Extracting and Storing Document Metadata

Christian Schönberg and Burkhard Freitag

Department of Informatics and Mathematics, University of Passau
{`Christian.Schoenberg, Burkhard.Freitag`}`@uni-passau.de`

Technical Report, Number MIP-0907
Department of Informatics and Mathematics
University of Passau, Germany
August 2009

Contents

Abstract

This paper gives an overview of information extraction techniques, metadata storage practices, and metadata querying and transformation methods as they are employed in the context of the Verdikt research project.

As a major part of document verification, an abstract internal model of the document to be processed has to be generated. We describe the overall model-generating procedure with a focus on metadata extraction and storage. Good practice is presented and potential problems are discussed.

1 Introduction

With online-publishing of documents steadily growing, it is becoming increasingly difficult to keep the structure and content of documents like technical documentations or e-learning material in a consistent state. Manufacturers reuse content across several documentations, compiling documents from a host of separate resources and text fragments that depend on current requirements and priorities. Similarly, e-learning courses are often assembled from existing modules, while their actual instances depend on the specific didactical intentions. Documents published in a hypertext format such as XML or HTML add another difficulty: there is usually more than one (linear) path through the document. For realistically large documents this makes it almost impossible to check consistency criteria manually. Therefore, a means to verify digital documents against specified criteria is gaining in impact and relevance [KSS04].

The goal of the Verdikt[1] project is to provide a method of automatic document verification. An overview of the general approach of the project is shown in Figure 1: a user specifies some consistency criteria using a high-level specification method (left hand side, top). The resulting specification is transformed into a formal specification using temporal description logics (left hand side, bottom). The relevant information is extracted from the document to be checked (right hand side, top) and transformed into a document model. This model can be enriched with background knowledge (right hand side, middle). A formal verification model is generated from the enriched document model (right hand side, bottom). Model checking is used to test the formal specification against the formal model. A positive result is returned if the document satisfies the specification. Otherwise, an error report detailing the specification violations is created. A description of the overall verification process can be found in [WJF09]. Details about the employed temporal description logic, about the model checking process, and further information about the verification process can be found in [Wei08]. An approach to support the user-level specification of consistency criteria is detailed in [JF08].

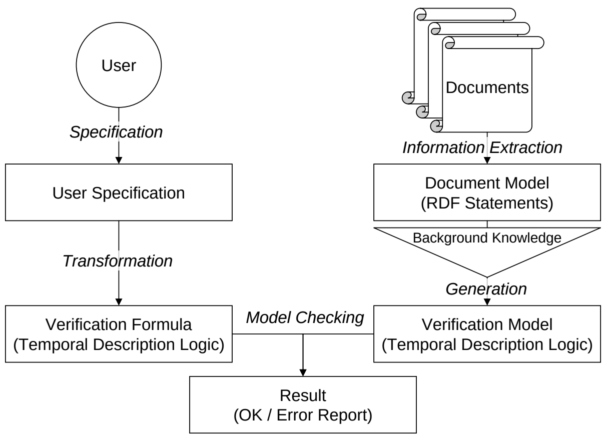


Figure 1: Overview of the Verdikt approach

A detailed view of the right-hand-side of Figure 1 is shown in Figure 2. Documents are preprocessed into a normalised and simplified form (top), from which the document model

[1]This work is funded by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) under grant number FR 1021/7-1.

is extracted in the form of RDF statements using Java or XQuery (centre). Employing existing RDF frameworks like Jena [CDD+04] or Sesame [BKH02], the statements of the document model can be stored persistently (left hand side). Using ontology reasoning, the document model can also be enriched with further background knowledge (right hand side). Two options exist for generating the verification model from the document model: on the one hand, the verification model can be generated directly using Java. On the other hand, the verification model can be generated from an XML view of the RDF statement graph using XQuery (left and right hand side, bottom).

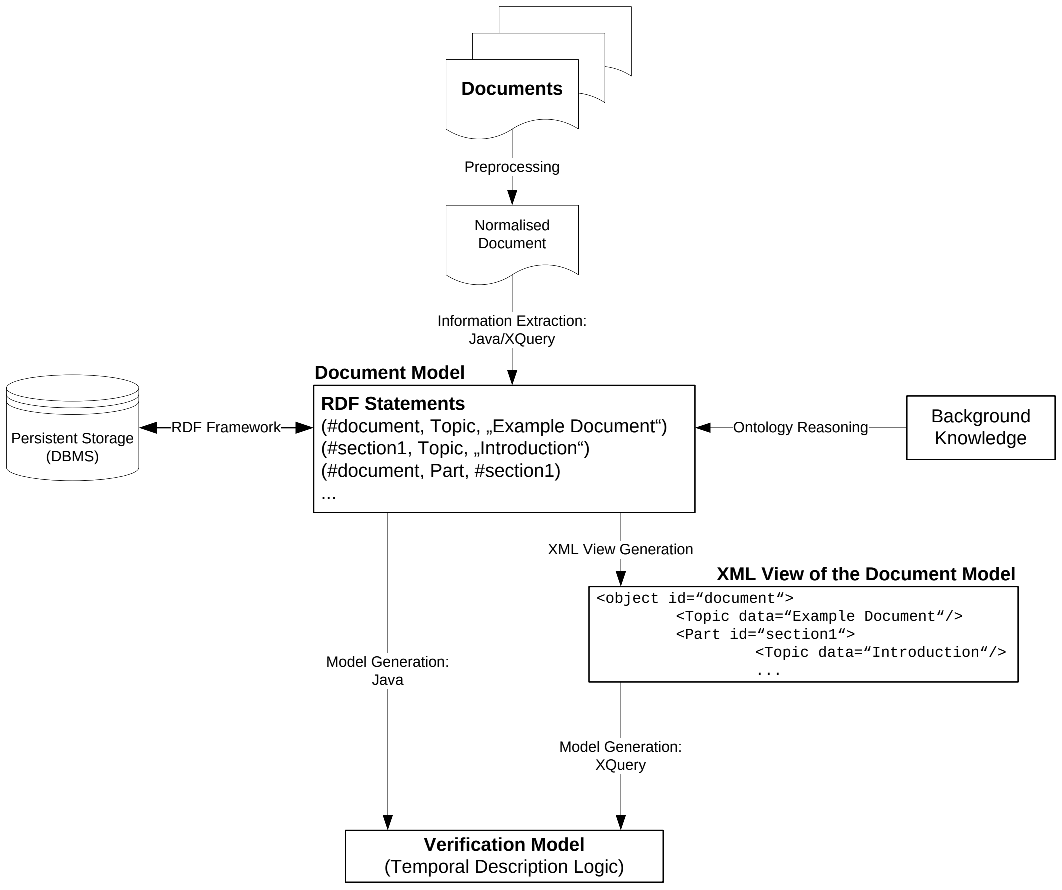

Documents
Preprocessing
Normalised Document
Information Extraction: Java/XQuery
Document Model
RDF Statements
(#document, Topic, „Example Document“)
(#section1, Topic, „Introduction“)
(#document, Part, #section1)
...
Persistent Storage (DBMS)
RDF Framework
Ontology Reasoning
Background Knowledge
XML View Generation
XML View of the Document Model
<object id="document">
<Topic data="Example Document"/>
<Part id="section1">
<Topic data="Introduction"/>
...
Model Generation: Java
Model Generation: XQuery
Verification Model
(Temporal Description Logic)

Figure 2: Detailed view of the information extraction and model generation

The framework developed as part of the Verdikt project has been tested on documents from the domains of technical documentation, e-learning, and web pages. To that end, the framework has to be able to handle several different file formats, including HTML, DocBook, DITA, LMML, <ML^3> and Microsoft Word.

This report focuses on a description of the metadata extraction and model generation process. A description of the document model used to represent and store the metadata information is provided in Section 2. The extraction process is outlined in Section 3. Requirements for persistently storing the document model are detailed in Section 4, while the implemented solution is explained in Section 5. The generation of a verication model from the document model is presented in Section 6. Section 7 concludes this report with a summary.

