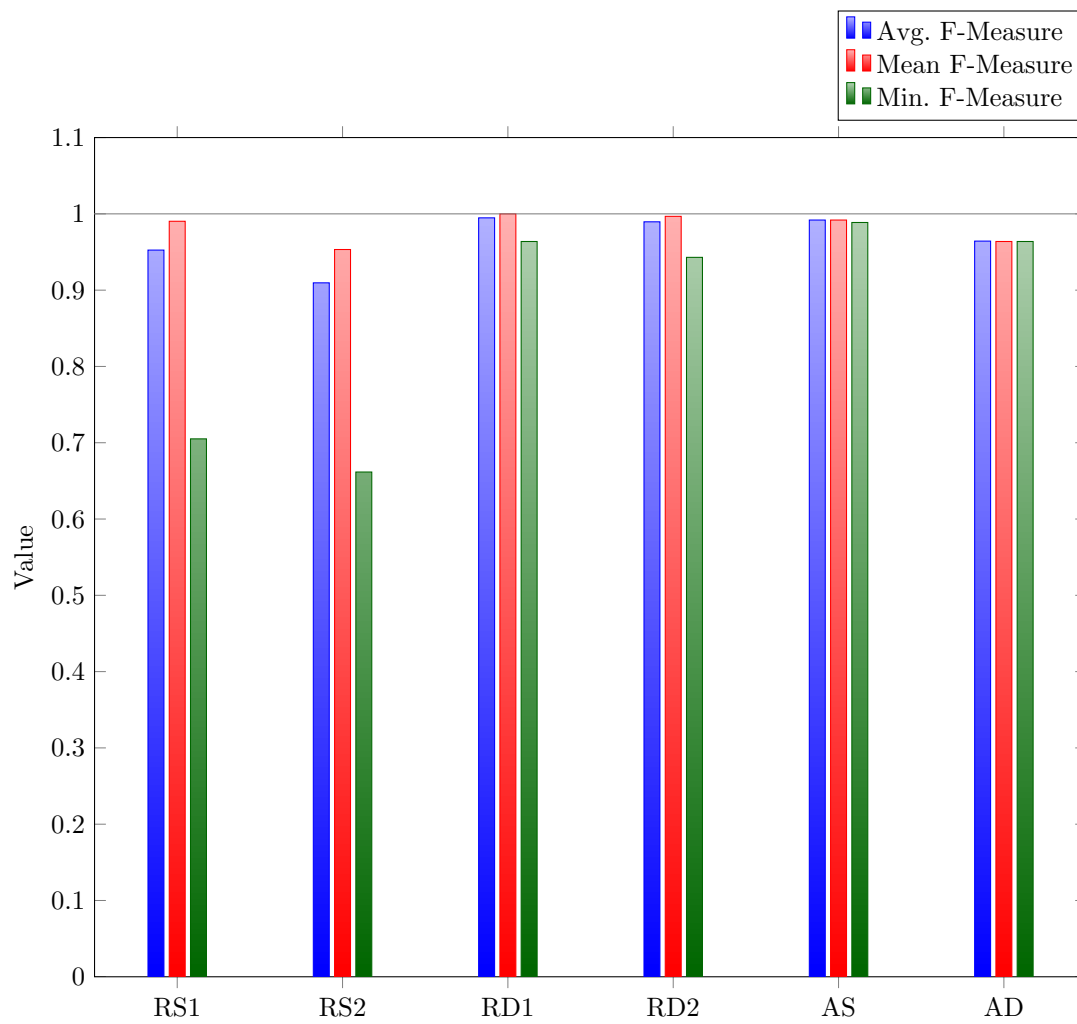


Figure 12.3: F-measure for document models with modified background knowledge

SVG is a vector graphics format that is based on XML and therefore highly structured on the technical level. In [Plö09], a successful attempt was made to derive high-level structural and semantic information about the image's content from this technical data, demonstrating the suitability of at least some vector documents for our approach.

Image, audio and video documents such as bitmaps, wave-audio and traditional linear video have very little technical structure to exploit. Advanced feature-, voice- and object recognition techniques would be required to learn anything about their content's structure and semantics.

As stated in section 4.1.5, data in a database lacks the visualisation instructions and the structural cohesion that characterise our understanding of a document. Such instructions can be provided however, for example by creating a hyperdocument and integrating media fragments from a database. On the other hand, if the content of a hypertext changes dynamically, for example because it is randomly loaded from a database, or because it is frequently updated, then it is hard to represent this dynamic nature in a semantic document model. A document model can easily represent a snap shot of a dynamic page, but any change of the source document must trigger a re-creation of the model.

Conclusion

In this chapter, we have introduced two different norms for measuring the quality of semantic document models. We also have discussed new ways to measure the quality of background knowledge. Additionally, we have investigated the impact of errors and omissions in background knowledge on the quality of the resulting document models. Unsurprisingly, the quality of the background knowledge directly affects the quality of the document models that can be obtained. For small errors and omissions, however, this effect was smaller than expected, still resulting in basically sound models. Finally, we have discussed how several different types of documents lend themselves to the effectiveness of our approach.

Part V
Conclusion

Chapter 13

Conclusion

In this thesis we have developed a formal model to represent documents on a technical level, as well as a formal model to represent documents on a structural and semantic level. We have created a new technique for obtaining the second type of model from the first, using background knowledge. By analysing runtime and scaling performance, we have shown that our implementation of this technique is both efficient and suitable for live processing. We have also introduced new methods for measuring the quality of semantic document models, and with these methods we have shown that our technique is also effective and yields results of very high quality. In addition, we have demonstrated several ways in which background knowledge can be obtained, and how its quality can be determined.

Our technique for obtaining a semantic document model has proven to be easily transferable to new document formats and new domains. It is even possible to transfer the approach to an entirely new application domain, as was shown by applying the technique intended for documents and document models to processes and process models. This transfer requires almost no changes to the transformation rules used, only the background knowledge needs to be updated or replaced. This could be achieved by adhering to a strict separation of extraction logics and domain knowledge, and by following a structured approach based on the formalisation of both the starting point and the end point of the technique. The level of abstraction that we used is well suited to make the mapping from the technical model onto the semantic model effective and adaptable.

In addition to the semantic and structural data obtained from the original document, it is possible to extend a semantic document model with further information, and to extract a multitude of different models for various use-cases from it. We have provided different approaches to this semantic extension and weighted their advantages and disadvantages. This semantic processing is possible because our novel document model directly exposes the relevant structure and semantics that were only indirectly available in the original document.

The obtained semantic document model has successfully been used as a source for a formal verification system as introduced in the VERDIKT research project. This system finds errors based on consistency criteria in a document, and attempts to identify the underlying cause in the document. This application covers a full circle in terms of document models: starting with the original document and a technical model of this document, a semantic model is obtained, from which one or more verification models are extracted that are used to find errors and error locations, which in turn point back to the original document.

Chapter 14

Extensions and Future Work

The principles developed in this work can also be applied to interactive documents, videos and programs because these documents have the necessary inherent structure and can be modelled as base document models. Especially the adaptation to hypervideos and to program verification would most likely be a rewarding extension of this work.

Another interesting topic is to consider the life cycle of documents, where we have only scratched the surface. The life cycle of a document can be treated as a sequence of snap shot models, or the entire history of a document can be included in a single model. At this point, the respective advantages are not entirely clear. For documents with frequent changes, a method to update an existing document model, as opposed to re-building the whole model each time, would also be advantageous.

A novel application for the results of this thesis could be an advanced search algorithm for documents that not only considers related topics, but that allows for restrictions in the structure, for example producing only results from certain chapters, or different topics that occur in the same chapter.

There is also potential in expanding on the concept of the narrative path and the reading path, which is the order of document fragments in which the author intends them to be read, and in which the reader actually reads them, respectively. Especially in fiction, the path of the “action” may not always coincide with the typical narrative/reading path, which instead often alternates between multiple narratives. Finding inconsistencies or omissions in the timeline of the narrative and in the mapping of this narrative onto the reading path is most likely an exiting endeavour.

Another issue that we will leave to future work are the multiple layers of truth as discussed in the outlook of section 4.2.1.

Bibliography

- [ABF04] M. Alpuente, D. Ballis, and M. Falaschi. Verdi: An automated tool for web sites verification. In *Proc. of JELIA 2004*, volume 3299 of *LNAI*, pages 726–729. Springer, 2004.
- [AG05] Renzo Angles and Claudio Gutierrez. Querying RDF data from a graph database perspective. In *In Proceedings of the Second European Semantic Web Conference*, pages 346–360, 2005.
- [Ahr12] Norbert Ahrend. Nationale Prozessbibliothek. <http://www.prozessbibliothek.de/>, 2012. Visited 05/2013.
- [AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.
- [AL07] Sören Auer and Jens Lehmann. What have Innsbruck and Leipzig in common? Extracting semantics from wiki content. In *4th European Semantic Web Conference*, 2007.
- [All10] Thomas Allweyer. *BPMN 2.0 - Introduction to the Standard for Business Process Modeling*. BoD, 2010.
- [AMMH07] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases, VLDB '07*, pages 411–422. VLDB Endowment, 2007.
- [AMMH09] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Sw-store: a vertically partitioned dbms for semantic web data management. *The VLDB Journal*, 18(2):385–406, 2009.
- [AMS07] Kemafor Anyanwu, Angela Maduko, and Amit Sheth. SPARQ2L: Towards Support For Subgraph Extraction Queries in RDF Databases. In *16th International World Wide Web Conference (WWW2007)*, pages 797–806, New York, NY, USA, May 2007. ACM.
- [AP05] Olivier Aubert and Yannick Prié. Advene: active reading through hypervideo. In *ACM Hypertext'05*, pages 235–244, 2005.
- [BBH03] Steven R. Bagley, David F. Brailsford, and Matthew R. B. Hardy. Creating reusable well-structured pdf as a sequence of component object graphic (cog) elements. In *Proceedings of the 2003 ACM symposium on Document engineering, DocEng '03*, pages 58–67, New York, NY, USA, 2003. ACM.

- [BBP07] Tracy Bost, Phillippe Bonnard, and Mark Proctor. Implementation of Production Rules for a RIF Dialect: A MISMO Proof-of-Concept for Loan Rates. In Adrian Paschke and Yevgen Biletskiy, editors, *RuleML*, volume 4824 of *Lecture Notes in Computer Science*, pages 160–165. Springer, 2007.
- [BCF⁺10] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. XQuery 1.0: An XML Query Language (Second Edition), W3C Recommendation 14 December 2010. <http://www.w3.org/TR/xquery/>, 2010. Visited 05/2013.
- [BCM⁺03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
- [BG04] Dan Brickley and R. V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema, W3C Recommendation 10 February 2004. <http://www.w3.org/TR/2004/REC-rdf-schema-20040210/>, 2004. Visited 05/2013.
- [BHP87] Wiebe E. Bijker, Thomas P. Hughes, and Trevor J. Pinch, editors. *The Social Construction of Technological Systems*. MIT Press, 1987.
- [BKvH02] Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In Ian Horrocks and James Hendler, editors, *The Semantic Web ISWC 2002*, volume 2342 of *Lecture Notes in Computer Science*, pages 54–68. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-48005-6_7.
- [BLK⁺09] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. DBpedia - a crystallization point for the web of data. *Web Semantics: Science, Services and Agents on the World Wide Web*, July 2009.
- [BM04a] Dave Beckett and Brian McBride. RDF/XML Syntax Specification (Revised), W3C Recommendation 10 February 2004. <http://www.w3.org/TR/REC-rdf-syntax/>, 2004. Visited 05/2013.
- [BM04b] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition, W3C Recommendation 28 October 2004. <http://www.w3.org/TR/xmlschema-2/>, 2004. Visited 05/2013.
- [BNJ03] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, March 2003.
- [Bol01] Jay David Bolter. *Writing Space: Computers, Hypertext, and the Remediation of Print*. Lawrence Erlbaum Associates, second edition edition, 2001.
- [Bor48] Jorge Luis Borges. *The Garden of Forking Paths*. Editorial Sur, 1948. Translated from the original Spanish by Anthony Boucher. Original title: El jardín de senderos que se bifurcan (1941).
- [Bor99] Jorge Luis Borges. *John Wilkins' Analytical Language*. Penguin Books, 1999. Eliot Weinberger, Selected nonfictions. Translated from the original Spanish. Original title: El idioma analítico de John Wilkins (1942).

- [BPSM⁺08] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), W3C Recommendation 26 November 2008. <http://www.w3.org/TR/2008/REC-xml-20081126/>, 2008. Visited 05/2013.
- [BTW] Harold Boley, Said Tabet, and Gerd Wagner. RuleML. <http://ruleml.org/>. Visited 05/2013.
- [Bus45] Vannevar Bush. As we may think. *The Atlantic Monthly*, July 1945.
- [BvHH⁺] Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference, W3C Recommendation 10 February 2004. <http://www.w3.org/TR/owl-ref/>. Visited 05/2013.
- [CD99] James Clark and Steve DeRose. XML Path Language (XPath) Version 1.0, W3C Recommendation 16 November 1999. <http://www.w3.org/TR/xpath/>, 1999. Visited 05/2013.
- [CDD⁺04] Jeremy J. Carroll, Ian Dickinson, Chris Dollin, Dave Reynolds, Andy Seaborne, and Kevin Wilkinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, WWW Alt. '04, pages 74–83, New York, NY, USA, 2004. ACM.
- [CMLE08] Niklas Carlsson, Anirban Mahanti, Zongpeng Li, and Derek Eager. Optimized periodic broadcast of non-linear media, August 2008.
- [Cow06] Phil Cowans. *Probabilistic Document Modelling*. PhD thesis, University of Cambridge, 2006.
- [Cyg05] Richard Cyganiak. A relational algebra for SPARQL. Technical report, HP Labs, 2005.
- [Dal09] Andrew Dalby. *The World and Wikipedia*. Siduri Books, 2009.
- [DEAJ10] Don Day, Kristen James Eberlein, Robert D. Anderson, and Gershon Joseph. Darwin Information Typing Architecture. <http://docs.oasis-open.org/dita/v1.2/os/spec/DITA1.2-spec.html>, 2010. Visited 05/2013.
- [dMW10] Gerard de Melo and Gerhard Weikum. Menta: inducing multilingual taxonomies from wikipedia. In *Proceedings of the 19th ACM international conference on Information and knowledge management*, CIKM '10, pages 1099–1108, New York, NY, USA, 2010. ACM.
- [Dok06] Jiri Dokulil. Evaluation of SPARQL queries using relational databases, 2006.
- [Doo95] Robert B. Doorenbos. *Production matching for large learning systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.
- [ECTO09] Brendan Elliott, En Cheng, Chimezie Thomas-Ogbuji, and Z. Meral Ozsoyoglu. A complete translation from SPARQL into efficient SQL. In *IDEAS '09: Proceedings of the 2009 International Database Engineering & Applications Symposium*, pages 31–42, New York, NY, USA, 2009. ACM.