2 Document Properties

The Verdikt project aims at verifying consistency criteria in documents. To allow for a better understanding of this process, we will explain the notion of a *Document* as it is used and understood in the Verdikt context. To that end, we also explain the terms *Corpus* and *Fragment*.
A *document* is a tree of fragments: *node fragments* make up the inner structural parts of the document, while *atomic fragments* are its leaves and contain the actual text or media content. Any sensible partitioning of a document that is both useful from the point of view of the intended application and compatible with the underlying document format is a valid set of fragments. An atomic fragment is a part of a document that is chosen to be treated as indivisible.
Child nodes of a fragment are called *sub-fragments*. Sub-fragments are ordered, so that a unique document order (cf. [XPA99]) is imposed on the document tree. A fragment's immediate successor is its first sub-fragment; if no sub-fragment exists, the successor is its first sibling (the next sub-fragment of the parent node); if no sibling exists, the successor is the first sibling of the first parent that has at least one sibling. The only fragment without a successor is the very last leaf fragment of the document. Figure 3 shows an illustration of the document order, of sub-fragments, and of successors.
Our understanding of documents and their fragments is closely related to the data model of XPath 2.0 [XPa07] and XML Information Set [XML04].

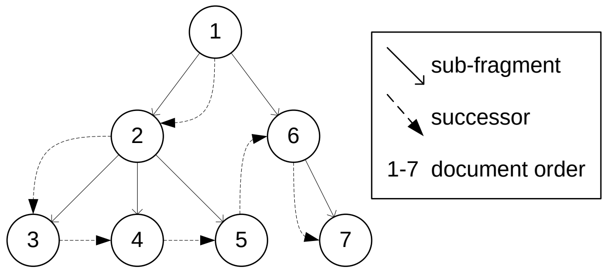


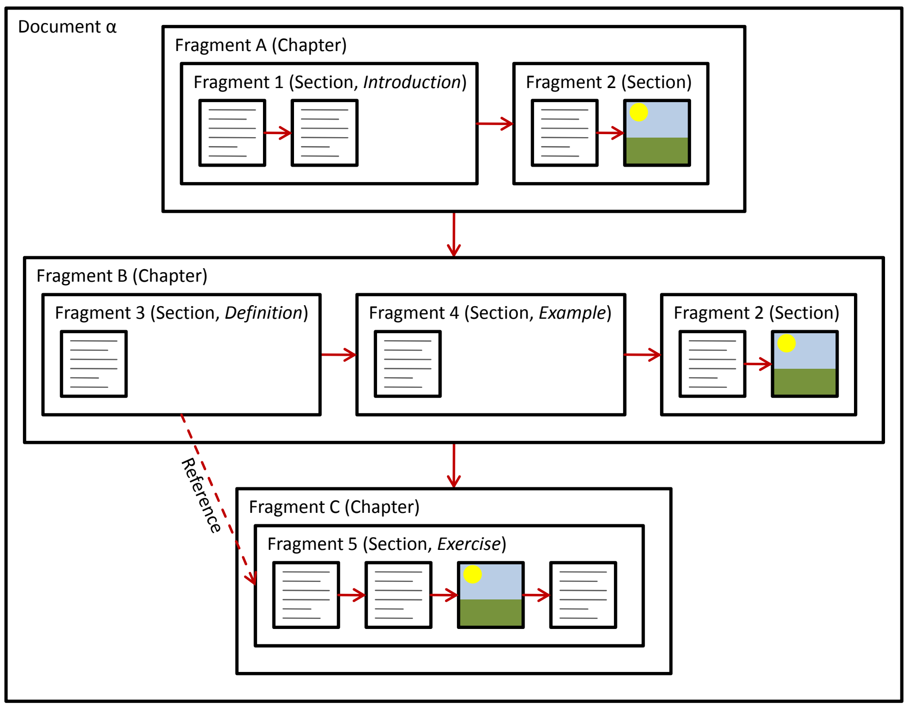
Figure 3: Document order

There can be references between fragments. They have no bearing on the document's tree structure, but they offer additional valid paths through the document, apart from traversing the tree in document order. Possible types of references include direct links, indirect references (e.g. "See Chapter IV" or "See the definition of Binary Trees"), citations, references to external entities (e.g. hyperlinks to external web sites), and includes (including fragments into the document tree from an external source). Includes are the only kind of references that extend the document tree: included fragments and their structure become part of the tree. At this point, we only consider finite documents, e.g. we do not allow self-including fragments or circular inclusions.
Node fragments can often be mapped onto structural components of the document (e.g. chapters or sections). In such cases, they are assigned a *structural type* (e.g. "Chapter"). Apart from their structural properties, they can often also be mapped onto functional descriptions (e.g. "Introduction" or "Definition"), in which case they are assigned a matching

function type. Fragments are often assigned additional document metadata, such as topics or didactic information. While node fragments represent no textual or other content of their own (apart from topics and headlines), they aggregate such content in the form of their atomic sub-fragments. Non-content metadata like didactic information or data about the fragment's source file are passed down the document tree and inherited by the sub-fragments of a document node.

Illustration
Document

***Illustration* 2.1** (Document)
The illustration shows an example document consisting of three chapters. Fragment 2 is included twice, first as a sub-fragment of Fragment A, then – by re-use – as a sub-fragment of Fragment B. Fragments 1, 3, 4 and 5 have function types (in *italics*), while all fragments have a structural type. Fragments that are not explicitly named have the structural types "Paragraph" and "Image", respectively. There is a cross-reference between Fragment 3 and Fragment 5, which is not part of the document tree but creates a new path through the document: a reader can omit Fragments 4 and 2 and jump directly to Fragment 5.
All named fragments (including the document itself) are node fragments, while the unnamed fragments are atomic fragments.


Document α
Fragment A (Chapter)
Fragment 1 (Section, *Introduction*)
Fragment 2 (Section)
Fragment B (Chapter)
Fragment 3 (Section, *Definition*)
Fragment 4 (Section, *Example*)
Fragment 2 (Section)
Reference
Fragment C (Chapter)
Fragment 5 (Section, *Exercise*)

We do not regard documents or fragments that are not digitally stored. However, apart from this, no further storage limitations are imposed on a document: it can be stored as one or more files in the filesystem, as one or more files across a network, or in a database. A document does not even have to be written in a single file format: different fragments can have different formats. However, usually a document is given in only a small number of different formats, which are designed to work well together, e.g. HTML and PNG/SVG/JPEG, or different formats of the Microsoft Office family.

Document formats are usually either *semi-structured* (like the XML-based formats DITA [DIT07], DocBook [Doc09] or LMML [Süß05]), or *sparsely-structured* (like HTML, LaTeX, or Microsoft Word). Semi-structured formats add metadata information to the pure text, such as structural or function types (see above), as well as content-related data, such as keywords, important terms, or objectives. There can also be metadata concerning didactic information, source references, quality assessment, and more. Sparsely-structured formats also add metadata information to the textual content, but – as the name suggests – fewer in both quantity and quality. While for example the plain HTML format tags headlines (and by that, indirectly, sections and subsections), the LMML format also offers information about the function of and the terms covered by a section. Some information extraction techniques also provide meta-metadata, i.e. data about the quality (e.g. probability of correctness) of the metadata that was extracted from a document.
A listing of relevant metadata used in the Verdikt project can be found in Appendix A. A listing of the metadata information supported by different file formats can be found in Appendix B.
After having described single documents and their structure and building blocks, i.e. fragments, we now proceed to sets of documents. A *corpus* is usually a collection of documents that have some connection with one another: they share the same topic or domain, they describe the same or a similar product, they are written by the same author, or they serve a similar purpose. Grouping documents has several advantages for the Verdikt system in general, and the metadata extraction component in particular. The consistency criteria checked with the Verdikt system often differ very little – if at all – across a suitably defined corpus, thus limiting the effort required to specify them. Also, the documents of a corpus usually share common characteristics, such as structure, terminology, key words, or other background knowledge. Therefore, there is reason to assume that if the metadata extraction works well for one of the documents of a corpus it will work similarly well for the rest of its documents.

2.1 Verdikt Document Model

The Verdikt project does not require the entire document for its verification process. Instead, an abstract document model containing certain metadata, including structural information about the document and some indication of the content of its fragments, such as occurring terms or abbreviations, is sufficient.

Definition
Document Model

***Definition* 2.1** (Document Model)
The *document model* is a set of statements, each of which is a triple consisting of *subject*, *predicate* and *object*. In general, the statements form a directed graph. The edges are annotated and represent the predicates of statements, while the nodes represent subjects and objects. The directionality of each edge indicates which adjunct node is the subject and which is the object of the statement. Nodes can contain textual data: these nodes are called *Data Objects* – graphically denoted as rectangular boxes with values enclosed in double quotes – and can only be used as objects in statements. Nodes that do not contain textual data are called *Content Objects* – graphically denoted as ellipsoids – because they are used to cluster other nodes and as such are often hubs for multiple statements. Nodes representing statements are called *Statement Objects* – graphically denoted as diamond-shaped boxes.
Documents and their fragments are modelled as *Content Objects* in this graph, with the relevant metadata attached to them in the form of *Data Objects*. The relations between nodes are defined by the semantics of the connecting *predicates*, which for the purpose of

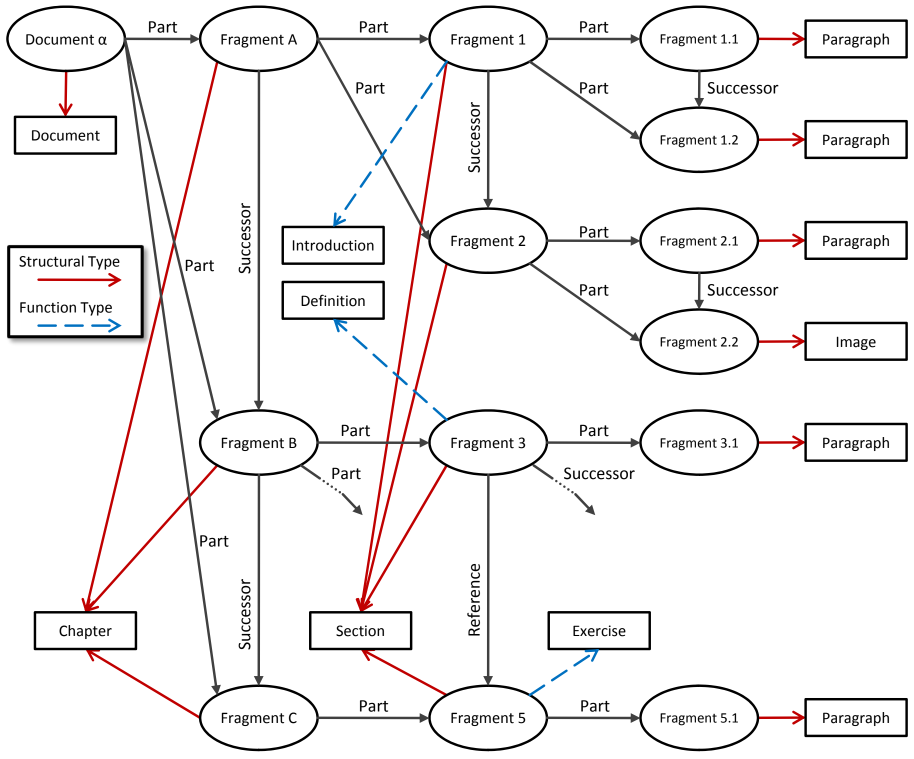
the Verdikt project use a standardised vocabulary. A listing of this vocabulary can be found in Appendix C.

Definition
Reification

***Definition* 2.2** (Reification)
Statements can have entire other statements as their subjects. The process of using statements as nodes to facilitate statements about other statements is called *reification*. A statement that is the subject of another statement is called a *reified statement*. A statement that reifies another statement (i.e. that has another statement as a subject) is called a *reification statement*.
Statement Objects are used to model reification, while their graphical representations are used to represent reification graphically (see e.g. Figure 8).

Illustration
Document Model

***Illustration* 2.2** (Document Model)
The illustration shows a graphical representation of parts of the model of the document shown in Illustration 2.1. Some fragments and sub-fragments have been omitted for the sake of clarity.



Example
Document Model

***Example* 2.3** (Document Model)
The example lists some of the statements corresponding to the model shown in Illustration 2.2. Statements 2 through 4 as well as statements 6 and 7 indicate how the document is structured: Fragments A through C are the parts (sub-fragments) the document itself is composed of, while Fragments 1 and 2 are both parts of Fragment A. Statements 1, 5 and 8 describe the structural type of the indicated fragments: the document is labelled "Document", Fragment A is labelled "Chapter", while Fragment 1 is denoted to be a "Section". Statement 9 asserts the functional type of Fragment 1 to be "Introduction". Statement 10 specifies that Fragment 2 is an immediate successor of Fragment 1, meaning that Fragment 2 follows directly after Fragment 1 according to the document order.

$$\left\langle (\text{Document } \alpha)\,, \xrightarrow{\text{StructuralType}}, [\text{“Document”}] \right\rangle \tag{1}$$

$$\left\langle (\text{Document } \alpha)\,, \xrightarrow{\text{Part}}, (\text{Fragment A}) \right\rangle \tag{2}$$

$$\left\langle (\text{Document } \alpha)\,, \xrightarrow{\text{Part}}, (\text{Fragment B}) \right\rangle \tag{3}$$

$$\left\langle (\text{Document } \alpha)\,, \xrightarrow{\text{Part}}, (\text{Fragment C}) \right\rangle \tag{4}$$

$$\left\langle (\text{Fragment A})\,, \xrightarrow{\text{StructuralType}}, [\text{“Chapter”}] \right\rangle \tag{5}$$

$$\left\langle (\text{Fragment A})\,, \xrightarrow{\text{Part}}, (\text{Fragment 1}) \right\rangle \tag{6}$$

$$\left\langle (\text{Fragment A})\,, \xrightarrow{\text{Part}}, (\text{Fragment 2}) \right\rangle \tag{7}$$

$$\left\langle (\text{Fragment 1})\,, \xrightarrow{\text{StructuralType}}, [\text{“Section”}] \right\rangle \tag{8}$$

$$\left\langle (\text{Fragment 1})\,, \xrightarrow{\text{FunctionType}}, [\text{“Introduction”}] \right\rangle \tag{9}$$

$$\left\langle (\text{Fragment 1})\,, \xrightarrow{\text{Successor}}, (\text{Fragment 2}) \right\rangle \tag{10}$$

3 Metadata Extraction

The Verdikt framework provides two different approaches to metadata extraction from documents. Figure 4 shows part of the general architecture of the framework.
The `DocumentModel` package contains the abstract classes and interfaces that represent the document model:

- The `MetadataModel` interface represents the document model as a whole, storing all pertinent data and providing access to its statements.
- The `VerdiktObject` interface is the base interface for the objects that make up the statements and data of the document model.

- The `DataObject` interface represents simple data entries, like e.g. strings or integers. It can only be used as the object of a statement, not as its subject.
- The `ResourceObject` interface is the base interface for more complex objects than `DataObjects`. Like RDF's *resources*, `ResourceObjects` can be used as both the subject and the object of a statement.
- The `ContentObject` interface represents objects that do not have any value themselves, but are used as hubs to group other objects.
- The `VocabularyObject` interface represents terms that can be used as the predicate in a statement.
- The `StatementObject` interface represents statements. `StatementObjects` themselves can be used as a subject or a an object in other statements.

The `DocumentAdapter` package contains classes used to create, modify, transform or display document models. The abstract `DocumentAdapter` class provides a set of utility methods, such as XSLT capabilities, reading and writing of files, creating checksums etc., that can be used by classes extending the `DocumentAdapter` class. It also defines the two primary methods `load` and `save` any extending class has to define. These methods are used to import a document model from a source or transform it into a destination format. While several classes extending the `DocumentAdapter` class have been defined to facilitate transformation capabilities from and to various formats, several have been omitted in Figure 4 for the sake of clarity. For classes like `DocBookDocumentAdapter` or `LMMLDocumentAdapter`, the `load` method provides the most important functionality: these classes are used to import documents in various formats (e.g. DocBook [Doc09] and LMML [Süß05]), which is a specialisation of their common base class `DocumentAdapter`. The `RDFDocumentAdapter` class relies on both the `load` and `save` methods: the adapter is used to provide both read and write access to RDF type files. Classes like `JTreeDocumentAdapter` primarily use the `save` method: they are used for display purposes (e.g. creating a Java tree view). Classes like `RuleDocumentAdapter` also focus on using the `save` method: they are used to transform the document model into another format (e.g. into a verification model). The `XQueryDocumentAdapter` is a special case: it is used to run an external XQuery program. This program can be used on an XML document to extract metadata and to create a document model, or it can be used on an XML serialisation of the document model to create a verification model.
The `rules` package contains the necessary abstract classes and interfaces for specifying how to generate a verification model from the document model using the `RuleDocumentAdapter` class, which extends the abstract `DocumentAdapter` class.
The `Memory`, `Jena`, and `Sesame` packages contain implementations of the `MetadataModel` that have been optimised for usage in combination with RDF storage in main memory, RDF storage using the Jena framework, and RDF storage using the Sesame framework, respectively.
Figure 5 shows how RDF frameworks like Sesame or Jena can be integrated into the Verdikt framework for managing RDF data. Since the integration is based on interfaces, external frameworks can be used transparently, and can be exchanged or updated easily.
Extracting the relevant metadata from a document is a difficult process. Two possible approaches to metadata extraction that have been developed as part of the Verdikt project are detailed below.