- [Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics*, pages 996–1072. Elsevier, 1990.
- [EPU] EPUB 3 Overview, Recommended Specification 11 October 2011. <http://www.idpf.org/epub/30/spec/>. Visited 05/2013.
- [Esp97] Espen J. Aarseth. *Cybertext - Perspectives on Ergodic Literature*. The Johns Hopkins University Press, Baltimore and London, 1997.
- [ESS05] U. Egly, B. Schiemann, and J. Schneeberger. Technical documentation authoring based on semantic web methods. *Künstliche Intelligenz*, 2:56–59, 2005.
- [FBY92] William B. Frakes and Ricardo A. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.
- [Fra] Franz Inc. AllegroGraph. <http://www.franz.com/agraph/allegrograph/>. Visited 05/2013.
- [Fre09] Burkhard Freitag. Datenmodellierung, 2009. Lecture notes.
- [Fre11] Burkhard Freitag. Semantische technologien, 2011. Lecture notes.
- [FS90] Richard Furuta and P. David Stotts. A functional meta-structure for hypertext models and systems. *Electronic Publishing*, 3(4):179–205, 1990.
- [FWJS08] Burkhard Freitag, Franz Weitzl, Mirjana Jakšić, and Christian Schönberg. Verdikt – Verifikation semistrukturierter Dokumente im Kontext. <http://www.verdikt.uni-passau.de>, 2008. Visited 05/2013.
- [Gar70] Martin Gardner. Mathematical games: The fantastic combinations of john conway’s new solitaire game “life”. *Scientific American*, pages 120–123, October 1970.
- [Gar87] Pankaj K. Garg. Abstraction mechanisms in hypertext. In *Proceedings of the ACM conference on Hypertext*, HYPERTEXT ’87, pages 375–395, New York, NY, USA, 1987. ACM.
- [GHP10] Stephan Gillmeier, Urs Hengartner, and Sandro Pedrazzi. Wie man mit Wikipedia semantische Verfahren verbessern kann. In *HMD - Praxis der Wirtschaftsinformatik 47*, 2010.
- [Goe04] Lutz Goertz. *Wie interaktiv sind Medien?*, pages 97–117. Campus Verlag, Frankfurt am Main, 1 edition, 2004.
- [Got06] David Gotz. Scalable and adaptive streaming for non-linear media. In *Proceedings of the 14th annual ACM international conference on Multimedia*, MULTIMEDIA ’06, pages 357–366, New York, NY, USA, 2006. ACM.
- [Gru06] Schema Gruppe. Schema ST4 Leistungsbeschreibung. SCHEMA Electronic Documentation Solutions GmbH, 2006.
- [Haa02] Johannes Haack. Interaktivität als Kennzeichen von Multimedia und Hypermedia. *Information und Lernen mit Multimedia*, 2002.

- [Har05] Steve Harris. SPARQL query processing with conventional relational database systems. In *International Workshop on Scalable Semantic Web Knowledge Base Systems*, pages 235–244, 2005.
- [HBBB07] Rinke Hoekstra, Joost Breuker, Marcello Di Bello, and Alexander Boer. The lkif core ontology of basic legal concepts. In Pompeu Casanovas, Maria Angela Biasiotti, Enrico Francesconi, and Maria-Teresa Sagri, editors, *Proceedings of the Workshop on Legal Ontologies and Artificial Intelligence Techniques (LOAIT)*, volume 321 of *CEUR Workshop Proceedings*, pages 43–63. CEUR-WS.org, 2007.
- [HKR09] Pascal Hitzler, Markus Krötzsch, and Sebastian Rudolph. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, 2009.
- [HKS06] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The even more irresistible sroiq. In Patrick Doherty, John Mylopoulos, and Christopher A. Welty, editors, *KR*, pages 57–67. AAAI Press, 2006.
- [HPSB⁺04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosz, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML, W3C Member Submission 21 May 2004. <http://www.w3.org/Submission/SWRL/>, 2004. Visited 05/2013.
- [HPSH03] Ian Horrocks, Peter F. Patel-Schneider, and Frank Van Harmelen. From shiq and rdf to owl: The making of a web ontology language. *Journal of Web Semantics*, 1:2003, 2003.
- [HR04] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.
- [HS94] Frank Halasz and Mayer Schwartz. The dexter hypertext reference model. *Commun. ACM*, 37(2):30–39, February 1994.
- [HS12] Steve Harris and Andy Seaborne. SPARQL 1.1 Query Language. <http://www.w3.org/TR/sparql11-query/>, 2012. Visited 05/2013.
- [HSB07] Martin Hepp, Katharina Siorpaes, and Daniel Bachlechner. Harvesting wiki consensus: Using Wikipedia entries as vocabulary for knowledge management. *IEEE Internet Computing*, 11(5):54–65, 2007.
- [HTMa] HTML 4.01 Specification, W3C Recommendation 24 December 1999. <http://www.w3.org/TR/html4/>. Visited 05/2013.
- [HTMb] HTML5: A vocabulary and associated APIs for HTML and XHTML, W3C Candidate Recommendation 17 December 2012. <http://www.w3.org/TR/html5/>. Visited 05/2013.
- [Jel02] R. Jelliffe. The schematron assertion language 1.6. <http://xml.ascc.net/resource/schematron/>, 2002. Visited 05/2013.
- [Jen] Apache Foundation. Jena. <http://jena.apache.org/>. Visited 05/2013.
- [Joy87] Michael Joyce. *Afternoon, a story*, 1987. Electronic publication.

- [KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF) Concepts and Abstract Syntax, W3C Recommendation 10 February 2004. <http://www.w3.org/TR/rdf-concepts/>, 2004. Visited 05/2013.
- [KJ07] Krys J. Kochut and Maciej Janik. SPARQLeR: Extended SPARQL for Semantic Association Discovery. In *Proc. of the 4th European Semantic Web Conference (ESWC)*, pages 145–159, Berlin, Heidelberg, 2007. Springer-Verlag.
- [Knu73] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.
- [Koc06] Christoph Koch. On the complexity of nonrecursive xquery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, December 2006.
- [Kol08] Sergiy Kolesnikov. Metadata extraction from LMML. Bachelor thesis, Chair for Information Management, University of Passau, 2008.
- [Kos04] Harald Kosch. *Distributed Multimedia Database Technologies Supported by MPEG-7 and MPEG-21*. CRC Press, 2004.
- [KVV⁺07] Markus Krötzsch, D. Vrandečić, M. Völkel, H. Haller, and Rudi Studer. Semantic Wikipedia. *Journal of Web Semantics*, 5:251–261, 2007.
- [LCS09] Library of Congress Subject Headings. <http://id.loc.gov/authorities/subjects.html>, 2009. Visited 05/2013.
- [Lev94] David M. Levy. Fixed or fluid? Document stability and new media. In *Proceedings of the 1994 ACM European conference on Hypermedia technology*, ECHT '94, pages 24–31, New York, NY, USA, 1994. ACM.
- [LF73] F. W. Lancaster and E.G. Fayen. *Information Retrieval On-Line*. Melville Publishing Company, 1973.
- [Llo87] J. W. Lloyd. *Foundations of logic programming (2nd extended ed.)*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.
- [LM95] David M. Levy and Catherine C. Marshall. Going digital: A look at assumptions underlying digital libraries. *Commun. ACM*, 38(4):77–84, 1995.
- [LM07] Preetha Lakshmi and Chris Mueller. Comparing path-based and vertically-partitioned rdf databases, 2007.
- [Mil06] George A. Miller. WordNet – a lexical database for the English language. <http://wordnet.princeton.edu/>, 2006. Visited 05/2013.
- [MMGK12] Britta Meixner, Katarzyna Matusik, Christoph Grill, and Harald Kosch. Towards an easy to use authoring tool for interactive non-linear video. *Multimedia Tools and Applications*, pages 1–26, 2012.
- [MnPG07] Sergio Muñoz, Jorge Pérez, and Claudio Gutierrez. Minimal deductive systems for rdf. In *ESWC '07: Proceedings of the 4th European conference on The Semantic Web*, pages 53–67, Berlin, Heidelberg, 2007. Springer-Verlag.
- [Mou92] Stuart Moulthrop. Victory garden, 1992. Electronic publication.

- [MS04] Christoph Meinel and Harald Sack. *WWW – Kommunikation, Internetworking, Web Technologien*. Springer, 2004.
- [MW08] David Milne and Ian H. Witten. An Effective, Low-Cost Measure of Semantic Relatedness Obtained from Wikipedia Links. In *Proceedings of the first AAAI Workshop on Wikipedia and Artificial Intelligence (WIKIAI'08)*, Chicago, US, 2008.
- [MWM08] Olena Medelyan, Ian H. Witten, and David Milne. Topic Indexing with Wikipedia. In *Proceedings of the WIKI-AI: Wikipedia and AI Workshop at the AAAI'08 Conference*, Chicago, US, 2008.
- [NCEF02] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein. xlinkit: a consistency checking and smart link generation service. *ACM Transactions on Internet Technology (TOIT)*, 2(2):151–185, 2002.
- [Nel80] Theodor H. Nelson. *Literary Machines*. Mindful Press, 1980.
- [Neo] Neo Technology Inc. Neo4j. <http://neo4j.org/>. Visited 05/2013.
- [NS08] Vivi Nastase and Michael Strube. Decoding Wikipedia categories for knowledge acquisition. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1219–1224. AAAI Press, 2008.
- [Ope] OpenLink Software. Virtuoso Universal Server. <http://virtuoso.openlinksw.com/>. Visited 05/2013.
- [OVvdA⁺07] Chun Ouyang, Eric Verbeek, Wil M. P. van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M. ter Hofstede. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67(2-3):162–198, 2007.
- [OWL] University of Manchester. The OWL API. <http://owlapi.sourceforge.net/>. Visited 05/2013.
- [Pag00] Margherita Pagani. *Multimedia and Interactive Digital TV: Managing the Opportunities Created by Digital Convergence*. IRM Press, 2000.
- [PAG09] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3):16:1–16:45, September 2009.
- [PB09] Eric Prud'hommeaux and Alexandre Bertails. A Mapping of SPARQL Onto Conventional SQL. <http://www.w3.org/2008/07/MappingRules/StemMapping>, 2009.
- [PBB11] Alexander J. Pinkney, Steven R. Bagley, and David F. Brailsford. Reflowable documents composed from pre-rendered atomic components. In *Proceedings of the 11th ACM symposium on Document engineering, DocEng '11*, pages 163–166, New York, NY, USA, 2011. ACM.
- [PCC⁺11] Ricardo Farias Bidart Piccoli, Rodrigo Chamun, Nicole Carrion Cogo, João Batista Souza de Oliveira, and Isabel Harb Manssour. A novel physics-based interaction model for free document layout. In *Proceedings of the 11th ACM symposium on Document engineering, DocEng '11*, pages 153–162, New York, NY, USA, 2011. ACM.
- [PDF] PDF Reference, fifth Edition. <http://partners.adobe.com/public/developer/en/pdf/PDFReference16.pdf>. Visited 05/2013.

- [Plö09] Jörn Plötz. Metadatenextraktion aus Vektorgrafiken. Bachelor thesis, Chair for Information Management, University of Passau, 2009.
- [Pre08] Helmuth Pree. DITA Testfallgenerator für das Verdikt Verifikationssystem. Bachelor thesis, Chair for Information Management, University of Passau, 2008.
- [Pro07] Mark Proctor. Relational Declarative Programming with JBoss Drools. In Viorel Negru, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, editors, *SYNASC*, page 5. IEEE Computer Society, 2007.
- [PS07] Simone Paolo Ponzetto and Michael Strube. Deriving a large-scale taxonomy from wikipedia. In *AAAI*, pages 1440–1445. AAAI Press, 2007.
- [PS08] Eric Prud’hommeaux and Andy Seaborne. SPARQL Query Language for RDF. <http://www.w3.org/TR/rdf-sparql-query/>, 2008. Visited 05/2013.
- [PVB09] Mark Proctor, Kris Verlaenen, and Alexander Bagerman. Drools JBoss Rules 5.0. <http://www.jboss.org/drools/>, 2009. Visited 05/2013.
- [Que61] Raymond Queneau. A hundred thousand billion poems, 1961. Translated from the original French by Stanley Chapman. Original title: Cent mille milliards de poèmes.
- [Ren] Paul Rendell. A Turing Machine in Conway’s Game of Life, extendable to a Universal Turing Machine. <http://www.rendell-attic.org/gol/tm>. Visited 05/2013.
- [RG65] Herbert Rubenstein and John B. Goodenough. Contextual correlates of synonymy. *Commun. ACM*, 8(10):627–633, October 1965.
- [Röt99] Florian Rötzer. *Megamaschine Wissen*. Campus Verlag, Frankfurt/New York, 1999.
- [RRZvdH03] A. L. Rector, J. E. Rogers, P. E. Zanstra, and E. van der Haring. OpenGALEN: Open source medical terminology and tools. *Proceedings of the AMIA Annual Symposium*, page 982, 2003.
- [Sch04] J. Scheffczyk. *Consistent Document Engineering*. Dissertation, Universität der Bundeswehr München, 2004.
- [Sch07] Simon Schenk. A SPARQL Semantics Based on Datalog. In *KI ’07: Proceedings of the 30th annual German conference on Advances in Artificial Intelligence*, pages 160–174, Berlin, Heidelberg, 2007. Springer-Verlag.
- [Sch10] Christian Schlupp. Grafische Spezifikation einer Regelsprache nach Prinzipien der Human Computer Interaction. Bachelor thesis, Chair for Information Management, University of Passau, 2010.
- [Ses] Aduna Software. Sesame. <http://www.openrdf.org/>. Visited 05/2013.
- [SF89] P. David Stotts and Richard Furuta. Petri net based hypertext: Document structure with browsing semantics. *ACM TRANSACTIONS ON INFORMATION SYSTEMS*, 7:3–29, 1989.
- [SF99] Christian Süß and Burkhard Freitag. Learning Material Markup Language, 1999.

- [SF09] Christian Schönberg and Burkhard Freitag. Evaluating RDF Querying Frameworks for Document Metadata. Technical Report MIP-0903, Univ. of Passau, 2009.
- [SFC98] David P. Stotts, Richard Furuta, and Cyrano R. Cabarrus. Hyperdocuments as Automata: Verification of Trace-based Browsing Properties by Model Checking. *ACM Trans. Inf. Syst.*, 16(1):1–30, January 1998.
- [SGD⁺09] Florian Stegmaier, Udo Grbner, Mario Dller, Harald Kosch, and Gero Baese. Evaluation of current rdf database solutions. In *Proceedings of the 10th International Workshop on Semantic Multimedia Database Technologies (SeMuDaTe 2009)*, volume 239, pages 39–55, Graz, Austria, December 2009.
- [SGK⁺08] Lefteris Sidiropoulos, Romulo Goncalves, Martin Kersten, Niels Nes, and Stefan Manegold. Column-store support for rdf data management: not all swans are white. *Proc. VLDB Endow.*, 1(2):1553–1563, 2008.
- [SJWF09] Christian Schönberg, Mirjana Jakšić, Franz Weitzl, and Burkhard Freitag. Verification of Web-Content: A Case Study on Technical Documentation. In *Proc. of the 5th International Workshop on Automated Specification and Verification of Web Systems (WWV 09)*, Linz, Austria, 2009.
- [SKO] SKOS Simple Knowledge Organization System Reference, W3C Proposed Recommendation 15 June 2009. <http://www.w3.org/TR/2009/PR-skos-reference-20090615/>. Visited 05/2013.
- [SKS06] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.
- [SKW07] Fabian M. Suchanek, G. Kasneci, and Gerhard Weikum. Yago: a core of semantic knowledge. In *WWW '07: Proc. of the 16th intern. conference on World Wide Web*, pages 697–706, New York, NY, USA, 2007. ACM Press.
- [SMI] Synchronized Multimedia Integration Language (SMIL 3.0), W3C Recommendation 01 December 2008. <http://www.w3.org/TR/smil/>. Visited 05/2013.
- [SMK97] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB*, 6(3):191–208, 1997.
- [SPF10] Christian Schönberg, Helmuth Pree, and Burkhard Freitag. Rich Ontology Extraction and Wikipedia Expansion Using Language Resources. In *Proceedings of the 11th International Conference on Web-Age Information Management (WAIM'10)*, Jiuzhaigou, China, 2010.
- [SS89] Manfred Schmidt-Schauß. Subsumption in KL-ONE is undecidable. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 421–431, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
- [SSD09] Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of constraint handling rules. *ACM Trans. Program. Lang. Syst.*, 31(2):8:1–8:42, February 2009.
- [Ste10] Katharina Sterner. Regelbasierte Modellgenerierung aus E-Learning Dokumenten. Bachelor thesis, Chair for Information Management, University of Passau, 2010.