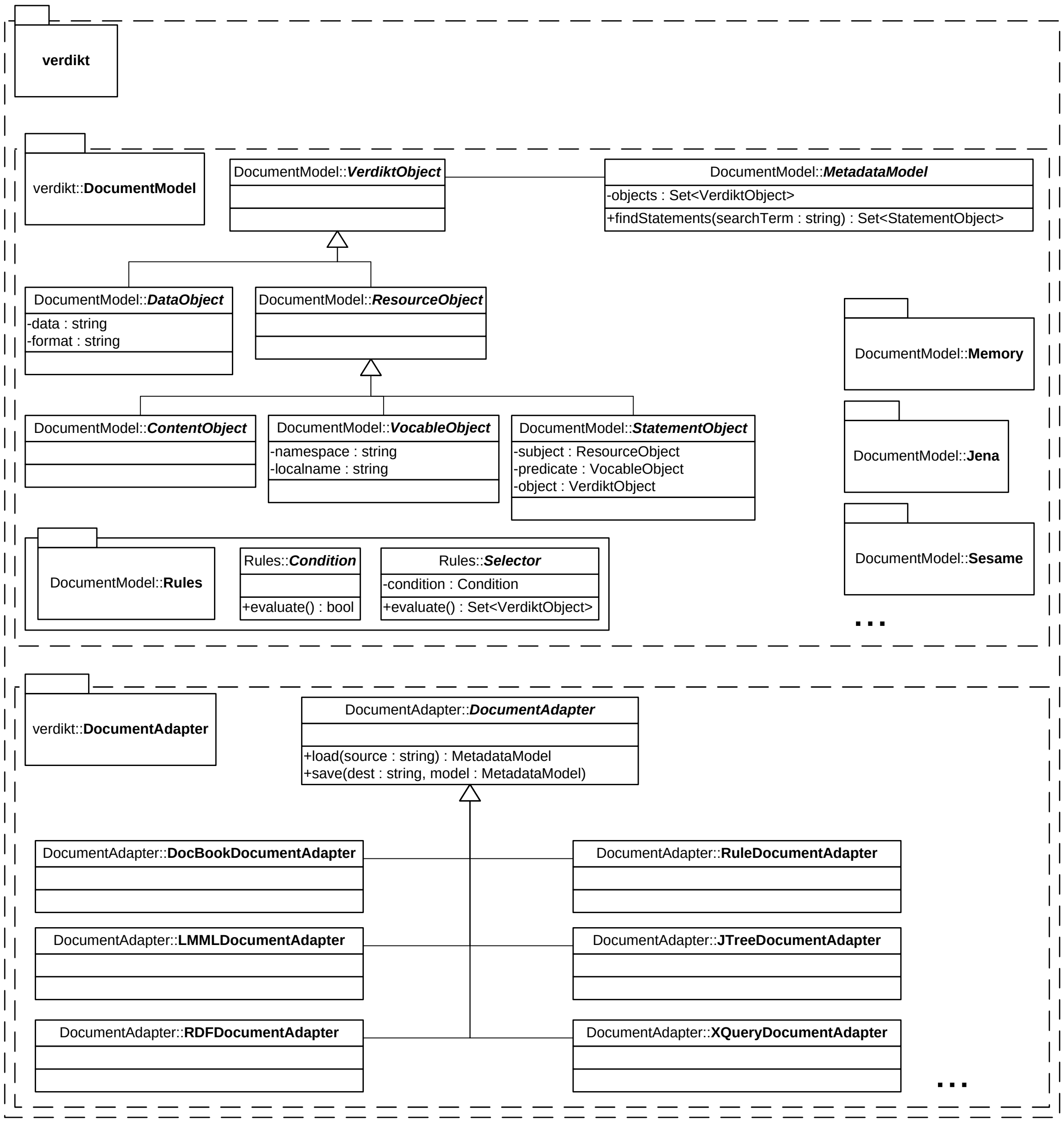

verdikt
verdikt::DocumentModel
DocumentModel::VerdiktObject
DocumentModel::MetadataModel
-objects : Set<VerdiktObject>
+findStatements(searchTerm : string) : Set<StatementObject>
DocumentModel::DataObject
-data : string
-format : string
DocumentModel::ResourceObject
DocumentModel::Memory
DocumentModel::ContentObject
DocumentModel::VocableObject
-namespace : string
-localname : string
DocumentModel::StatementObject
-subject : ResourceObject
-predicate : VocableObject
-object : VerdiktObject
DocumentModel::Jena
DocumentModel::Rules
Rules::Condition
+evaluate() : bool
Rules::Selector
-condition : Condition
+evaluate() : Set<VerdiktObject>
DocumentModel::Sesame
...
verdikt::DocumentAdapter
DocumentAdapter::DocumentAdapter
+load(source : string) : MetadataModel
+save(dest : string, model : MetadataModel)
DocumentAdapter::DocBookDocumentAdapter
DocumentAdapter::RuleDocumentAdapter
DocumentAdapter::LMMLDocumentAdapter
DocumentAdapter::JTreeDocumentAdapter
DocumentAdapter::RDFDocumentAdapter
DocumentAdapter::XQueryDocumentAdapter
...

Figure 4: General architecture of the Verdikt framework

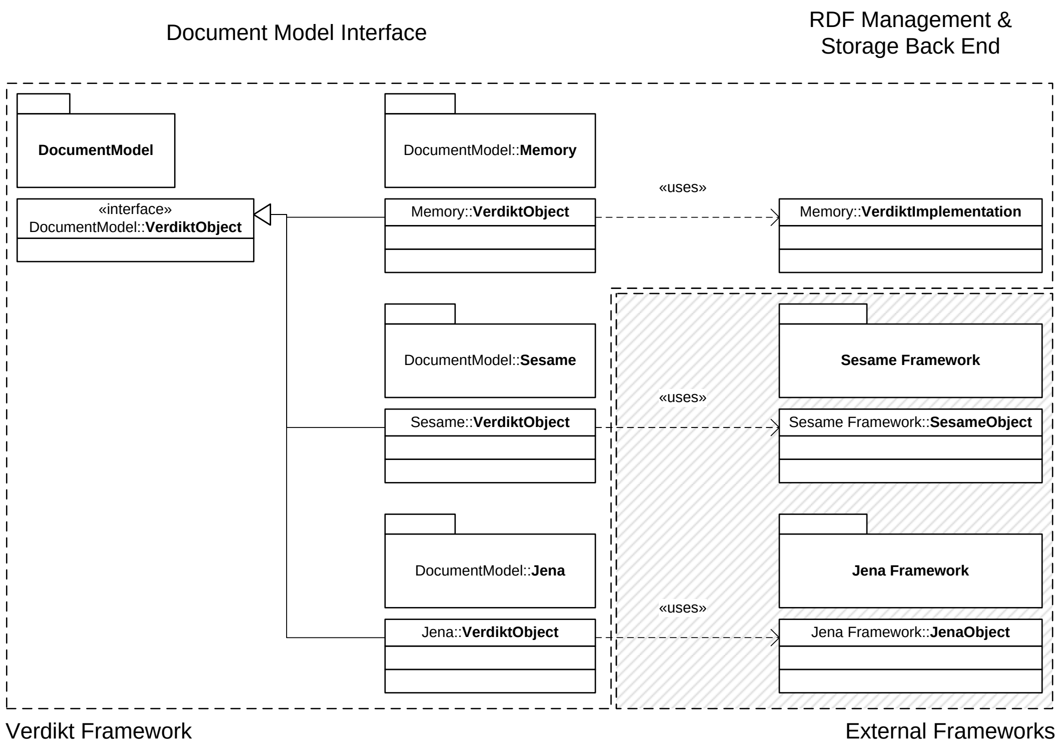


Figure 5: RDF storage back end of the Verdikt framework

3.1 Metadata Extraction using Java

A powerful and versatile approach to metadata extraction is based on the creation of separate Java class libraries for each required format. This method is powerful because it can take advantage of the expressiveness of the Java programming language and any libraries available; it is versatile because it can be used to support any document format. It is, however, also limiting in the sense that it requires not only knowledge and ability in programming, but relies also on program libraries. These have to be pre-compiled and integrated into the Verdikt framework each time a change or extension is to be incorporated.

As another approach to metadata extraction using Java, one can create a single program for all possible input formats, and parameterise it using appropriate background knowledge for each format. Knowledge in the form of facts, rules, and constraints can be used to guide the program on how metadata is to be extracted from any applicable format. This approach mitigates the limitation that the introduction of a new format requires a person with Java programming skills to create a new program. However, it only moves this limitation to a different level: now an expert in the specification of complex background knowledge is required to create the parameters and rules for the existing program.

The approach also introduces a new limitation: a specification language for these parameters and rules powerful enough to represent any extraction rule for documents has to be either found or designed. While candidates for such a specification language exist (e.g. RuleML [Ger03] or Drools [DRO09]), it still has to be shown that all necessary background knowledge can be specified effectively and efficiently.

Therefore, the Verdikt project employs the more viable approach of creating specialised Java programs for different document formats, incorporating all required knowledge directly into the Java implementation. Nonetheless, parametrisation can still be a useful concept in a more limited context. A prototype for an information extraction program based on XML Schema files for different document formats, augmented with metadata extraction rules, is currently under development as part of the Verdikt project.

***Note* 3.1**
To alleviate the complexity of integrating new libraries into an existing framework as noted above, the Verdikt framework provides a convenient plug-in mechanism. Java class files adhering to the necessary interfaces can simply be copied into a dedicated directory within the framework and will be recognised by the Verdikt system automatically.

3.1.1 Information Extraction Procedure

Information extraction, including metadata extraction, is usually done in three stages: *preprocessing*, *extraction*, and *postprocessing*. Preprocessing is performed to decrease the complexity of the core data extraction process. It can be used to bring the source file(s) into a concise and cohesive form, to reduce redundancies, to simplify the structure, or to convert files from a less easily accessible format to a more convenient one. The extraction process locates, selects, accumulates, aggregates and filters relevant data from the preprocessed source document(s) into a standardised form. Postprocessing is done to optimise the extracted data for a specialised use case, e.g. for display purposes, or for domain specific subsequent processing (see Section 6).

Preprocessing The Verdikt framework supports three major preprocessing components:

- The first is an instance of the *XML Tidy* library [XML08], which is used to correct errors in XML documents that are not well-formed. It can also be used to bring HTML files into XHTML form, which is better suited for being processed.
- The second component is a support structure for *XSL transformation*, giving libraries easy access to powerful transformation capabilities between XML formats.
- The third component is an external library, written in Microsoft .Net, that can read files in *Microsoft Word* format and produces an XML file containing relevant features of the Word file (see Figure 6 for an example of such features). This XML file can be further processed by other components of the Verdikt framework.

***Note* 3.2**
Other proprietary file formats such as Adobe PDF, Adobe Indesign or QuarkXPress that lack open interfaces are not supported directly. However, files in proprietary formats can often be converted into a supported file format using an external application.

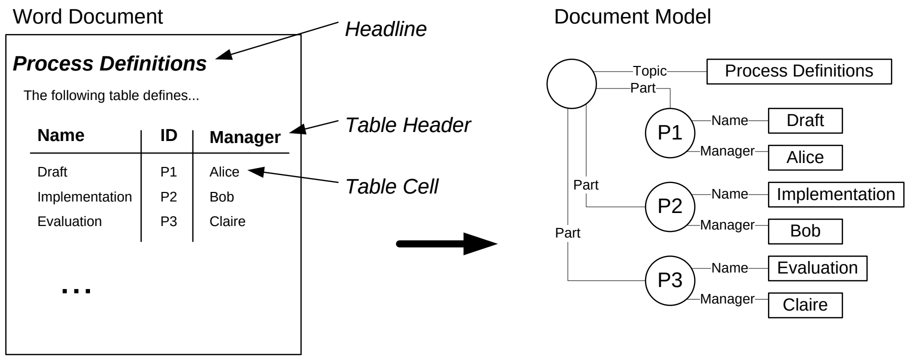


Figure 6: Feature extraction from a Microsoft Word document

Extraction In the field of information extraction/information retrieval, three major applications are the *extraction of specific data* from a document (e.g. names, dates, or dates related to names) [Ham99, Ste03, Rob05], the *classification* of the components of a document (e.g. examples or definitions) [Mic00], and the *indexing* of relevant terms in a document (e.g. for search engines) [Ber01]. In the context of the Verdikt project, it is not only necessary to extract any *possibly relevant* data and terms, but also to apply a classification process to generate a model of the source document that is as complete as possible (see Section 2). For details on why the document model should be complete, see Section 6.

Many information extraction tools and techniques employ either XPath, rules, or regular expressions, all with similar expressiveness [STS00, CM04, Mil02]. Since the Verdikt system focusses on processing XML-based file formats and on XML representations converted from other formats (cf. the paragraph on preprocessing above), XPath seems to be a sensible choice. XPath is a powerful tool for locating data at well-defined positions. However, specifying locations on specific or even alternating recursive paths (e.g. $a/[b]^*/c$ or $a/[b/c]^*/d$) is beyond the expressiveness of XPath (see also Note 6.5). Furthermore, specifying locations that depend on complex contextual information (regarding both content and structure, see below) is both time consuming and error prone. Therefore, another approach is required.

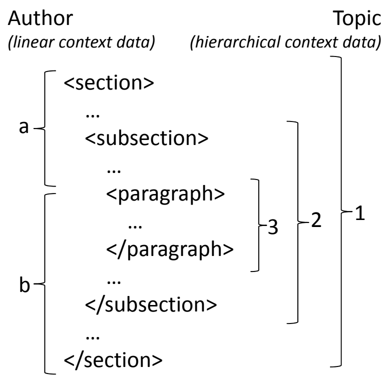
Recursive Processing For data that is difficult or impossible to locate using XPath, a recursive algorithm is required that traverses the document along the lines of sub-fragments, successor fragments, and references. This process needs to keep track of the document context as it moves between document fragments.

Definition
Document Context

***Definition* 3.1** (Document Context)
In general, the *Document Context* of a document fragment is the complete path through the document (cf. Section 2) that was followed starting from the document root to reach the fragment.
However, in most cases it is not necessary to regard the entire path to a fragment. It is often sufficient to gather relevant properties that were encountered along that path, such as information about chapters or topics. This leads to a distinction between two types of context data: context data that changes its value when the document context changes from a fragment to one of its sub-fragments, and that reverts to its former value when the context changes back to the original fragment is called *hierarchical context data*. Context data that changes its value when the document context changes from one fragment to another, but remains unaffected when the context changes back to a parent fragment of the current sub-fragment is called *linear context data*.

Illustration
Document Context

***Illustration* 3.3** (Document Context)
The illustration shows a small part of a document and the points at which the two different types of context data can change their values.



The *linear context data* about the author that is marked on the left hand side changes its value at an arbitrary point in the document, thus ignoring its hierarchical structure. The *hierarchical context data* about the current topic that is marked on the right hand side only changes in accordance with the document structure.

***Note* 3.4**
It is possible that a fragment can be reached on multiple paths through the document. Therefore, a fragment can occur in different document contexts in the same document. The document context will be determined dynamically at run time, instead of statically before processing the document. Dynamic context determination corresponds to the intuition that the same text fragment can play different roles depending on its occurrence within a document or document corpus.

***Note* 3.5**
When processing a document recursively, the context data of the current fragment depends on the path through the document that was taken to reach this fragment. If the same fragment can be reached on different paths, it is processed multiple times with different context data. The Verdikt system supports this recursive process by providing a specialised data structure to model the document context during the recursion.

3.1.2 Information Extraction Utilities and Techniques

Relevant utilities and techniques for information extraction include *dictionaries*, *language ontologies*, *language recognition* and *domain ontologies*.

Dictionaries *Keyword recognition* is an important aspect of information extraction. It is also important to extract terms in such a way that they can be mapped on or included into a standardised vocabulary, to detect semantical relations between text fragments. For both purposes, dictionaries can be used in combination with *grammar rules*. By detecting and altering word forms, extracted terms can be brought into a grammatical normal form (e.g. nominative singular) through stemming (root reduction). This makes it easier to compare and recognise extracted words and phrases.
Dictionaries from the OpenOffice project [Ope09] are employed in the Verdikt system by means of the JMySpell Java interface[2], which also provides access to the grammatical rules included with the dictionaries. The Verdikt system provides simplified access to the dictionary-related commands and techniques of the JMySpell interface that are useful for information extraction.

Language ontologies Language ontologies can specify semantic relations between words. They are useful for detecting alternate keywords for keyword recognition. They can also be used as a substitute for specialised domain ontologies when these are not available. An ontology for the English language is included into the Verdikt system in the form of Wordnet [Wor06].