- [SVG] Scalable Vector Graphics (SVG) Full 1.2 Specification, W3C Working Draft 13 April 2005. <http://www.w3.org/TR/SVG12/>. Visited 05/2013.
- [SW88] John B. Smith and Stephen F. Weiss. Hypertext: Introduction to the special issue. *Commun. ACM*, 31(7):816–819, July 1988.
- [SWF11] Christian Schönberg, Franz Weigl, and Burkhard Freitag. Verifying the Consistency of Web-based Technical Documentations. *Journal of Symbolic Computation, Special Issue on Automated Specification and Verification of Web Systems*, 46(2):183–206, February 2011.
- [SWJF09] Christian Schönberg, Franz Weigl, Mirjana Jakšić, and Burkhard Freitag. Logic-based Verification of Technical Documentation. In *Proceedings of the 9th ACM symposium on Document engineering, DocEng '09*, pages 251–252, New York, NY, USA, 2009. ACM.
- [SWY75] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, November 1975.
- [Sys] SYSTAP, LLC. Bigdata. <http://www.systap.com/bigdata.htm>. Visited 05/2013.
- [TBMM04] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema Part 1: Structures Second Edition, W3C Recommendation 28 October 2004. <http://www.w3.org/TR/xmlschema-1/>, 2004. Visited 05/2013.
- [TEI07] TEI P5: Guidelines for Electronic Text Encoding and Interchange, 2007.
- [TLV03] Djamshid Tavangarian, Ulrike Lucke, and Denny Voigt. Multidimensional LearningObjects and Modular Lectures Markup Language. <http://www.ml-3.org/>, 2003. Visited 05/2013.
- [TvdAS09] Nikola Trcka, Wil M. P. van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In Pascal van Eck, Jaap Gordijn, and Roel Wieringa, editors, *CAiSE*, volume 5565 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2009.
- [vdA03] Wil M.P. van der Aalst. Challenges in business process management: Verification of business processes using petri nets. *Bulletin of the EATCS*, 80:174–198, 2003.
- [vDK83] Teun A. van Dijk and Walter Kintsch. *Strategies of Discourse Comprehension*. Academic Press, New York, 1983.
- [VRL+10] Maria Esther Vidal, Edna Ruckhaus, Tomas Lampo, Javier Sierra, Amadis Martinez, and Axel Polleres. On the Efficiency of Joining Group Patterns in SPARQL Queries. In *7th Extended Semantic Web Conference (ESWC2010)*, June 2010.
- [VTP08] Anne-Marie Vercoestre, James A. Thorn, and Jovan Peheceviski. Entity Ranking in Wikipedia. In *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC08)*, 2008.
- [Wal09] Norman Walsh. DocBook. <http://docs.oasis-open.org/docbook/specs/docbook-5.0-spec.html>, 2009. Visited 05/2013.

- [WBC⁺03] Michael Witbrock, David Baxter, Jon Curtis, Dave Schneider, Robert Kahlert, Pierluigi Miraglia, Peter Wagner, Kathy Panton, Gavin Matthews, and Amanda Vizedom. An interactive dialogue system for knowledge acquisition in cyc. In *In Proceedings of the Workshop on Mixed-Initiative Intelligent Systems. IJCAI*, pages 138–145, 2003.
- [Wei08] Franz Weigl. *Document Verification with Temporal Description Logics*. PhD thesis, University of Passau, 2008.
- [Wil68] John Wilkins. An essay towards a real character and a philosophical language, 1668. London.
- [Wil11] Martin Wilhelm. Semantische Rollengewichtung in Wikipedia-basierten Ontologien. Bachelor thesis, Chair for Information Management, University of Passau, 2011.
- [WJF09] F. Weigl, M. Jakšić, and B. Freitag. Towards the Automated Verification of Semi-structured Documents. *Journal of Data & Knowledge Engineering*, 68:292–317, 2009.
- [WSKR03] Kevin Wilkinson, Craig Sayers, Harumi Kuno, and Dave Reynolds. Efficient rdf storage and retrieval in jena2. Technical report, Hewlett-Packard, 2003.
- [WW08] Fei Wu and Daniel S. Weld. Automatically refining the Wikipedia infobox ontology. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 635–644, New York, NY, USA, 2008. ACM.
- [WWR04] WWR - Wissenswerkstatt Rechensysteme. <http://www.wwr-project.de>, 2004. Visited 05/2013.

List of Figures

3.1	RDF Graph	35
3.2	Temporal models	52
4.1	Left: Legal document with seal impression, Babylon, 414 BCE. Right: Seal and seal impression, Babylon, 5 th /6 th century BCE. Pergamon Museum, Berlin, Germany.	58
4.2	Fragment of a marble tablet with an Arabic line of text, Egypt, 10 th /11 th century CE. Pergamon Museum, Berlin, Germany.	59
4.3	Collection of books. Pražský hrad, Prague, Czech Republic.	61
4.4	Document abstraction layers	69
4.5	Illustration of the base document model from example 4.1.24	73
4.6	Document models	76
4.7	Illustration of the relations s (red) and p (blue) in various structural document models (directionality omitted for clarity)	79
4.8	Partial illustration of a semantic document model	84
4.9	Structure of musical notes	90
4.10	Example process: application for leave	94
5.1	Values for a variable <code>topic</code> in an environment \mathcal{E} at different positions in a document	104
5.2	Simple base document model to illustrate the processing order of media objects .	107
5.3	Metalayers of examples 5.3.5 and 5.3.9 (black arrowheads represent instantiation relationships, empty arrowheads represent generalisation relationships)	124
5.4	Metalayers of example 5.3.10 (black arrowheads represent instantiation relationships, empty arrowheads represent generalisation relationships, open arrowheads represent other relationships)	125
5.5	Metalayers and knowledge bases (black arrowheads represent instantiation relationships)	128
6.1	Wikipedia categories before (a) and after (b) extension	135
6.2	Database schema for Wikipedia	147
7.1	System architecture	156
7.2	Package overview	156
7.3	UML diagram: document model	158
7.4	UML diagram: background knowledge	160
7.5	UML diagram: document adapters (1/2)	161
7.6	UML diagram: document adapters (2/2)	162
7.7	UML diagram: preprocessing	164

7.8	UML diagram: semantic processing	166
7.9	UML diagram: postprocessing	167
7.10	UML diagram: verification model	169
8.1	Implementation options for reference relations	173
8.2	Implementation options for document versions (arrows represent has-part relationships)	177
9.1	Hierarchical source document and semantic document model	197
9.2	Flat source document and semantic document model	209
9.3	Chapter-level view on a semantic document model	220
9.4	Paragraph-level view on a semantic document model	220
10.1	VDK 1501 document. The primary reading path is shown in black, references from the <i>Table of Contents</i> are shown in blue, and references from the <i>Index</i> are shown in red.	230
10.2	VDK 1109 document. The primary reading path is shown in black, references to and from the <i>Table of Contents</i> are shown in blue, and cross references between sections are shown in green.	231
10.3	VDK 1108 document. The primary reading path is shown in black. Every section except the <i>Title Page</i> has a bidirectional reference to and from the <i>Table of Contents</i> (not shown).	232
10.4	Sample of a WWR e-learning document (German)	235
10.5	DLR document layout	237
10.6	DLR process model (excerpt)	238
10.7	Views on the DLR process model	239
11.1	Time measurements for the processing time of documents with a varying number of XML nodes	246
11.2	Time measurements for the processing time of documents with varying number of processing rules	247
11.3	Time measurements for the processing time of documents with varying number of data and fragment indicators in the background knowledge	248
11.4	Time measurements for the <i>total</i> processing time of documents from different domains	256
11.5	Time measurements for the <i>average</i> processing time of documents from different domains	257
11.6	Comparison of processing times for JBoss Drools and XQuery: obtaining a semantic document model from an HTML document	259
11.7	Comparison of processing times for JBoss Drools and XQuery: obtaining a semantic document model from a DocBook document	260
11.8	Comparison of processing times for SPARQL and XQuery: obtaining a verification model from a semantic document model (HTML)	261
11.9	Comparison of processing times for SPARQL and XQuery: obtaining a verification model from a semantic document model (DocBook)	262
11.10	Ontology inference scaled by number of individuals	265
11.11	Ontology inference scaled by number of concepts	266
11.12	Huge ontology retrieval times, scaled by number of concepts.	268
11.13	Huge ontology retrieval times, scaled by number of individuals per concept	269

<i>LIST OF FIGURES</i>	301
12.1 Quality of the extracted semantic document/process models	275
12.2 Incorrect and missing assertions in document models with modified background knowledge	278
12.3 F-measure for document models with modified background knowledge	279
A.1 Model vocabulary: object properties	312
A.2 Model vocabulary: datatype properties, coloured by namespace	313
A.3 Model vocabulary: classes	314
B.1 Process specification in Visio, courtesy of the Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH	339

List of Tables

3.1	\mathcal{ALC} tableau rules for an ABox \mathcal{A} , for complex concepts C_1 and C_2 , an atomic role R , and individuals p and p' , and a “new” individual q . p , p' and q are parameters.	32
3.2	Example: triple table	39
3.3	Example: property table	39
3.4	Example: class-property tables	40
3.5	Example: vertical partitioning	40
3.6	OWL constructs and their corresponding description logics expressions, with classes C_1 and C_2 , concepts C_1 and C_2 , properties r_1 and r_2 , roles r_1 and r_2 , objects i_1 and i_2 , and individuals i_1 and i_2 . $\langle \dots \rangle$ denotes a list.	43
3.7	Example: SKOS document.	45
6.1	Statistics for the Wikipedia categories from example 6.1.3, as of 2012-05-30.	134
6.2	Statistics for the Wikipedia articles from example 6.1.3, as of 2012-05-30.	134
6.3	Word pairs with their respective Adjusted Judged Synonymy (AJS), after [RG65].	144
6.4	Statistics for Wikipedia cores, as of 2012-05-30.	146
6.5	Database indices for Wikipedia	148
6.6	Time measurements for the Wikipedia extraction, as of 2012-05-30.	148
11.1	E-learning documents: XML document sizes (across 125 documents).	251
11.2	E-learning documents: document model sizes (across 125 document models).	251
11.3	E-learning documents: processing times in seconds (for 125 documents/document models).	251
11.4	Technical documentation documents: XML document sizes (across seven documents).	252
11.5	Technical documentation: document model sizes (across seven document models).	252
11.6	Technical documentation documents: processing times in seconds (for seven documents/document models).	253
11.7	Process description documents (NPB): XML document sizes (across four documents).	253
11.8	Process description documents (NPB): document model sizes (across four document models).	254
11.9	Process description documents (NPB): processing times in seconds (for four documents/document models).	254
11.10	Process description documents (DLR): XML document sizes (across 20 documents).	254
11.11	Process description documents (DLR): document model sizes (for one document model).	255

11.12	Process description documents (DLR): processing times in seconds (for 20 documents/one document model).	255
11.13	Adaptations to the processing rules for different domains.	263

Index

- ABox, 24
- \mathcal{AL} , 30
- \mathcal{ALC} , 31
- \mathcal{ALCCTL} , 54
- \mathcal{ALCCTL} concept, 54
- \mathcal{ALCCTL} formula, 54
- \mathcal{ALCCTL} model checking problem, 56
- \mathcal{ALCCTL} path, 55
- \mathcal{ALCCTL} temporal model, 55
- atomic concept, 22
- atomic content object, 70
- atomic entity, 71
- atomic role, 22
- Audio Video Interleave, 10
- augment operator, 80
- average, 250
- AVI, 10

- background knowledge, 98, 131
- base document model, 71
- base process model, 93
- blank node, 33

- Cascading Style Sheets, 9
- clash, 32
- class, 41
- class-property table, 39
- closed, 93
- closed world assumption, 29
- closest fragment induced by m , 100
- compatible, 85
- complete, 71
- complex concept, 22
- complex role, 22
- Computation Tree Logic, 51
- concept realisation problem, 29
- conclusion, 101
- conclusion of a transformation rule, 103
- concrete data type, 23
- concrete role, 23
- condition, 102

- confluent, 113
- connected, 75, 93
- connected base document model, 74, 75
- connected base process model, 92, 93
- consistent with respect to \mathcal{T} , 28
- content domain, 83
- content unit, 70
- control path, 93
- CSS, 9
- CTL, 51
- CTL formula, 51
- CTL model checking problem, 53
- CTL path, 52
- CTL temporal model, 51, 52

- data layer, 120
- data ontology, 95
- definition, 23
- description logic system, 24
- DITA, 9
- DocBook, 9
- Document Object Model, 18
- DOM, 18
- domain, 25
- domain knowledge, 131
- DTD, 17

- environment, 103
- ePub, 9
- equivalent, 98
- equivalent with respect to \mathcal{T} , 28
- evaluation context, 118
- evaluation of a graph selector against a model,
118
- expansion, 27

- fixed, 59
- Flash Video, 10
- fluid, 59
- fulfilled, 105

- Game of Life, 272
- global environment, 104
- graft operator, 80
- graph selector, 118
- group pattern, 47

- HTML, 9
- HTML5, 10
- hypermedia, 66
- hypertext, 57
- Hypertext Markup Language, 9
- hypervideo, 66

- iframe, 70
- include, 70
- individual, 22
- individual retrieval problem, 29
- inference service, 28
- instance assertion, 24
- interactive hypertext, 65
- interpretation, 25

- Javascript, 9

- keyword taxonomy, 98
- knowledge base, 25

- language literal, 33
- L^AT_EX, 9
- lexical database, 136
- link, 66
- links, 58
- LMML, 9
- local environment, 104
- logical consequence of \mathcal{T} and \mathcal{A} , 28

- Matroska, 10
- maximal, 120
- mean, 250
- media object, 71
- meta reference, 139
- metaconflict, 120
- metadata, 74
- metadata associated with B , 74
- metalayer, 120
- Microsoft Excel, 9
- Microsoft Powerpoint, 9
- Microsoft Visio, 9
- Microsoft Word, 9
- ML3, 9

- model, 27
- move operator, 80
- MPEG-2, 10
- MPEG-4, 10
- MPEG-7, 10

- named RDF resource, 33
- named style, 98
- namespace, 15
- navigation path, 66

- object, 41
- ontology, 24
- open world assumption, 29, 30
- operation, 103
- OWL, 40

- partial hypertext, 64
- path expression, 118
- PDF, 9
- Portable Document Format, 9
- premise, 101
- premise of a transformation rule, 102
- presentation domain, 83
- process, 92
- property, 41
- property table, 39
- prune operator, 80
- punning, 42, 121

- Quicktime, 10

- RDF, 33
- RDF literal, 33
- RDF resource, 33
- RDF Schema, 37
- RDF statement, 34
- RDF subgraph, 45
- RDF subgraph pattern, 45
- RDFS, 37
- reading path, 59, 75
- rendering instructions, 68
- Resource Description Framework, 33

- satisfiable with respect to \mathcal{T} , 28
- SAX, 18
- Scalable Vector Graphics, 10
- semantic document model, 82
- semantic process model, 93, 95
- semi-structured, 71

- semi-structured data, 71
- Server Side Includes, 70
- Simple API for XML, 18
- Simple Interactive Video Authoring Suite, 10
- Simple Knowledge Organization System, 43
- SIVA, 10
- SKOS, 43
- SMIL, 10
- social construction of technology, 60
- SPARQL, 45
- SPARQL Protocol And RDF Query Language, 45
- SPARQL query, 46
- SPARQL query result, 46
- specialisation, 23
- stop word, 137
- structural document model, 77
- structural layer, 68, 70
- structural ontology, 82, 95
- structural process model, 93
- structure indicators, 68
- style, 98
- style taxonomy, 99
- subsumed with respect to \mathcal{T} , 28
- successor relation, 70
- SVG, 10
- symbolic name, 23
- Synchronized Multimedia Integration Language, 10

- tableau, 32
- tableau algorithm, 32
- tableau algorithm for \mathcal{ACC} , 32
- taxonomy, 25
- TBox, 24
- technical layer, 68, 70
- template, 98
- terminological axiom, 23
- terminological ontology, 82
- Theora, 10
- three-layered document perspective, 71
- tower of meta, 120
- transformation rule, 101
- triple, 47
- triple table, 39
- typed literal, 33
- types of digital documents, 62

- unbound, 47
- unique name assumption, 26

- untyped literal, 33

- valid, 18
- valid in an interpretation $I_{\mathcal{O}}$, 27
- variable, 47
- vertical partitioning, 39
- view, 219
- visual layer, 68, 70

- Web Ontology Language, 40
- well-formed, 16
- Wikipedia, 133
- Wikipedia ontology, 145

- XML, 15
- XPath, 18
- XQuery, 20

Part VI

Appendix

Appendix A

Model Vocabulary

In this chapter, we will present the concepts and roles used in document models. We will show the vocabulary in the form of OWL/RDF properties and classes, including their namespace as used in our implementation.