Language recognition A prerequisite for employing any of the language dependent utilities named above is the correct identification of the language of any given text fragment. In some cases, the name of the language of the text is encoded in the document metadata (e.g. using an `xml:lang` attribute or a `dc:Language` element). In other cases, i.e. for converted Word documents, the name of the language as provided by the word processor used

[2] `http://jmyspell.javahispano.net/index_en.html`, last visited Aug. 2009

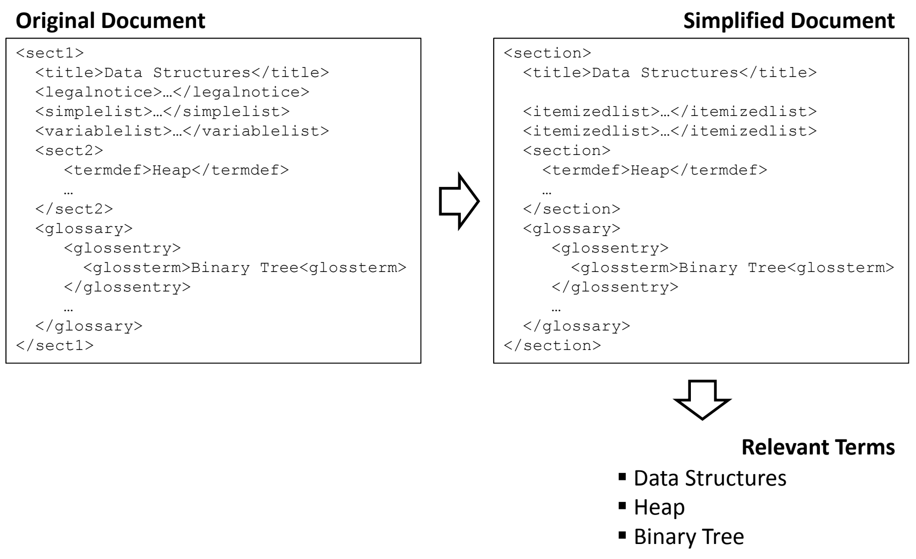
to create the document is included in the document metadata. The Verdikt system also implements a language recognition heuristic based on the occurrence of common words for several languages.

Domain ontologies Lists of relevant terms, as well as extended semantic relations between relevant terms can be found in specialised domain ontologies. A predefined list of important domain terms can improve the *recall* value of the information extraction, i.e. the percentage of terms that were recognised in the document as opposed to those that were missed. Knowledge about semantic relations between extracted terms can be used to create a more complete and meaningful model of the extracted data. At the moment, the Verdikt system supports basic domain ontologies, but does not make use of the richer semantic background these ontologies can provide.

Miscellaneous In addition to supporting the techniques and tools described above, the Verdikt framework provides a number of utilities to assist in creating new libraries for new document formats. Among them are several convenience methods: a graphical interface for user input, graphical feedback to the user, handling of temporary files, conversion between simple XML Schema types and Java types, reading and writing text and XML files, switching between multiple available XML implementations, creating file checksum data, handling data compression and compressed files, and providing a common way to pass external options to the library. The latter method is useful to change the default behaviour of static libraries. These methods are accumulated in the abstract `DocumentAdapter` class (cf. Figure 4). For a list of these methods see Appendix D.

Example
Java/DocBook

***Example* 3.6** (Java/DocBook)
In this example, we will describe briefly how a metadata model is extracted from documents in the XML-based DocBook format.
First, using XSL preprocessing, the DocBook document is transformed to conform to a simplified version of the DocBook standard: multiple files are integrated into one, different yet equivalent ways to encode the same data are unified, and comments and other superfluous elements are removed.



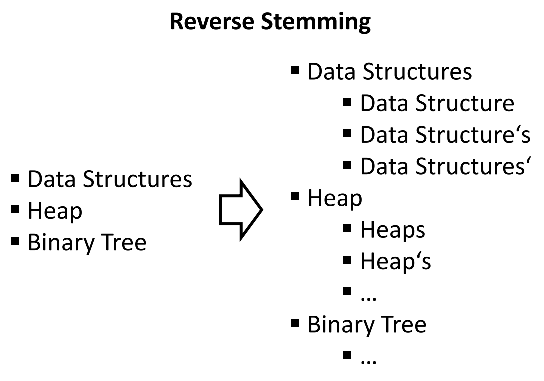
```
<sect1>
  <title>Data Structures</title>
  <legalnotice>…</legalnotice>
  <simplelist>…</simplelist>
  <variablelist>…</variablelist>
  <sect2>
     <termdef>Heap</termdef>
     …
  </sect2>
  <glossary>
     <glossentry>
       <glossterm>Binary Tree<glossterm>
     </glossentry>
     …
  </glossary>
</sect1>
```

```
<section>
  <title>Data Structures</title>

  <itemizedlist>…</itemizedlist>
  <itemizedlist>…</itemizedlist>
  <section>
    <termdef>Heap</termdef>
    …
  </section>
  <glossary>
     <glossentry>
       <glossterm>Binary Tree<glossterm>
     </glossentry>
     …
  </glossary>
</section>
```

Next, also using XSL technology, a list of relevant terms is extracted from the document by probing appropriate XML elements such as definition or glossary elements.

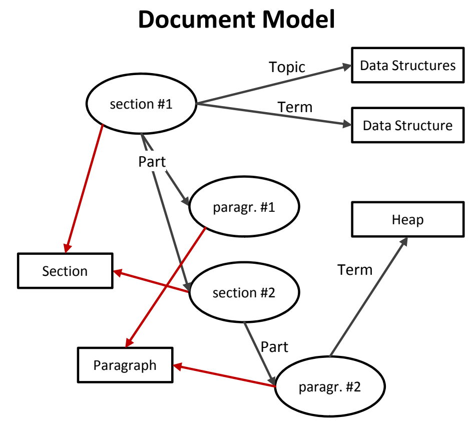
At the beginning of the main processing phase, this list is then extended with the help of a dictionary to cover all possible grammatical forms of the terms, so that they can be easily recognised in the text of the document. This is a reverse form of *stemming*, i.e. the reduction of a word to its base stem [FBY92]. Instead of trying to detect matches to relevant terms by stemming every word in the document, only the relevant terms are stemmed and expanded to cover their grammatical forms ("reverse stemming"). These grammatical forms can then easily be found in the document. Among a collection of dictionaries those matching the language of the document are selected. If the language is not encoded in the HTML markup, it is detected automatically. If automatical detection fails, the user is prompted for the information.



Finally, traversing the DOM tree recursively using XPath, the relevant metadata are collected from the simplified document: information about the fragments is derived from the appropriate structural elements of the DocBook standard such as *chapter*, *section*, *example* or *note*, while information about their content is gathered from their topics and from the list of terms.

- The *structural type* of all fragements found by `//section` is "Section"
- The *function type* of all fragments found by `//partintro` is "Introduction"
- The topic of any fragment is indicated by `./title/text()`
- For all listed terms (and their word forms) T: if a fragment F contains T, then T is a *relevant term* for F
- ...

The extracted metadata information is inserted into the statement-based document model of Section 2.1 using the vocabulary defined in Appendix C.



3.2 Metadata Extraction using XQuery

As a second approach to metadata extraction the Verdikt framework can apply XQuery programs, too. While XQuery is a Turing-complete language and can make full use of XPath expressions, it still has several limitations that need to be addressed. The first and most obvious one is that XQuery programs can only process XML-based documents. Although this problem can be solved easily using the same techniques as for the Java-based extraction, it requires external components that are not part of the XQuery program. The same is true for other pre-processing steps as well: running XSL transformations or converting Microsoft Word files exceeds the scope of the XQuery language. Similar limitations apply to the actual processing of the document: XQuery does not provide the necessary stack-based data structures for a hierarchical context, it lacks support for even basic user interaction, and it cannot determine file checksums or handle data compression or perform a host of other basic yet central tasks.
To overcome these limitations, the Verdikt framework provides a number of extensions to the XQuery language. Implemented as Java callback functions in the Qexo XQuery implementation [Qex07], these extensions provide access to most of the commands and components available to the Java-based metadata extraction described in Section 3.1[3]. For a listing of these XQuery extensions, see Appendix E.1. Some custom auxiliary XQuery functions are defined in the `vdk` namespace. They are not required for metadata extraction, but provide convenient access to commonly used techniques. For a listing of these XQuery functions, see Appendix E.2.
Using the auxiliary functions mentioned above, an XQuery program creates an RDF XML document that represents the document model and that can be imported as a document model by the Verdikt framework. The general approach is illustrated in Figure 7.

***Example* 3.7** (XQuery/HTML)
This example describes how an XQuery program is used to extract a metadata model from documents in HTML format by means of pre-defined CSS classes. Fragments of an XQuery program are shown to illustrate the method.

Example
XQuery/HTML

[3]In Example 3.7 and in Illustration 3.8, all Verdikt extensions to XQuery are syntactically denoted by ^...^

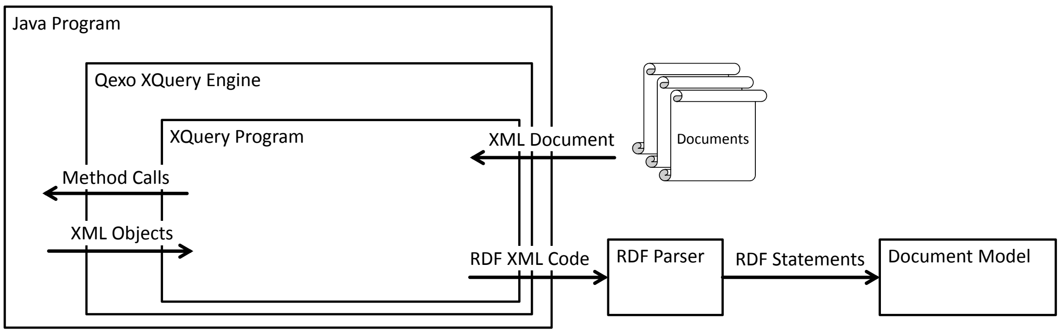
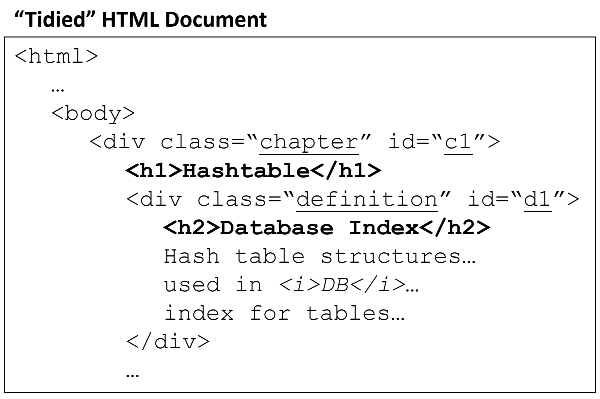


Figure 7: Java Callback in XQuery

Before the document is processed, any syntactical errors found are corrected if possible. This is done automatically using XML Tidy through the Verdikt XQuery extension function `^load($filename)^`.



“Tidied” HTML Document

```
<html>
  …
  <body>
    <div class="chapter" id="c1">
      <h1>Hashtable</h1>
      <div class="definition" id="d1">
        <h2>Database Index</h2>
        Hash table structures…
        used in <i>DB</i>…
        index for tables…
      </div>
      …
```

A recursive traversal of its DOM tree determines the document structure according to HTML headline elements and previously known CSS stylesheet class definitions such as “chapter” or “section” (e.g. line 4 in Illustration 3.8). If no stylesheet definitions are available, the search is restricted to HTML elements, resulting in less detailed structural information. Further querying of stylesheet classes such as “introduction”, as well as recognising distinct keywords such as “Example” or “Definition” in the text help identifying the functions of recognised fragments (line 24).

When looking for content information of fragments (e.g. relevant terms or technical abbreviations), the HTML code itself offers only limited clues. However, HTML elements merely specifying layout can still help to discover at least some important phrases. Emphasised text, e.g. text in italics, is recognised as relevant terms or items (line 31 in Illustration 3.8). Afterwards, a pre-compiled list of domain-relevant abbreviations can be used to detect abbreviations and to separate them from regular terms (line 26).

List of abbreviations

```
<abbreviations>
   <entry>
      <long>Database</long>
      <short>DB</short>
      <short>D.B.</short>
   </entry>
   …
```

This list of abbreviations defines a set of terms (such as "Database") and a list of common abbreviations for each term (e.g. "DB"). Whenever one of the abbreviated forms of a term is encountered in the text, it can be mapped to its extended form. This allows for an abstract view of e.g. "DB" and "D.B." as "Database". Thus, any consistency checking can be done on the representative of the equivalence class formed by the term and its syntactical variations. Moreover, because of this abstraction the consistency checking can even take the usage of the abbreviated and the extended form of a term into account, for example to check that each term is used in its extended form before it is used in its abbreviated form.

As an additional optimisation, a small ontology – basically a list of semantically equivalent terms or terms equivalent except for spelling such as "standby" and "stand-by" – is employed to combine terms that represent the same concept (line 9 in Illustration 3.8).

Ontology of equivalent terms

```
<equivalences>
   <entry>
      <word>Hash table</word>
      <equiv>Hashtable</equiv>
      <equiv>Hash-table</equiv>
   </entry>
   …
```

The ontology defines a set of standardised terms (such as "Hash table"), as well as a list of alternate spellings or synonyms for each term. If one of these alternate forms is encountered in the document, its standardised form is used for the document model instead of the form originally used in the document. This process results in an abstract view of the original document, because terms are grouped together by their semantical meaning, rather then by their purely syntactical representation.

The XQuery program generates a set of RDF statements in XML representation, using the appropriate auxiliary functions (cf. Appendix E.2). For the sake of clarity, the XQuery code fragment shown below in Illustration 3.8 employs a simplified version of these functions: instead of using separate functions for different types of subjects and objects (e.g. content objects or data objects) to assemble statements from (`vdk:statement_subject_content()`, `vdk:predicate_object_data()`, ...), a generic version (`vdk:statement`) that works for all types of subjects and objects is used.

The resulting RDF XML code can be incorporated into the document model by using an RDF parser.

Resulting RDF statements

(doc,	*Part*,	c1)
(c1,	*StructuralType*,	"Chapter")
(c1,	*Topic*,	"Hash table")
(d1,	*FunctionType*,	"Definition")
(d1,	*Topic*,	"Database Index")
(d1,	*Abbreviation*,	"Database")
...		

Illustration
XQuery/HTML

***Illustration* 3.8** (XQuery/HTML)
This illustration shows part of the XQuery code used to process the HTML document as discussed in Example 3.7.

```
(: $element is an XML element representing the current fragment;
     $abbr is an XML structure containing a list of abbreviations;
     $equi is an XML structure containing a small ontology of equivalent terms :)
declare function local:traversal($element, $abbr, $equi) {

    (: detect the document structure from CSS definitions or HTML headlines :)
    if ($element/@class="chapter" or $element/h1)
    then (

        (: find the fragment's title :)
        let $pretitle := $element/span[@class="headline"] | $element/h1,

            (: if there is an equivalent term, use it instead of the fragment's title :)
            $title := if ($equi//entry[equiv=$pretitle])
                 then $equi//entry[equiv=$pretitle]/word
                 else $pretitle
         return (

             (: get the parent fragment from the context and declare the current
                 fragment a part of it :)
             vdk:statement(^context.get("parent")^, "Part", $element/@id),

             (: set the structural type of the current fragment to "Chapter" :)
             vdk:statement($element/@id, "StructuralType", "Chapter"),

             (: set the topic of the current fragment to the one determined above :)
             vdk:statement($element/@id, "Topic", $title)),

             (: save the current element as the new parent in the context :)
             vdk:ignore(^context.add("parent", $element/@id)^)
         )
     ) else (

         (: detect fragment function types from CSS definitions or from the
             content of HTML headlines :)
         if ($element/@class="definition" or
            contains($element/h2, "Definition"))
        then (
            (: set the function type of the current fragment to "Defintion" :)
            vdk:statement($element/@id, "FunctionType", "Definition"),
            ...
        ) else (
            (: detect relevant terms from HTML formatting elements :)
            if (local-name($element)="i")
            then (
                let $term := $element/text()
                return (
                    (: if the detected term is a known abbreviation... :)
                    if ($abbr//entry[short=$term])
                    (: ...then add its long form as an abbreviation
                        to the parent fragment :)
                    then vdk:statement(^context.get("parent")^,
                        "Abbreviation", $abbr//entry[short=$term]/long)
                    (: ...otherwise add the term itself to the parent fragment :)
                    else vdk:statement(^context.get("parent")^,
                        "Term", $term),
                    ...
                )
            ) else (
                (: process the document fragments recursively :)
                for $child in ($element/*)
                return (
                    (: push the current context onto a stack before the recursion... :)
                    vdk:ignore(^context.backup^),
                    local:traversal($child, $abbr, $equi),
                    (: ...and get it back from the stack after returning
                        from the recursion :)
                    vdk:ignore(^context.restore^)
                )
            )
        )
    )
}
```

4 Metadata Storage Requirements

Several requirements for metadata storage have been identified in the course of the Verdikt project:

- First, a storage scheme has to be *independent of the file format* the metadata was extracted from. The same metadata store must be able to handle metadata derived from e.g. HTML sources, Microsoft Word documents, or image files.