The vocabulary is divided into three sets. The first set covers common concepts and roles that are used in most domains, such as **date**, **creator**, or **Fragment**. The second set covers the vocabulary typical for documents, such as **subtitle**, **Chapter**, or **Footnote**. The third set contains concepts and roles suitable for modelling processes, such as **input**, **manager**, or **Choice**.

For each property or class, we will give its name, its super-properties or super-classes (if any) in parentheses, and a brief description. Note that some properties are taken verbatim from the Dublin Core definition and are only shown here for the sake of completeness.

We will use the following namespace prefixes:

Prefix	URI
dc	http://purl.org/dc/elements/1.1/
vdk	http://www.verdikt.uni-passau.de#
data	http://www.verdikt.uni-passau.de/data#
ref	http://www.verdikt.uni-passau.de/reference#
src	http://www.verdikt.uni-passau.de/source#
did	http://www.verdikt.uni-passau.de/didactic#
eval	http://www.verdikt.uni-passau.de/evaluation#
ent	http://www.verdikt.uni-passau.de/entities#

An overview of the defined object properties (roles) can be seen in figure A.1. Figure A.2 shows the datatype properties (concrete roles), and figure A.3 shows the defined classes (concepts).

A.1 Common Vocabulary

A.1.1 Object Properties

ref:include (ref:part) Indicates that a fragment was included from an external source and integrated into the current fragment.

ref:part Indicates that the current fragment is a super-fragment of the other fragment.

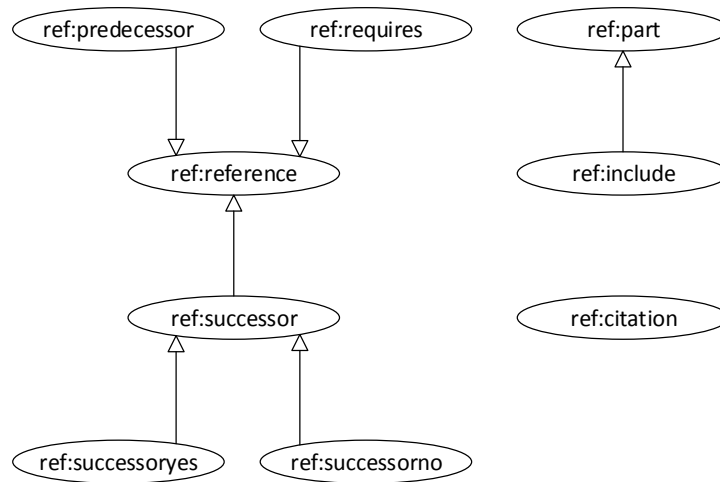


Figure A.1: Model vocabulary: object properties

ref:predecessor (ref:reference) Indicates that the current fragment is a successor of the other fragment.

ref:reference A general purpose reference from one fragment to another.

ref:requires (ref:reference) Indicates that the content of the current fragment depends on the content of the other fragment.

ref:successor (ref:reference) Indicates that the current fragment is a predecessor of the other fragment.

A.1.2 Datatype Properties

dc:contributor (ent:legalentity) An entity responsible for making contributions to the resource. Examples of a Contributor include a person, an organization, or a service. Typically, the name of a Contributor should be used to indicate the entity.¹

dc:coverage The spatial or temporal topic of the resource, the spatial applicability of the resource, or the jurisdiction under which the resource is relevant. Spatial topic and spatial applicability may be a named place or a location specified by its geographic coordinates. Temporal topic may be a named period, date, or date range. A jurisdiction may be a named administrative entity or a geographic place to which the resource applies. Recommended best practice is to use a controlled vocabulary such as the Thesaurus of Geographic Names [<http://www.getty.edu/research/tools/vocabulary/tgn/index.html>]. Where appropriate, named places or time periods can be used in preference to numeric identifiers such as sets of coordinates or date ranges.¹

¹All descriptions for properties and classes in the Dublin Core (dc) namespace have been taken verbatim from <http://dublincore.org/>.

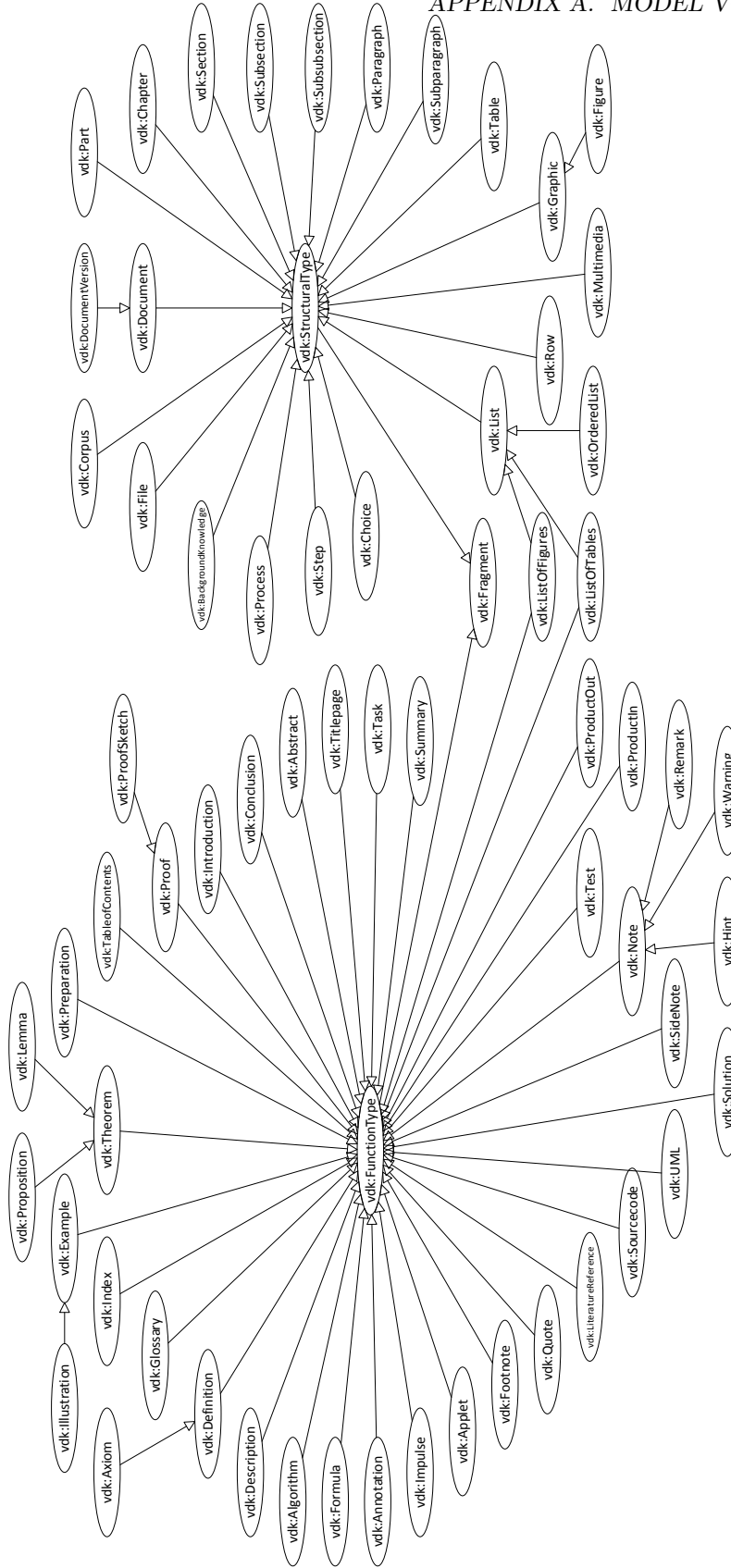


Figure A.3: Model vocabulary: classes

dc:creator (ent:legalentity) An entity primarily responsible for making the resource. Examples of a Creator include a person, an organization, or a service. Typically, the name of a Creator should be used to indicate the entity.¹

dc:date A point or period of time associated with an event in the lifecycle of the resource. Date may be used to express temporal information at any level of granularity. Recommended best practice is to use an encoding scheme, such as the W3CDTF profile of ISO 8601 [<http://www.w3.org/TR/NOTE-datetime>].¹

dc:description (data:content) An account of the resource. Description may include but is not limited to: an abstract, a table of contents, a graphical representation, or a free-text account of the resource.¹

dc:format The file format, physical medium, or dimensions of the resource. Examples of dimensions include size and duration. Recommended best practice is to use a controlled vocabulary such as the list of Internet Media Types [<http://www.iana.org/assignments/media-types/>].¹

dc:identifier An unambiguous reference to the resource within a given context. Recommended best practice is to identify the resource by means of a string conforming to a formal identification system.¹

dc:language A language of the resource. Recommended best practice is to use a controlled vocabulary such as RFC 4646 [<http://www.ietf.org/rfc/rfc4646.txt>].¹

dc:publisher (ent:legalentity) An entity responsible for making the resource available. Examples of a Publisher include a person, an organization, or a service. Typically, the name of a Publisher should be used to indicate the entity.¹

dc:relation A related resource. Recommended best practice is to identify the related resource by means of a string conforming to a formal identification system.¹

dc:rights Information about rights held in and over the resource. Typically, rights information includes a statement about various property rights associated with the resource, including intellectual property rights.¹

dc:source A related resource from which the described resource is derived. The described resource may be derived from the related resource in whole or in part. Recommended best practice is to identify the related resource by means of a string conforming to a formal identification system.¹

dc:subject (data:term) The topic of the resource. Typically, the subject will be represented using keywords, key phrases, or classification codes. Recommended best practice is to use a controlled vocabulary. To describe the spatial or temporal topic of the resource, use the Coverage element.¹

dc:title (data:topic) A name given to the resource.¹

dc:type The nature or genre of the resource. Recommended best practice is to use a controlled vocabulary such as the DCMI Type Vocabulary [<http://dublincore.org/documents/dcmi-type-vocabulary/>]. To describe the file format, physical medium, or dimensions of the resource, use the Format element.¹

vdk:id (dc:identifier) A corpus-wide unique id for a fragment.

vdk:initial Indicates if the current fragment is the first fragment in the default reading order for the resource. If there is more than one default reading order, multiple initial attributes can be specified. The default value for initial is false. Recommended practise is to put the initial attribute on the innermost initial fragment and optionally on its super-fragment up to but excluding the fragment representing the document.

vdk:position Indicates the position of the current fragment relative to the other sibling fragments that are also part of the current super-fragment. Recommended practise is to number sub-fragments starting with 1 and incrementing by 1, but this is not a requirement.

vdk:version Indicates the version of the current document. Recommended practise is to use a standardised numbering schema.

vdk:previousversion Indicates a previous version of the current document. Recommended practise is to use a standardised numbering schema.

data:abbreviation (data:term) Indicates that a term is used in an abbreviated form in the current fragment. Recommended practise is to use the term in its regular form, not in its abbreviated form as in the content of the fragment. The term should be used in a normalised grammatical form and spelling.

data:code Indicates a very small fragment of programming code. Recommended practise is to use one or two words, or the signature of a method at the most.

data:content Some extract or total of the content of the resource.

data:documentid (dc:identifier) A domain-dependent system-wide unique id for a document.

data:issue The issue of the current document. Recommended practise is to use a standardised numbering schema.

data:name (data:term) The name of an entity, a place, an event, or something similar. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

data:programminglanguage (data:term) The name of a programming language.

data:term A term, consisting of one or more words, in a normalised grammatical form and spelling. Recommended practise is to use the nominative singular for nouns, infinitive for verbs, and nominative singular masculine for adjectives. If a collection of knowledge about such terms is maintained, they should be identified by their preferred label.

data:topic (data:content) A topic, possibly in fragment or sentence form, of the resource.

did:duration (dc:date) The estimated duration required for working through a section of a document.

did:importance The didactical importance of a fragment in relation to the other fragments of the same resource. Recommended practise is to use a domain-wide standardised model of importance, or the high-medium-low model.

ent:approved (dc:contributor) An entity who has approved the current version of a document or process. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:author (dc:creator) An entity who has written or drafted the current version of a fragment. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:legality A legal entity such as a person or an organisation. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:location A location entity. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:organisation (ent:legality) An organisation as a legal entity. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:person (ent:legality) A person as a legal entity. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:prepared (dc:contributor) An entity who has prepared the current version of a document or process. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:released (dc:contributor) An entity who has released the current version of a document or process. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:reviewed (dc:contributor) An entity who has reviewed the current version of a document or process. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

eval:correctness (eval:truth) A truth value of a statement or fact, based on fuzzy theory. Values should be normalised to fall between 0 and 1.

eval:message An evaluation or error message. Messages like this are often added by the document parser or the extraction component.

eval:parseerrors Indicates the number of errors encountered while parsing the resource.

eval:probability (eval:truth) A truth value of a statement or fact, based on probability theory. Values should be normalised to fall between 0 and 1.

eval:quality (eval:truth) A truth value of a statement or fact, based on a custom quality measurement style. Values should be normalised to fall between 0 and 1.

eval:truth Indicates the truth value of a statement or fact. Recommended practise is to derive values for one or more of the sub-properties of truth (correctness, probability, quality) from actual measurements, and to derive the accumulated truth value using a standardised mathematical model. Values should be normalised to fall between 0 and 1.

ref:external (dc:relation) A reference to an external document or fragment. Usually specified as a URI/IRI.

src:change (dc:date) A date and time when the resource was changed.

src:checksum The checksum of the current version of the file containing the current fragment. Recommended practise is to use a standardised checksum method like MD5 or SHA-1.

src:filename (dc:source) The source filename of a document or fragment. Specified either as a system-dependent path or as a local URI.

src:idref An internal reference to a document id.

src:lastchange (src:change) The date and time of the last change to the resource.

src:uri (dc:source) The source URI of a document or fragment.

A.1.3 Classes

vdk:BackgroundKnowledge (vdk:StructuralType) A collection of background knowledge, integrated into the document model.

vdk:Corpus (vdk:StructuralType) A collection of resources like documents or processes.

vdk:Fragment Base class for all fragments.

vdk:FunctionType (vdk:Fragment) Base class for the function type of fragments.

vdk:List (vdk:StructuralType) A list.

vdk:Section (vdk:StructuralType) A generic section.

vdk:Sourcecode (vdk:FunctionType) A section of source code.

vdk:Step (vdk:StructuralType) A generic step.

vdk:StructuralType (vdk:Fragment) Base class for the structural type of fragments.

vdk:Table (vdk:StructuralType) A table.

vdk:UML (vdk:FunctionType) A UML diagram.

A.2 Vocabulary for Documents

A.2.1 Object Properties

ref:citation A reference to a literature reference.

A.2.2 Datatype Properties

data:columnheader (data:topic) The title of a table column.

data:rowheader (data:topic) The title of a table row.

data:subtitle (data:topic) A subtitle.

did:audience The intended audience of a document or fragment. Recommended practise is to use a domain-wide standardised vocabulary. If a collection of knowledge about such terms is maintained, they should be identified by their preferred label.

did:complexity The complexity of the content of a document or fragment. Recommended practise is to use a domain-wide standardised vocabulary. If a collection of knowledge about such terms is maintained, they should be identified by their preferred label.

did:medium (dc:format) The intended medium of a document. Recommended practise is to use a domain-wide standardised vocabulary. If a collection of knowledge about such terms is maintained, they should be identified by their preferred label.

did:objective (data:term) A learning objective for the current fragment in e-learning documents.

src:pagenumber The page number of the document containing the start of the current fragment.

A.2.3 Classes

vdk:Abstract (vdk:FunctionType) An abstract.

vdk:Algorithm (vdk:FunctionType) An algorithm.

- vdk:Annotation** (vdk:FunctionType) An annotation.
- vdk:Applet** (vdk:FunctionType) An applet.
- vdk:Axiom** (vdk:Definition) An axiom.
- vdk:Chapter** (vdk:StructuralType) A chapter.
- vdk:Conclusion** (vdk:FunctionType) A conclusion.
- vdk:Definition** (vdk:FunctionType) A formal definition.
- vdk:Description** (vdk:FunctionType) A description.
- vdk:Document** (vdk:StructuralType) A document.
- vdk:DocumentVersion** (vdk:Document) A version of a document.
- vdk:Example** (vdk:FunctionType) An example.
- vdk:Figure** (vdk:Graphic) A figure.
- vdk:File** (vdk:StructuralType) A file. This class should only be used if the represented file does not contain any other suitable root fragments.
- vdk:Footnote** (vdk:FunctionType) A footnote.
- vdk:Formula** (vdk:FunctionType) A formula.
- vdk:Glossary** (vdk:FunctionType) A glossary.
- vdk:Graphic** (vdk:StructuralType) A graphic.
- vdk:Hint** (vdk:Note) A hint.
- vdk:Illustration** (vdk:Example) An illustration.
- vdk:Impulse** (vdk:FunctionType) An impulse.
- vdk:Index** (vdk:FunctionType) An index.
- vdk:Introduction** (vdk:FunctionType) An introduction.
- vdk:Lemma** (vdk:Theorem) A lemma.
- vdk:ListOfFigures** (vdk:FunctionType, vdk:List) A list of figures.

vdk:ListOfTables (vdk:FunctionType, vdk:List) A list of tables.

vdk:LiteratureReference (vdk:FunctionType) A literature reference.

vdk:Multimedia (vdk:StructuralType) A multimedia element.

vdk:Note (vdk:FunctionType) A note.

vdk:OrderedList (vdk:List) An ordered list.

vdk:Paragraph (vdk:StructuralType) A paragraph.

vdk:Part (vdk:StructuralType) A part. This class should only be used for books that are split into different parts.

vdk:Preparation (vdk:FunctionType) A preparation step.

vdk:Proof (vdk:FunctionType) A formal proof.

vdk:ProofSketch (vdk:Proof) A proof sketch.

vdk:Proposition (vdk:Theorem) A proposition.

vdk:Quote (vdk:FunctionType) A quote.

vdk:Remark (vdk:Note) A remark.

vdk:Row (vdk:StructuralType) A row in a table.

vdk:SideNote (vdk:FunctionType) A side note. This class can also be used to represent excursions spanning entire sections or chapters.

vdk:Solution (vdk:FunctionType) A solution to a test.

vdk:Subparagraph (vdk:StructuralType) A subparagraph.

vdk:Subsection (vdk:StructuralType) A subsection.

vdk:Subsubsection (vdk:StructuralType) A sub-subsection.

vdk:Summary (vdk:FunctionType) A summary.

vdk:TableOfContents (vdk:FunctionType) A table of contents.

vdk:Task (vdk:FunctionType) A task in a test.

vdk:Test (vdk:FunctionType) A test.

vdk:Theorem (vdk:FunctionType) A theorem.

vdk:Titlepage (vdk:FunctionType) A titlepage.

vdk:Warning (vdk:Note) A warning.

A.3 Vocabulary for Processes

A.3.1 Object Properties

ref:successorno (ref:successor) A successor for a conditional negative.

ref:successories (ref:successor) A successor for a conditional positive.

A.3.2 Datatype Properties

data:dataflow Content related to the data flow of a process.

data:entityinput (data:input, data:product) Content related to the input data of an entity.

data:entityname (data:name) The name of an entity.

data:entityoutput (data:output, data:product) Content related to the output data of an entity.

data:entitypath (data:content) The path of an entity.

data:facilityname (data:name) The name of a facility.

data:facilitypath (data:content) The path a a facility.

data:input (data:dataflow) Content related to the input data of a process.

data:output (data:dataflow) Content related to the output data of a process.

data:processid (data:term) The id of a process.

data:processname (data:name) The name of a process.

data:activity (data:term) Content related to an activity of a process.

data:optionalactivity (data:activity) Content related to an optional activity of a process.

data:product Content related to the data of a process.

data:productinput (data:input, data:product) Content related to the input data of a process.

data:productoutput (data:output, data:product) Content related to the output data of a process.

ent:doesnotneedsubstitute (ent:person) A specialised domain-dependent property.

ent:manager (ent:person) A person who is a manager in the local context. The entity should be represented by its name. If a collection of knowledge about such entities is maintained, they should be identified by their preferred label.

ent:needssubstitute (ent:person) A specialised domain-dependent property.

ent:processmanager (ent:manager) The name of a process manager.

A.3.3 Classes

vdk:Choice (vdk:StructuralType) A process step involving a choice.

vdk:Process (vdk:StructuralType) A process.

vdk:ProductIn (vdk:FunctionType) A section representing the input of a product.

vdk:ProductOut (vdk:FunctionType) A section representing the output of a product.

Appendix B

Use Cases

In this chapter, we will show some excerpts of our implementation. We will present the actual Drools rules for obtaining semantic document models from base document models in DocBook format, as well as the required background knowledge. We will also show the SPARQL queries used to obtain a verification model from the semantic document model. For four different formats of process specifications, we will show both the Drools rules and the background knowledge.

Additionally, we will provide the code and some examples for implementing Conway's Game of Life in JBoss Drools.

B.1 Technical documentations in DocBook format

Drools extraction rules

```
1 package de.uni_passau.verdikt.DocumentModel.adapter.implementation;
2
3 import de.uni_passau.verdikt.DocumentModel.*;
4 import de.uni_passau.verdikt.DocumentModel.BackgroundKnowledge.*;
5 import org.w3c.dom.*;
6 import java.util.*;
7
8 function void annotate(Fragment fragment, String name, Object value,
9     DroolsGlobalContext context) {
10     if (name == null)
11         return;
12     if (name.equals(""))
13         return;
14     if (value == null)
15         return;
16     fragment.addAnnotation(context.createDataAnnotation(name, value));
17 }
18
19 function List getChildElementsRecursive(Element node) {
20     List result = new ArrayList();
21     NodeList children = node.getChildNodes();
22     for (int i=children.getLength()-1;i>=0;i--) {
23         if (children.item(i) instanceof Element) {
24             Element child = (Element) children.item(i);
25             result.addAll(getChildElementsRecursive(child));
26             result.add(child);
27         }
28     }
29 }
```

```

28     return result;
29 }
30
31 function List getChildElements(DroolsLocalContext lcontext) {
32     return getChildElementsRecursive(lcontext.getNode());
33 }
34
35 function String getIndicators(DroolsLocalContext lcontext) {
36     // adapted and simplified
37     return lcontext.getNode().getLocalName();
38 }
39
40 function String getId(DroolsLocalContext lcontext, DroolsGlobalContext context) {
41     // adapted
42     if (lcontext.getNode().hasAttribute("id"))
43         return lcontext.getNode().getAttribute("id");
44     else
45         return context.generateId(lcontext);
46 }
47
48 rule "Element Recursion"
49 salience 20
50 when
51     $context: DroolsGlobalContext()
52     $lcontext: DroolsLocalContext(parent == null && recursed == false)
53 then
54     for (Object $child: getChildElements($lcontext)) {
55         DroolsLocalContext $childcontext = $context.createLocalContext();
56         $childcontext.setNode((Element) $child);
57         $childcontext.setParent($lcontext);
58         insert($childcontext);
59     }
60     modify($lcontext) {
61         setRecursed(true);
62     }
63     // removed: not necessary (only one file)
64     // Fragment $fragment = $context.createFragment($lcontext);
65     // $context.getModel().getRoot().addChild($fragment);
66 end
67
68 // removed: not necessary (only one file)
69 // rule "Reference Insertion" ...
70
71 rule "Data Annotation"
72 when
73     $context: DroolsGlobalContext()
74     $lcontext: DroolsLocalContext($context.knowledgebases["dataIndicators"].
75         termGroups["indicators"] contains node.localName)
76 then
77     List<DataMapping> $mappings = $context.getMappings("data");
78     for (DataMapping $mapping: $mappings) {
79         // adapted: multiple values
80         for (String $value: $context.evaluateXPathValues($mapping.getSource(),
81             $lcontext.getNode())) {
82             if (!$value.equals("")) {
83                 annotate($context.getStack().peek(), $mapping.getTarget(), $value,
84                     $context);
85             }
86         }
87     }
88 }
89 }
90 end
91

```



```

87 rule "Reference"
88 when
89     $context: DroolsGlobalContext()
90     $lcontext: DroolsLocalContext($context.knowledgebases["referenceIndicators"].
        termGroups["indicators"] contains node.localName)
91 then
92     List<DataMapping> $mappings = $context.getMappings("reference");
93     for (DataMapping $mapping: $mappings) {
94         String $value = $context.evaluateXPathValue($mapping.getSource(),
            $lcontext.getNode());
95         if (!$value.equals("")) {
96             annotate($context.getStack().peek(), $mapping.getTarget(), $context.
                createFragment($value), $context);
97         }
98     }
99 end
100
101 rule "Fragment"
102 when
103     $context: DroolsGlobalContext()
104     $lcontext: DroolsLocalContext($context.knowledgebases["fragmentIndicators"].
        termGroups["indicators"] contains node.localName)
105 then
106     Fragment $fragment = $context.createFragment(getId($lcontext, $context));
107     String $ind = getIndicators($lcontext);
108     List<DataMapping> $mappings = $context.getMappings("styleFragment");
109     for (DataMapping $mapping: $mappings) {
110         if ($ind.equals($mapping.getSource())) {
111             $fragment.addStructuralType($mapping.getTarget());
112         }
113     }
114     //removed: not necessary (no keywords)
115     // $mappings = $context.getMappings("keywordFragment");
116     $lcontext.setFragment($fragment);
117     Assertions $kb = (Assertions) $context.getKnowledgebases().get("
        documentStructure");
118     boolean changed = true;
119     while (changed) {
120         changed = false;
121         for (OntologyClass $newtype: $fragment.getStructuralTypes()) {
122             for (OntologyClass $stacktype: $context.getStack().peek().
                getStructuralTypes()) {
123                 if ($kb.getBroaderSelf($stacktype.getURI()).contains($newtype.
                    getURI())) {
124                     $context.getStack().pop();
125                     changed = true;
126                 }
127             }
128         }
129     }
130     // added: mark the first chapter
131     if ($fragment.hasStructuralType("vdk:Chapter"))
132         annotate($fragment, "vdk:initial", $context.initial(), $context);
133     $context.getStack().peek().addChild($fragment);
134     $context.getStack().push($fragment);
135 end
136
137 /* removed: not necessary (only one file)
138 * rule "File Root" */

```

Background knowledge: document structure

```

1 <assertions>
2   <roleAssertion role="hasNarrower"
3     left="vdk:Chapter" right="vdk:Section"/>
4   <roleAssertion role="hasNarrower"
5     left="vdk:Section" right="vdk:Paragraph"/>
6   <roleAssertion role="hasNarrower"
7     left="vdk:Section" right="vdk:Table"/>
8   <roleAssertion role="hasNarrower"
9     left="vdk:SideNote" right="vdk:Paragraph"/>
10  <roleAssertion role="hasNarrower"
11    left="vdk:Section" right="vdk:List"/>
12  <roleAssertion role="hasNarrower"
13    left="vdk:List" right="vdk:Paragraph"/>
14 </assertions>

```

Background knowledge: fragment indicators chapter, section, sect1, sect2, sect3, sect4, para, appendix, glossary, preface, itemizedlist, orderedlist, programlisting, example, table

Background knowledge: style fragment mappings

chapter	→	vdk:Chapter
section	→	vdk:Section
sect1	→	vdk:Section
sect2	→	vdk:Section
sect3	→	vdk:Section
sect4	→	vdk:Section
para	→	vdk:Paragraph
appendix	→	vdk:Chapter
appendix	→	vdk:Appendix
glossary	→	vdk:Chapter
glossary	→	vdk:Glossary
preface	→	vdk:Chapter
itemizedlist	→	vdk:List
orderedlist	→	vdk:List
programlisting	→	vdk:Sourcecode
programlisting	→	vdk:Paragraph
example	→	vdk:Example
example	→	vdk:Paragraph
table	→	vdk:Table

Background knowledge: data indicators chapter, section, sect1, sect2, sect3, sect4, para, appendix, glossary, preface, itemizedlist, orderedlist, example, table

Background knowledge: data mappings

./title/text()	→	dc:title
self::node()[local-name()='para']//command/text()	→	data:command
self::node()[local-name()='para']//primary/text()	→	data:term
self::node()[local-name()='para']//firstterm/text()	→	data:term
self::node()[local-name()='para']//acronym/text()	→	data:abbreviation
self::node()[local-name()='glossary']//glossterm/text()	→	data:term

Background knowledge: reference indicators xref

Background knowledge: reference mappings

./@linkend → ref:reference

SPARQL queries for verification model: select states

```
1 SELECT ?s
2 WHERE { ?s rdf:type vdk:Chapter }
```

SPARQL queries for verification model: select starting states

```
1 SELECT ?s
2 WHERE {
3   ?s vdk:id "$state" .
4   { ?s doc:initial "true" }
5   UNION {
6     { ?s reference:part ?p }
7     UNION
8     { ?s reference:part ?v0 . ?v0 reference:part ?p }
9     UNION
10    { ?s reference:part ?v1 . ?v1 reference:part ?v2 . ?v2 reference:part ?p }
11    UNION
12    { ?s reference:part ?v3 . ?v3 reference:part ?v4 . ?v4 reference:part ?v5
13      . ?v5 reference:part ?p } .
14  } UNION {
15    { ?p reference:part ?s }
16    UNION
17    { ?p reference:part ?v6 . ?v6 reference:part ?s }
18    UNION
19    { ?p reference:part ?v7 . ?v7 reference:part ?v8 . ?v8 reference:part ?s }
20    UNION
21    { ?p reference:part ?v9 . ?v9 reference:part ?v10 . ?v10 reference:part ?
22      v11 . ?v11 reference:part ?s } .
23  }
24 }
```

SPARQL queries for verification model: select successors for states

```
1 SELECT ?s
2 WHERE {
3   ?x vdk:id "$state" .
4   { ?x reference:part ?p }
5   UNION
6   { ?x reference:part ?v0 . ?v0 reference:part ?p }
7   UNION
8   { ?x reference:part ?v1 . ?v1 reference:part ?v2 . ?v2 reference:part ?p }
9   UNION
10  { ?x reference:part ?v3 . ?v3 reference:part ?v4 . ?v4 reference:part ?v5 . ?
11    v5 reference:part ?p } .
12  { ?x reference:related ?r }
13  UNION
14  { ?x reference:reference ?r }
```

```

14 UNION
15 { ?p reference:related ?r }
16 UNION
17 { ?p reference:reference ?r } .
18 { ?x reference:related ?s }
19 UNION
20 { ?x reference:reference ?s }
21 UNION
22 { ?p reference:related ?s }
23 UNION
24 { ?p reference:reference ?s }
25 UNION {
26   { ?r reference:part ?s }
27   UNION
28   { ?r reference:part ?v6 . ?v6 reference:part ?s }
29   UNION
30   { ?r reference:part ?v7 . ?v7 reference:part ?v8 . ?v8 reference:part ?s }
31   UNION
32   { ?r reference:part ?v9 . ?v9 reference:part ?v10 . ?v10 reference:part ?
      v11 . ?v11 reference:part ?s }
33 } UNION {
34   { ?s reference:part ?r }
35   UNION
36   { ?s reference:part ?v12 . ?v12 reference:part ?r }
37   UNION
38   { ?s reference:part ?v13 . ?v13 reference:part ?v14 . ?v14 reference:part
      ?r }
39   UNION
40   { ?s reference:part ?v15 . ?v15 reference:part ?v16 . ?v16 reference:part
      ?v17 . ?v17 reference:part ?r }
41 } .
42 ?s rdf:type vdk:Chapter
43 }

```

SPARQL queries for verification model: select values for the concept “term”

```

1 SELECT ?v
2 WHERE {
3   ?x vdk:id "$state" .
4   {
5     { ?x reference:part ?p }
6     UNION
7     { ?x reference:part ?v0 . ?v0 reference:part ?p }
8     UNION
9     { ?x reference:part ?v1 . ?v1 reference:part ?v2 . ?v2 reference:part ?p }
10    UNION
11    { ?x reference:part ?v3 . ?v3 reference:part ?v4 . ?v4 reference:part ?v5
      . ?v5 reference:part ?p }
12    UNION
13    { ?x reference:part ?v6 . ?v6 reference:part ?v7 . ?v7 reference:part ?v8
      . ?v8 reference:part ?v9 . ?v9 reference:part ?p }
14    UNION
15    { ?x reference:part ?v10 . ?v10 reference:part ?v11 . ?v11 reference:part
      ?v12 . ?v12 reference:part ?v13 . ?v13 reference:part ?v14 . ?v14
      reference:part ?p } .
16   ?p data:term ?v
17 } UNION {
18   ?x data:term ?v
19 } UNION {

```

```

20     { ?p reference:part ?x }
21     UNION
22     { ?p reference:part ?v15 . ?v15 reference:part ?x }
23     UNION
24     { ?p reference:part ?v16 . ?v16 reference:part ?v17 . ?v17 reference:part
25       ?x }
26     UNION
27     { ?p reference:part ?v18 . ?v18 reference:part ?v19 . ?v19 reference:part
28       ?v20 . ?v20 reference:part ?x } .
29   }

```

SPARQL queries for verification model: select values for the concept “glossary term”

```

1 SELECT ?v
2 WHERE {
3   ?x vdk:id "$state" .
4   ?x rdf:type vdk:Glossary .
5   {
6     { ?x reference:part ?p }
7     UNION
8     { ?x reference:part ?v0 . ?v0 reference:part ?p }
9     UNION
10    { ?x reference:part ?v1 . ?v1 reference:part ?v2 . ?v2 reference:part ?p }
11    UNION
12    { ?x reference:part ?v3 . ?v3 reference:part ?v4 . ?v4 reference:part ?v5
13      . ?v5 reference:part ?p }
14    UNION
15    { ?x reference:part ?v6 . ?v6 reference:part ?v7 . ?v7 reference:part ?v8
16      . ?v8 reference:part ?v9 . ?v9 reference:part ?p }
17    UNION
18    { ?x reference:part ?v10 . ?v10 reference:part ?v11 . ?v11 reference:part
19      ?v12 . ?v12 reference:part ?v13 . ?v13 reference:part ?v14 . ?v14
20      reference:part ?p } .
21    ?p data:term ?v
22  } UNION {
23    ?x data:term ?v
24  } UNION {
25    { ?p reference:part ?x }
26    UNION
27    { ?p reference:part ?v15 . ?v15 reference:part ?x }
28    UNION
29    { ?p reference:part ?v16 . ?v16 reference:part ?v17 . ?v17 reference:part
30      ?x }
31    UNION
32    { ?p reference:part ?v18 . ?v18 reference:part ?v19 . ?v19 reference:part
33      ?v20 . ?v20 reference:part ?x } .
34    ?p data:term ?v
35  }

```

SPARQL queries for verification model: select values for the concept “source”

```

1 SELECT ?v

```

```
2 WHERE { ?x vdk:id "$state" . ?x source:idref ?v }
```

B.2 Process specifications in BPMN format (NPB)

Drools extraction rules

```
1 package de.uni_passau.verdikt.DocumentModel.adapter.implementation;
2
3 import de.uni_passau.verdikt.DocumentModel.*;
4 import de.uni_passau.verdikt.DocumentModel.BackgroundKnowledge.*;
5 import org.w3c.dom.*
6 import java.util.*;
7
8 function void annotate(Fragment fragment, String name, Object value,
9     DroolsGlobalContext context) {
10     if (name == null)
11         return;
12     if (name.equals(""))
13         return;
14     if (value == null)
15         return;
16     fragment.addAnnotation(context.createDataAnnotation(name, value));
17 }
18
19 function List getChildElementsRecursive(Element node) {
20     List result = new ArrayList();
21     NodeList children = node.getChildNodes();
22     for (int i=children.getLength()-1;i>=0;i--) {
23         if (children.item(i) instanceof Element) {
24             Element child = (Element) children.item(i);
25             result.addAll(getChildElementsRecursive(child));
26             result.add(child);
27         }
28     }
29     return result;
30 }
31
32 function List getChildElements(DroolsLocalContext lcontext) {
33     return getChildElementsRecursive(lcontext.getNode());
34 }
35
36 function String getIndicators(DroolsLocalContext lcontext) {
37     // adapted and simplified
38     return lcontext.getNode().getLocalName();
39 }
40
41 function String getId(DroolsLocalContext lcontext, DroolsGlobalContext context) {
42     // simplified
43     return lcontext.getNode().getAttribute("id");
44 }
45
46 rule "Element Recursion"
47 salience 20
48 when
49     $context: DroolsGlobalContext()
50     $lcontext: DroolsLocalContext(parent == null && recursed == false)
51 then
52     for (Object $child: getChildElements($lcontext)) {
```

```

52     DroolsLocalContext $childcontext = $context.createLocalContext();
53     $childcontext.setNode((Element) $child);
54     $childcontext.setParent($lcontext);
55     insert($childcontext);
56 }
57 modify($lcontext) {
58     setReursed(true);
59 }
60 // removed: not necessary (only one file)
61 // Fragment $fragment = $context.createFragment($lcontext);
62 // $context.getModel().getRoot().addChild($fragment);
63 end
64
65 // removed: not necessary (only one file)
66 // rule "Reference Insertion" ...
67
68 rule "Data Annotation"
69 when
70     $context: DroolsGlobalContext()
71     $lcontext: DroolsLocalContext($context.knowledgebases["dataIndicators"].
        termGroups["indicators"] contains node.localName)
72 then
73     List<DataMapping> $mappings = $context.getMappings("data");
74     for (DataMapping $mapping: $mappings) {
75         String $value = $context.evaluateXPathValue($mapping.getSource(),
            $lcontext.getNode()).replaceAll("\\s+", " ").replace(" - ", " -- ").
            replace("- ", "");
76         if (!$value.equals("")) {
77             annotate($context.getStack().peek(), $mapping.getTarget(), $value,
                $context);
78         }
79     }
80 end
81
82 rule "Reference"
83 when
84     $context: DroolsGlobalContext()
85     $lcontext: DroolsLocalContext($context.knowledgebases["referenceIndicators"].
        termGroups["indicators"] contains node.localName)
86 then
87     // adapted and simplified: references are not modelled as part
88     // of the source, but as separate entities that hold the IDs
89     // of both the source and the target
90     Fragment $source = $context.createFragment($lcontext.getNode().getAttribute("
        sourceRef"));
91     Fragment $target = $context.createFragment($lcontext.getNode().getAttribute("
        targetRef"));
92     annotate($source, "ref:reference", $target, $context);
93 end
94
95 rule "Fragment"
96 when
97     $context: DroolsGlobalContext()
98     $lcontext: DroolsLocalContext($context.knowledgebases["fragmentIndicators"].
        termGroups["indicators"] contains node.localName)
99 then
100     Fragment $fragment = $context.createFragment(getId($lcontext, $context));
101     String $ind = getIndicators($lcontext);
102     List<DataMapping> $mappings = $context.getMappings("styleFragment");
103     for (DataMapping $mapping: $mappings) {
104         if ($ind.equals($mapping.getSource())) {
105             $fragment.addStructuralType($mapping.getTarget());

```

```

106     }
107 }
108 // adapted: other text content source (attribute instead of text content)
109 String $text = $lcontext.getNode().getAttribute("name");
110 $mappings = $context.getMappings("keywordFragment");
111 for (DataMapping $mapping: $mappings) {
112     if ($text.contains($mapping.getSource())) {
113         $fragment.addStructuralType($mapping.getTarget());
114     }
115 }
116 $lcontext.setFragment($fragment);
117 Assertions $kb = (Assertions) $context.getKnowledgebases().get("
    documentStructure");
118 boolean changed = true;
119 while (changed) {
120     changed = false;
121     for (OntologyClass $newtype: $fragment.getStructuralTypes()) {
122         for (OntologyClass $stacktype: $context.getStack().peek().
            getStructuralTypes()) {
123             if ($kb.getBroaderSelf($stacktype.getURI()).contains($newtype.
                getURI())) {
124                 $context.getStack().pop();
125                 changed = true;
126             }
127         }
128     }
129 }
130 $context.getStack().peek().addChild($fragment);
131 $context.getStack().push($fragment);
132 end
133
134 /* removed: not necessary (only one file)
135 * rule "File Root" */

```

Background knowledge: fragment indicators other, task, exclusiveGateway, otherEvent

Background knowledge: style fragment mappings

other	→	vdk:Step
task	→	vdk:Task
task	→	vdk:Step
exclusiveGateway	→	vdk:Choice
exclusiveGateway	→	vdk:Step
otherEvent	→	vdk:Event
otherEvent	→	vdk:Step

Background knowledge: data indicators other, task, otherEvent

Background knowledge: data mappings

./@name	→	vdk:name
---------	---	----------

Background knowledge: keyword mappings

(Vorlage)	→	vdk:Document
-----------	---	--------------

Background knowledge: reference indicators sequenceFlow

B.3 Process specifications in Visio format (NPB)

Drools extraction rules

```

1 package de.uni_passau.verdikt.DocumentModel.adapter.implementation;
2
3 import de.uni_passau.verdikt.DocumentModel.*;
4 import de.uni_passau.verdikt.DocumentModel.BackgroundKnowledge.*;
5 import org.w3c.dom.*;
6 import java.util.*;
7
8 function void annotate(Fragment fragment, String name, Object value,
9     DroolsGlobalContext context) {
10     if (name == null)
11         return;
12     if (name.equals(""))
13         return;
14     if (value == null)
15         return;
16     fragment.addAnnotation(context.createDataAnnotation(name, value));
17 }
18
19 function List getChildElementsRecursive(Element node) {
20     List result = new ArrayList();
21     NodeList children = node.getChildNodes();
22     for (int i=children.getLength()-1;i>=0;i--) {
23         if (children.item(i) instanceof Element) {
24             Element child = (Element) children.item(i);
25             result.addAll(getChildElementsRecursive(child));
26             result.add(child);
27         }
28     }
29     return result;
30 }
31
32 function List getChildElements(DroolsLocalContext lcontext) {
33     return getChildElementsRecursive(lcontext.getNode());
34 }
35
36 function String getIndicators(DroolsLocalContext lcontext) {
37     // adapted
38     return lcontext.getNode().getLocalName() + "_" + lcontext.getNode().
39         getAttribute("Master");
40 }
41
42 function int getId(DroolsLocalContext lcontext, DroolsGlobalContext context) {
43     // simplified
44     return Integer.parseInt(lcontext.getNode().getAttribute("ID"));
45 }
46
47 rule "Element Recursion"
48 salience 20
49 when
50     $context: DroolsGlobalContext()
51     $lcontext: DroolsLocalContext(parent == null && recursed == false)
52 then
53     for (Object $child: getChildElements($lcontext)) {
54         DroolsLocalContext $childcontext = $context.createLocalContext();
55         $childcontext.setNode((Element) $child);
56     }
57 }

```

```

54     $childcontext.setParent($lcontext);
55     insert($childcontext);
56 }
57 modify($lcontext) {
58     setRecurse(true);
59 }
60 // removed: not necessary (only one file)
61 // Fragment $fragment = $context.createFragment($lcontext);
62 // $context.getModel().getRoot().addChild($fragment);
63 end
64
65 // removed: not necessary (only one file)
66 // rule "Reference Insertion" ...
67
68 rule "Data Annotation"
69 when
70     $context: DroolsGlobalContext()
71     // adapted: split indicators
72     $lcontext: DroolsLocalContext($context.knowledgebases["dataIndicators"].
73         termGroups["indicators"] contains node.localName &&
74         $context.knowledgebases["dataIndicators2"].termGroups["indicators"]
75         contains attributes["Master"])
76 then
77     List<DataMapping> $mappings = $context.getMappings("data");
78     for (DataMapping $mapping: $mappings) {
79         String $value = $context.evaluateXPathValue($mapping.getSource(),
80             $lcontext.getNode()).replaceAll("\\s+", " ").replace(" - ", " -- ").
81             replace("-", "").trim();
82         if (!$value.equals("")) {
83             annotate($context.getStack().peek(), $mapping.getTarget(), $value,
84                 $context);
85         }
86     }
87 }
88 end
89
90 rule "Reference"
91 when
92     $context: DroolsGlobalContext()
93     $lcontext: DroolsLocalContext($context.knowledgebases["referenceIndicators"].
94         termGroups["indicators"] contains node.localName)
95 then
96     // adapted: references are modelled as separate entities that hold the IDs
97     // of both the source and the target, where either the source or the target
98     // is a connector (e.g., an arrow) and needs to be bypassed (because arrows
99     // are not part of the actual process model, only graphical objects)
100     if ($lcontext.getNode().getAttribute("FromCell").equals("BeginX")) {
101         String $rname = "ref_" + $lcontext.getNode().getAttribute("FromSheet");
102         Fragment $source = $context.createFragment(Integer.parseInt($lcontext.
103             getNode().getAttribute("ToSheet")));
104         Fragment $target = (Fragment) $context.getValue($rname);
105         if ($target == null) {
106             $context.setValue($rname, $source);
107         } else if (!$source.getStructuralTypes().isEmpty() && !$target.
108             getStructuralTypes().isEmpty()) {
109             annotate($source, "ref:reference", $target, $context);
110         }
111     }
112     } else if ($lcontext.getNode().getAttribute("FromCell").equals("EndX")) {
113         String $rname = "ref_" + $lcontext.getNode().getAttribute("FromSheet");
114         Fragment $target = $context.createFragment(Integer.parseInt($lcontext.
115             getNode().getAttribute("ToSheet")));
116         Fragment $source = (Fragment) $context.getValue($rname);
117         if ($source == null) {

```

```

107     $context.setValue($rname, $target);
108   } else if (!$source.getStructuralTypes().isEmpty() && !$target.
109     getStructuralTypes().isEmpty()) {
110     annotate($source, "ref:reference", $target, $context);
111   }
112 end
113
114 rule "Fragment"
115 when
116   $context: DroolsGlobalContext()
117   // adapted: split indicators
118   $lcontext: DroolsLocalContext($context.knowledgebases["fragmentIndicators"].
119     termGroups["indicators"] contains node.localName &&
120     $context.knowledgebases["fragmentIndicators2"].termGroups["indicators"
121     ] contains attributes["Master"])
122 then
123   Fragment $fragment = $context.createFragment(getId($lcontext, $context));
124   String $ind = getIndicators($lcontext);
125   List<DataMapping> $mappings = $context.getMappings("styleFragment");
126   for (DataMapping $mapping: $mappings) {
127     if ($ind.equals($mapping.getSource())) {
128       $fragment.addStructuralType($mapping.getTarget());
129     }
130   }
131   //removed: not necessary (no keywords)
132   // $mappings = $context.getMappings("keywordFragment");
133   $lcontext.setFragment($fragment);
134   Assertions $kb = (Assertions) $context.getKnowledgebases().get("
135     documentStructure");
136   boolean changed = true;
137   while (changed) {
138     changed = false;
139     for (OntologyClass $newtype: $fragment.getStructuralTypes()) {
140       for (OntologyClass $stacktype: $context.getStack().peek().
141         getStructuralTypes()) {
142         if ($kb.getBroaderSelf($stacktype.getURI()).contains($newtype.
143           getURI())) {
144           $context.getStack().pop();
145           changed = true;
146         }
147       }
148     }
149   }
150   $context.getStack().peek().addChild($fragment);
151   $context.getStack().push($fragment);
152 end
153
154 /* removed: not necessary (only one file)
155 * rule "File Root" */

```

Background knowledge: fragment indicators (1/2) Shape

Background knowledge: fragment indicators (2/2) 11, 12, 14, 15, 16, 17, 21

Background knowledge: style fragment mappings

```

Shape.11 → vdk:Start
Shape.11 → vdk:Step
Shape.12 → vdk:Task
Shape.12 → vdk:Step
Shape.14 → vdk:Event
Shape.14 → vdk:Step
Shape.15 → vdk:Choice
Shape.15 → vdk:Step
Shape.16 → vdk:Database
Shape.16 → vdk:Step
Shape.17 → vdk:File
Shape.17 → vdk:Step
Shape.21 → vdk:End
Shape.21 → vdk:Step

```

Background knowledge: data indicators (1/2) Shape

Background knowledge: data indicators (2/2) 11, 12, 14, 15, 16, 17, 21

Background knowledge: data mappings

```
concat(./Text/text()[1],./Text/text()[2],./Text/text()[3]) → vdk:name
```

Background knowledge: reference indicators Connect

B.4 Process specifications in XML format (NPB)

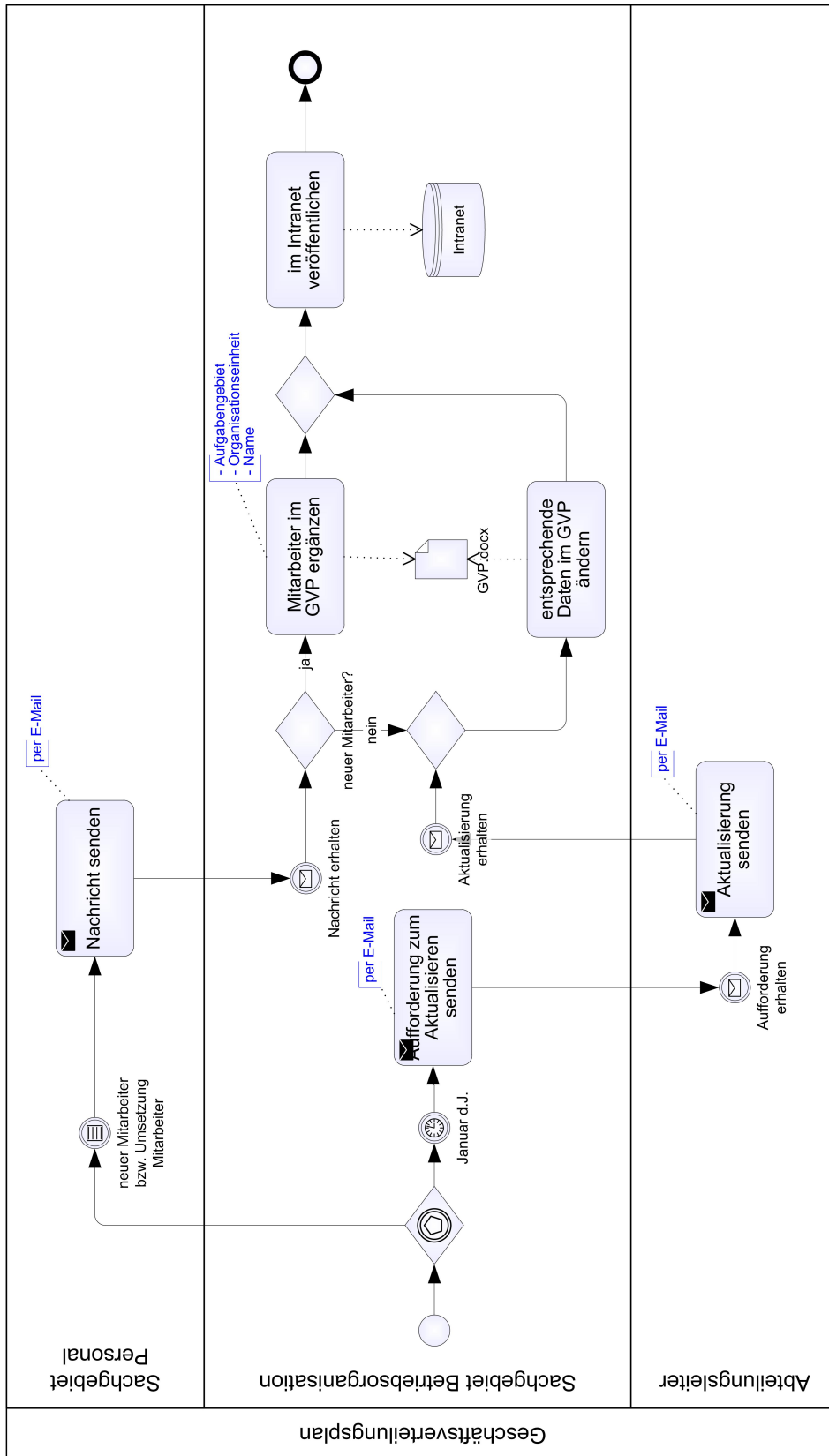
Figure B.1 shows an illustration of a process specified in Visio. This is the process that we used in the evaluation.

Drools extraction rules

```

1 package de.uni_passau.verdikt.DocumentModel.adapter.implementation;
2
3 import de.uni_passau.verdikt.DocumentModel.*;
4 import de.uni_passau.verdikt.DocumentModel.BackgroundKnowledge.*;
5 import org.w3c.dom.*
6 import java.util.*;
7
8 function void annotate(Fragment fragment, String name, Object value,
9     DroolsGlobalContext context) {
10     if (name == null)
11         return;
12     if (name.equals(""))
13         return;
14     if (value == null)
15         return;
16     fragment.addAnnotation(context.createDataAnnotation(name, value));
17 }
18
19 function List getChildElementsRecursive(Element node) {
20     List result = new ArrayList();
21     NodeList children = node.getChildNodes();
22     for (int i=children.getLength()-1;i>=0;i--) {

```



© DVZ M-V GmbH


 DVZ Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH		erstellt: 24.01.2013 11:18:54 geändert: 24.01.2013 16:13:21
Autor: prozessmanagement@dvz-mv.de Version: 1.0 Status: freigegeben	Geschäftsverteilungsplan_vsd	
Geschäftsverteilungsplan Geschäftsverteilungsplan LeiKa-Nr.: 99144005000000		

Figure B.1: Process specification in Visio, courtesy of the Datenverarbeitungszentrum Mecklenburg-Vorpommern GmbH

```

22     if (children.item(i) instanceof Element) {
23         Element child = (Element) children.item(i);
24         result.addAll(getChildElementsRecursive(child));
25         result.add(child);
26     }
27 }
28 return result;
29 }
30
31 function List getChildElements(DroolsLocalContext lcontext) {
32     return getChildElementsRecursive(lcontext.getNode());
33 }
34
35 function String getIndicators(DroolsLocalContext lcontext) {
36     // adapted and simplified
37     return lcontext.getNode().getLocalName();
38 }
39
40 function String getId(DroolsLocalContext lcontext, DroolsGlobalContext context) {
41     // adapted: id or blockId attribute
42     if (lcontext.getNode().hasAttribute("blockId"))
43         return lcontext.getNode().getAttribute("blockId");
44     else
45         return lcontext.getNode().getAttribute("id");
46 }
47
48 rule "Element Recursion"
49 salience 20
50 when
51     $context: DroolsGlobalContext()
52     $lcontext: DroolsLocalContext(parent == null && recursed == false)
53 then
54     for (Object $child: getChildElements($lcontext)) {
55         DroolsLocalContext $childcontext = $context.createLocalContext();
56         $childcontext.setNode((Element) $child);
57         $childcontext.setParent($lcontext);
58         insert($childcontext);
59     }
60     modify($lcontext) {
61         setRecursed(true);
62     }
63     // removed: not necessary (only one file)
64     // Fragment $fragment = $context.createFragment($lcontext);
65     // $context.getModel().getRoot().addChild($fragment);
66 end
67
68 // removed: not necessary (only one file)
69 // rule "Reference Insertion" ...
70
71 rule "Data Annotation"
72 when
73     $context: DroolsGlobalContext()
74     $lcontext: DroolsLocalContext($context.knowledgebases["dataIndicators"].
75         termGroups["indicators"] contains node.localName)
76 then
77     List<DataMapping> $mappings = $context.getMappings("data");
78     for (DataMapping $mapping: $mappings) {
79         String $value = $context.evaluateXPathValue($mapping.getSource(),
80             $lcontext.getNode()).replaceAll("\\s+", " ").replace(" - ", " -- ").
81             replace("-", "");
82         if (!$value.equals("")) {

```

```

80         annotate($context.getStack().peek(), $mapping.getTarget(), $value,
81                $context);
82     }
83 end
84
85 //removed: not necessary (no cross references)
86 //rule "Reference" ...
87
88 rule "Fragment"
89 when
90     $context: DroolsGlobalContext()
91     $lcontext: DroolsLocalContext($context.knowledgebases["fragmentIndicators"].
92     termGroups["indicators"] contains node.localName)
93 then
94     Fragment $fragment = $context.createFragment(getId($lcontext, $context));
95     String $ind = getIndicators($lcontext);
96     List<DataMapping> $mappings = $context.getMappings("styleFragment");
97     for (DataMapping $mapping: $mappings) {
98         if ($ind.equals($mapping.getSource())) {
99             $fragment.addStructuralType($mapping.getTarget());
100         }
101     }
102     //removed: not necessary (no keywords)
103     // $mappings = $context.getMappings("keywordFragment");
104     $lcontext.setFragment($fragment);
105     Assertions $kb = (Assertions) $context.getKnowledgebases().get("
106     documentStructure");
107     boolean changed = true;
108     while (changed) {
109         changed = false;
110         for (OntologyClass $newtype: $fragment.getStructuralTypes()) {
111             for (OntologyClass $stacktype: $context.getStack().peek().
112             getStructuralTypes()) {
113                 if ($kb.getBroaderSelf($stacktype.getURI()).contains($newtype.
114                 getURI())) {
115                     $context.getStack().pop();
116                     changed = true;
117                 }
118             }
119         }
120     }
121 }
122 // adapted: exception for child insertion
123 if (!$lcontext.getNode().getLocalName().equals("block"))
124     $context.getStack().peek().addChild($fragment);
125 else
126     $context.removeFragment($fragment);
127 $context.getStack().push($fragment);
128 end
129
130 /* removed: not necessary (only one file)
131 * rule "File Root" */

```

Background knowledge: document structure

```

1 <assertions>
2     <roleAssertion role="hasNarrower"
3         left="vdk:Process" right="vdk:SubProcess"/>
4     <roleAssertion role="hasNarrower"

```

```

5         left="vdk:SubProcess" right="vdk:Choice"/>
6     <roleAssertion role="hasNarrower"
7         left="vdk:Choice"      right="vdk:Step"/>
8 </assertions>

```

Background knowledge: fragment indicators subProcess, variant, blockOccurrence, block

Background knowledge: style fragment mappings

```

subProcess      → vdk:SubProcess
variant         → vdk:Choice
blockOccurrence → vdk:Step
block           → vdk:Step

```

Background knowledge: data indicators subProcess, variant, block

Background knowledge: data mappings

```

./name/text() → vdk:name

```

B.5 Process specifications in Word format (DLR)

Drools extraction rules

```

1 package de.uni_passau.verdikt.DocumentModel.adapter.implementation;
2
3 import de.uni_passau.verdikt.DocumentModel.*;
4 import de.uni_passau.verdikt.DocumentModel.BackgroundKnowledge.*;
5 import org.w3c.dom.*
6 import java.util.*;
7
8 function void annotate(Fragment fragment, String name, Object value,
9     DroolsGlobalContext context) {
10     if (name == null)
11         return;
12     if (name.equals(""))
13         return;
14     if (value == null)
15         return;
16     fragment.addAnnotation(context.createDataAnnotation(name, value));
17 }
18
19 function List getChildElementsRecursive(Element node) {
20     List result = new ArrayList();
21     NodeList children = node.getChildNodes();
22     for (int i=children.getLength()-1;i>=0;i--) {
23         if (children.item(i) instanceof Element) {
24             Element child = (Element) children.item(i);
25             result.addAll(getChildElementsRecursive(child));
26             result.add(child);
27         }
28     }
29     return result;
30 }

```



```

31 function List getChildElements(DroolsLocalContext lcontext) {
32     return getChildElementsRecursive(lcontext.getNode());
33 }
34
35 function String[] getIndicators(DroolsLocalContext lcontext) {
36     // adapted
37     String[] result = new String[2];
38     String ind = lcontext.getNode().getNodeName();
39     result[0] = ind;
40     result[1] = ind;
41     if (lcontext.getNode().hasAttribute("pStyle")) {
42         result[0] += "_" + lcontext.getNode().getAttribute("pStyle").replace(" ",
43             "");
44         result[1] += "_" + lcontext.getNode().getAttribute("pStyle").replace(" ",
45             "");
46     }
47     if (lcontext.getNode().hasAttribute("text"))
48         result[1] += "_" + lcontext.getNode().getAttribute("text").replace(" ", " ");
49     return result;
50 }
51
52 function String getId(DroolsLocalContext lcontext, DroolsGlobalContext context) {
53     // adapted
54     return context.generateId(lcontext);
55 }
56
57 rule "Element Recursion"
58 salience 20
59 when
60     $context: DroolsGlobalContext()
61     $lcontext: DroolsLocalContext(parent == null && recursed == false)
62 then
63     for (Object $child: getChildElements($lcontext)) {
64         DroolsLocalContext $childcontext = $context.createLocalContext();
65         $childcontext.setNode((Element) $child);
66         $childcontext.setParent($lcontext);
67         insert($childcontext);
68     }
69     modify($lcontext) {
70         setRecursed(true);
71     }
72     // removed: not necessary (only one file)
73     // Fragment $fragment = $context.createFragment($lcontext);
74     // $context.getModel().getRoot().addChild($fragment);
75 end
76
77 // removed: not necessary (only one file)
78 // rule "Reference Insertion" ...
79
80 // added: manually add background knowledge to the document model
81 rule "BGK"
82 when
83     $context: DroolsGlobalContext()
84     $lcontext: DroolsLocalContext(node.nodeName == "entity")
85 then
86     Fragment $bgk = $context.createFragment("bgk");
87     $bgk.addStructuralType("vdk:BackgroundKnowledge");
88     for (String $person: $context.getKnowledgebases().get("persons").getTermGroups()
89         .get("persons"))
90         $bgk.addAnnotation($context.createDataAnnotation("ent:person", $person));
91     $context.getModel().getRoot().addChild($bgk);

```

```

89 end
90
91 rule "Data Annotation"
92 when
93     $context: DroolsGlobalContext()
94     $lcontext: DroolsLocalContext($context.knowledgebases["dataIndicators"].
        termGroups["indicators"] contains node.nodeName)
95 then
96     List<DataMapping> $mappings = $context.getMappings("data");
97     for (DataMapping $mapping: $mappings) {
98         String $value = $context.evaluateXPathValue($mapping.getSource(),
        $lcontext.getNode()).trim();
99         if (!$value.equals("")) {
100             // use additional background knowledge: check names against list of
        personel
101             for (String $term: $context.getKnowledgebases().get("persons").
        getTermGroups().get("persons"))
102                 if ($value.toLowerCase().contains($term.toLowerCase())) {
103                     $value = ((SKOSOntology) $context.getKnowledgebases().get("
        persons")).getDefaultTerm($term);
104                     break;
105                 }
106             annotate($context.getStack().peek(), $mapping.getTarget(), $value,
        $context);
107         }
108     }
109 end
110
111 //removed: not necessary (no references)
112 //rule "Reference" ...
113
114 rule "Fragment"
115 when
116     $context: DroolsGlobalContext()
117     $lcontext: DroolsLocalContext($context.knowledgebases["fragmentIndicators"].
        termGroups["indicators"] contains node.nodeName)
118 then
119     Fragment $fragment = $context.createFragment(getId($lcontext, $context));
120     // adapted: multiple indicators
121     String[] $inds = getIndicators($lcontext);
122     for (String $ind: $inds) {
123         List<DataMapping> $mappings = $context.getMappings("styleFragment");
124         for (DataMapping $mapping: $mappings) {
125             if ($ind.equals($mapping.getSource())) {
126                 $fragment.addStructuralType($mapping.getTarget());
127             }
128         }
129     }
130     //removed: not necessary (no keywords)
131     // $mappings = $context.getMappings("keywordFragment");
132     $lcontext.setFragment($fragment);
133     Assertions $kb = (Assertions) $context.getKnowledgebases().get("
        documentStructure");
134     boolean changed = true;
135     while (changed) {
136         changed = false;
137         for (OntologyClass $newtype: $fragment.getTypes()) {
138             for (OntologyClass $stacktype: $context.getStack().peek().getTypes())
        {
139                 if ($kb.getBroaderSelf($stacktype.getURI()).contains($newtype.
        getURI())) {
140                     $context.getStack().pop();

```

```

141         changed = true;
142     }
143 }
144 }
145 }
146 // adapted: add only valid fragments
147 if (!$fragment.getTypes().isEmpty()) {
148     // added: mark the first document
149     if ($fragment.hasStructuralType("vdk:Document"))
150         annotate($fragment, "vdk:initial", $context.initial(), $context);
151     $context.getStack().peek().addChild($fragment);
152     $context.getStack().push($fragment);
153 } else
154     $context.removeFragment($fragment);
155 end
156
157 /* removed: not necessary (only one file)
158 * rule "File Root" */

```

Background knowledge: document structure

```

1 <assertions>
2   <roleAssertion role="hasNarrower"
3     left="vdk:Entity" right="vdk:Document"/>
4   <roleAssertion role="hasNarrower"
5     left="vdk:Document" right="vdk:Section"/>
6   <roleAssertion role="hasNarrower"
7     left="vdk:Document" right="vdk:Process"/>
8   <roleAssertion role="hasNarrower"
9     left="vdk:Document" right="vdk:ProductIn"/>
10  <roleAssertion role="hasNarrower"
11    left="vdk:Document" right="vdk:ProductOut"/>
12  <roleAssertion role="hasNarrower"
13    left="vdk:Process" right="vdk:ProductIn"/>
14  <roleAssertion role="hasNarrower"
15    left="vdk:Process" right="vdk:ProductOut"/>
16  <roleAssertion role="hasNarrower"
17    left="vdk:ProductIn" right="vdk:Process"/>
18  <roleAssertion role="hasNarrower"
19    left="vdk:ProductOut" right="vdk:Process"/>
20  <roleAssertion role="hasNarrower"
21    left="vdk:Process" right="vdk:Step"/>
22 </assertions>

```

Background knowledge: fragment indicators entity, document, paragraph

Background knowledge: style fragment mappings

entity	→	vdk:Entity
document	→	vdk:Document
paragraph_Title-Entity	→	vdk:Section
paragraph_Title-ProductIn_ProductIn	→	vdk:ProductIn
paragraph_Title-Process_Process	→	vdk:Process
paragraph_Title-ProductOut_ProductOut	→	vdk:ProductOut
paragraph_Title-berschrift1	→	vdk:Section
paragraph_Title-berschrift2	→	vdk:Section
paragraph_Title-berschrift3	→	vdk:Section

Background knowledge: data indicators document, paragraph

Background knowledge: data mappings

substring(self::node()[preceding::paragraph[1]/@text='Prepared:']/@text, 6)	→	ent:prepared
substring(self::node()[preceding::paragraph[1]/@text='Reviewed:']/@text, 6)	→	ent:reviewed
substring(self::node()[preceding::paragraph[1]/@text='Approved:']/@text, 6)	→	ent:approved
substring(self::node()[preceding::paragraph[1]/@text='Released:']/@text, 6)	→	ent:released
self::node()[preceding::paragraph[1]/@text='Process Name']/@text	→	data:processname
self::node()[preceding::paragraph[1]/@text='Process ID']/@text	→	data:processid
self::node()[preceding::paragraph[1]/@text='Process Manager']/@text	→	ent:processmanager
self::node()[preceding::paragraph[1]/@text='Entity Name:']/@text	→	data:entityname
self::node()[preceding::paragraph[1]/@text='Entity Path:']/@text	→	data:entitypath
self::node()[preceding::paragraph[1]/@text='Facility Name:']/@text	→	data:facilityname
self::node()[preceding::paragraph[1]/@text='Facility Path:']/@text	→	data:facilitypath
self::node()[preceding::paragraph[1]/@text='Document ID:']/@text	→	dc:subject
self::node()[preceding::paragraph[1]/@text='Issue:']/@text	→	vdk:version
self::node()[preceding::paragraph[1]/@text='Date:']/@text	→	source:change
self::node()[starts-with(@pStyle, 'Title-')]/@text	→	data:topic
self::node()[@numbering='true']/@text	→	data:topic
self::node()[@numbering='true' and @color='0000FF']/@text	→	data:product
self::node()[@table='true' and @column='1' and @pStyle='TableCellValue']/@text	→	vdk:previousversion

SPARQL queries for verification model: select states

```
1 SELECT ?s
2 WHERE { ?s rdf:type vdk:Document }
```

SPARQL queries for verification model: select starting states

```
1 SELECT ?s
2 WHERE {
3   ?s vdk:id "$state" .
4   ?p dc:subject "Entity Input and Output" .
5   { ?s ref:part ?p }
6   UNION
7   { ?s ref:part ?v0 . ?v0 ref:part ?p }
8   UNION
9   { ?s ref:part ?v1 . ?v1 ref:part ?v2 . ?v2 ref:part ?p }
10  UNION
11  { ?s ref:part ?v3 . ?v3 ref:part ?v4 . ?v4 ref:part ?v5 . ?v5 ref:part ?p }
12 }
```

SPARQL queries for verification model: select successors for states

```

1 SELECT ?s
2 WHERE { ?s rdf:type vdk:Document . ?s vdk:id ?i }

```

SPARQL queries for verification model: select values for the concept “ProductIn”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?t . ?t ref:part ?p . ?p rdf:type vdk:
    ProductIn . ?p data:topic ?d }

```

SPARQL queries for verification model: select values for the concept “ProductOut”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?t . ?t ref:part ?p . ?p rdf:type vdk:
    ProductOut . ?p data:topic ?d }

```

SPARQL queries for verification model: select values for the concept “EntityProductIn”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?t . ?t ref:part ?p . ?p rdf:type vdk:
    ProductIn . ?p data:topic ?d }

```

SPARQL queries for verification model: select values for the concept “EntityProductOut”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?t . ?t ref:part ?p . ?p rdf:type vdk:
    ProductOut . ?p data:topic ?d }

```

SPARQL queries for verification model: select values for the concept “Process”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?p . ?p data:processname ?d }

```

SPARQL queries for verification model: select values for the concept “Manager”

```

1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?p . ?p ent:processmanager ?d }

```

SPARQL queries for verification model: select values for the concept “ActivePersonnel”

```
1 SELECT ?d
2 WHERE { ?s rdf:type vdk:BackgroundKnowledge . ?s ent:person ?d }
```

SPARQL queries for verification model: select values for the concept “Approved”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ent:approved ?d }
```

SPARQL queries for verification model: select values for the concept “Reviewed”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ent:reviewed ?d }
```

SPARQL queries for verification model: select values for the concept “Released”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ent:released ?d }
```

SPARQL queries for verification model: select values for the concept “DocumentVersion”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s vdk:version ?d }
```

SPARQL queries for verification model: select values for the concept “Recorded-Version”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s vdk:previousversion ?d }
```

SPARQL queries for verification model: select values for the concept “Source”

```
1 SELECT ?d
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?p . ?p data:processid ?d }
```

SPARQL queries for verification model: select values for the role “hasManager” (base)

```
1 SELECT ?p
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?p . ?p ent:processmanager ?m . ?p data:
    processname ?n }
```

SPARQL queries for verification model: select values for the role “hasManager” (left hand side)

```
1 SELECT ?n
2 WHERE { ?s vdk:id "$base" . ?s data:processname ?n }
```

SPARQL queries for verification model: select values for the role “hasManager” (right hand side)

```
1 SELECT ?m
2 WHERE { ?s vdk:id "$base" . ?s ent:processmanager ?m }
```

SPARQL queries for verification model: select values for the role “managerOf” (base)

```
1 SELECT ?p
2 WHERE { ?s vdk:id "$state" . ?s ref:part ?p . ?p ent:processmanager ?m . ?p data:
    processname ?n }
```

SPARQL queries for verification model: select values for the role “managerOf” (left hand side)

```
1 SELECT ?m
2 WHERE { ?s vdk:id "$base" . ?s ent:processmanager ?m }
```

SPARQL queries for verification model: select values for the role “managerOf” (right hand side)

```
1 SELECT ?n
2 WHERE { ?s vdk:id "$base" . ?s data:processname ?n }
```

B.6 Game of Life

The following Drools rule set implements Conway's Game of Life:

```

1 rule "init"
2 when
3     $m: MediaObject(text == "init")
4     $field: Field()
5 then
6     $field.clear();
7 end
8
9 rule "starvation"
10 when
11     $m: MediaObject(metadata["neighbours"] == "0"
12                     || metadata["neighbours"] == "1")
13     $field: Field()
14 then
15     $field.setCell($m.getMetadata("x"), $m.getMetadata("y"), false);
16 end
17
18 rule "life"
19 when
20     $m: MediaObject(metadata["neighbours"] == "2"
21                     || metadata["neighbours"] == "3")
22     $field: Field()
23 then
24     $field.setCell($m.getMetadata("x"), $m.getMetadata("y"), true);
25 end
26
27 rule "overcrowding"
28 when
29     $m: MediaObject(metadata["neighbours"] == "4"
30                     || metadata["neighbours"] == "5"
31                     || metadata["neighbours"] == "6"
32                     || metadata["neighbours"] == "7"
33                     || metadata["neighbours"] == "8")
34     $field: Field()
35 then
36     $field.setCell($m.getMetadata("x"), $m.getMetadata("y"), false);
37 end
38
39 rule "generation"
40 when
41     $m: MediaObject(text == "generation")
42     $field: Field()
43 then
44     $field.display();
45     MediaObject $o = new MediaObject();
46     $o.setText("generation");
47     insert($o);
48     for (int $y=0;$y<Field.MAX;$y++) {
49         for (int $x=0;$x<Field.MAX;$x++) {
50             int $n = $field.countNeighbours($x, $y);
51             if ($field.getCell($x, $y) || $n == 3) {
52                 $o = new MediaObject();
53                 $o.setMetadata("neighbours", $n);
54                 $o.setMetadata("x", $x);
55                 $o.setMetadata("y", $y);
56                 insert($o);
57             }
58         }

```



```

59     }
60     $field.clear();
61 end

```

It can be applied to XML documents that are serialised as `MediaObjects` and inserted in reverse document order. For example, the “blinker” document

```

1 <gol>
2   <m text="init"/>
3   <m x="2" y="1" neighbours="1"/>
4   <m x="2" y="2" neighbours="2"/>
5   <m x="2" y="3" neighbours="1"/>
6   <m x="1" y="2" neighbours="3"/>
7   <m x="3" y="2" neighbours="3"/>
8   <m text="generation"/>
9 </gol>

```

results in the following alternating output:

```

.....      .....      .....
.....      ..X..      .....
.XXX. => ..X.. => .XXX. => ...
.....      ..X..      .....
.....      .....      .....

```

The slightly more complex “glider” document

```

1 <gol>
2   <m text="init"/>
3   <m x="2" y="1" neighbours="1"/>
4   <m x="3" y="2" neighbours="3"/>
5   <m x="1" y="3" neighbours="1"/>
6   <m x="2" y="3" neighbours="3"/>
7   <m x="3" y="3" neighbours="2"/>
8   <m x="2" y="4" neighbours="3"/>
9   <m x="1" y="2" neighbours="3"/>
10  <m text="generation"/>
11 </gol>

```

results in the following progressing output:

```

.....      .....      .....      .....      .....
.....      .....      .....      .....      .....
.X.X.. => ...X.. => ..X.. => ...X.. => ..... => ...
..XX.. => .X.X.. => ...XX. => ...X. => ..X.X. => ...
..X...      ..XX..      ..XX..      ..XXX.      ...XX.
.....      .....      .....      .....      ...X..
.....      .....      .....      .....      .....

```