- Second, the storage format must be an *open, standardised format.* This allows to creating extensions and optimisations, and facilitates the exchange of metadata between storage units.
- Of course, a *query language* is essential for data access, in particular for extraction purposes.
- To support metadata models of large sizes, the storage scheme has to be scalable. Currently, the availability of a *database implementation* is considered the best way to address this issue.
- Finally, the storage scheme has to be *expressive* enough to represent any metadata model that can conceivably be created.

The W3C recommendation *RDF* [RDF04] satisfies all these requirements. Not surprisingly, the document model defined in Definition 2.1 is compatible with the RDF data model, i.e. a collection of statements of the form (subject predicate object).
Generally, RDF models form directed graphs. For large models, these graphs can become huge, both in terms of node count and number of edges. Several possible optimisations for RDF implementations using databases have been suggested [CDD+04, AMMH07, HLQR07]. However all of them appear to work well only under specific conditions [SGK+08]. While document metadata models exhibit very specific characteristics, no specialised approach has been suggested yet that is capable of exploiting these characteristics for performance gain.
Document metadata, even though formally a graph structure, has very close resemblance to a tree structure. This is due to the original document structure, which is basically tree-like: most documents are divided into sections or chapters, and often subsections or subchapters, branching the document out like a tree. This is evident in various document formats, including XML, HTML or Word. Since this basic structure is retained in the metadata model, only a small number of cross-connections (statements) break up the tree structure into a full graph, most notably cross-references from the original document. As of today, no RDF storage and retrieval implementation is able to exploit this feature for better performance [SF09].

5 Metadata Storage

An evaluation of RDF frameworks has shown the Sesame framework [BKH02] to be the most promising storage solution for document metadata models available at the time of the study [SF09]. The evaluation also revealed a problem common to all tested RDF frameworks: they lack support for reification in their Java interfaces. The ability to make statements about other statements can be critical for adding information regarding the quality of other metadata. See Figure 8 for an example of a statement where reification is used to annotate the statement with information about its quality. Therefore, a solution to circumvent this limitation is required.
Fortunately, only Sesame's Java interface lacks support for reifying statements: its underlying database schema, its RDF parser, and its SPARQL [SPA08] query interface all support reification. Therefore, merely an insertion procedure for reified statements had to be defined. This was exploited in the Verdikt framework to create a work-around: creating and modifying reified statements is done by generating RDF XML code for the statements in

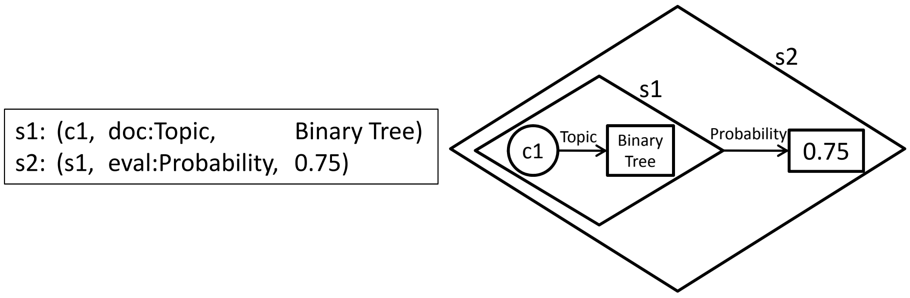

s1: (c1, doc:Topic, Binary Tree)
s2: (s1, eval:Probability, 0.75)
s2
s1
c1
Topic
Binary Tree
Probability
0.75

Figure 8: Reified statement (s1) as RDF triples (left hand side) and as RDF graph (right hand side)

question, which is then inserted into the Sesame database via the RDF parser. Querying reified statements is done using a combination of SPARQL queries. Figure 9 shows an overview of the work-around. In the future, new versions of Sesame, or other RDF frameworks that support reification directly might be used as the RDF management back end for the Verdikt framework, hopefully rendering the current work-around obsolete.

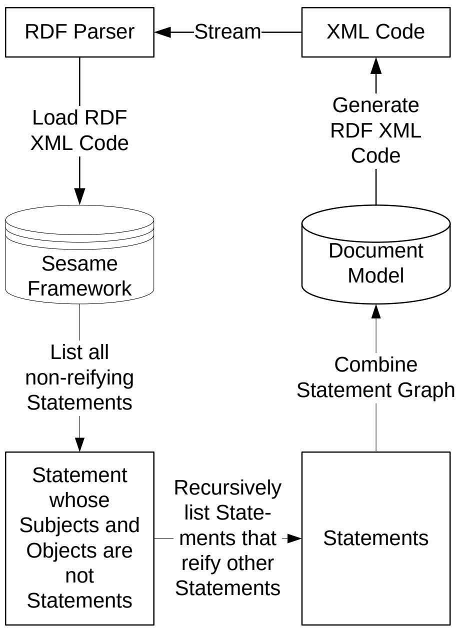

RDF Parser
Stream
XML Code
Load RDF XML Code
Generate RDF XML Code
Sesame Framework
Document Model
List all non-reifying Statements
Combine Statement Graph
Statement whose Subjects and Objects are not Statements
Recursively list Statements that reify other Statements
Statements

Figure 9: Employing Reification in Combination with the Sesame Framework

***Note* 5.1**
The Jena framework in its current version (2.6.0) offers some limited support for reification in its Java interface. While this support makes a complex work-around as for the Sesame framework unnecessary, it was clearly added as an afterthought. Therefore, using reification

with Jena still requires additional implementation effort, which will not be detailed here.

Inserting/modifying reified statements To insert reified statements into the Sesame database, the Verdikt framework generates RDF XML code for each of these statements. Figure 10 shows the XML code corresponding to the RDF statements in Figure 8. This XML code is then loaded into the Sesame database via Sesame's RDF parser.

```
<rdf:Description rdf:ID="c1">
   <doc:Topic rdf:ID="s1">Binary Tree</doc:Topic>
</rdf:Description>

<rdf:Description rdf:about="s1">
   <eval:Probability rdf:ID="s2">0.75</eval:Probability>
</rdf:Description>
```

Figure 10: Reified statement (s1) as RDF XML code

Querying reified statements Retrieving reified statements involves a more complex procedure. First, a list of all "pure" statements that do not reify other statements (i.e. that have both subjects and objects which are not statements themselves) is obtained through a SPARQL query. Based on this initial set of statements, a set of reified statements is generated by a recursive procedure that successively retrieves reifying statements. This procedure starts with the pure, non-reification statements, and creates a list of statements that reify these pure statements. Then, the procedure creates a list of statements that reify the statements that were found in the previous iteration of the procedure. This process is repeated, until no new statements are found.

Algorithm
Sesame
Reification
Retrieval

***Algorithm* 5.1** (Sesame Reification Retrieval)
The following algorithm returns a list of all statements from a SPARQL-enabled RDF store, including reified statements. It uses a custom data model for RDF statements instead of the Sesame model to be able to express reification. Statements are implicitly converted into the custom data model immediately after being returned by the SPARQL query engine, which still uses the Sesame data model.

```
LIST-STATEMENTS()
1  statements ← SPARQL-QUERY(list all resources r where
             (r rdf:type rdf:Statement) and
             (r rdf:subject s) and not (s rdf:type rdf:Statement) and
             (r rdf:object o) and not (o rdf:type rdf:Statement)
   )
2  return LIST-STATEMENTS-RECURSIVELY(statements)
```

```
LIST-STATEMENTS-RECURSIVELY(statements)
1  reification-statements ← NIL
2  for each statement stmt from statements
3       do
4           reification-statements ← reification-statements +SPARQL-QUERY(
                      list all resources r where (r rdf:type rdf:Statement) and
                      (r rdf:subject stmt) or (r rdf:object stmt)
            )
5
6  if reification-statements ≠ NIL
7     then
8           reification-statements ←
                      LIST-STATEMENTS-RECURSIVELY(reification-statements)
9  return statements + reification-statements
```

Since the Sesame SPARQL-QUERY method returns all resources using its own data model – which does not support reification – employing a single SPARQL query to list all (reified) statements is impossible: the Sesame data model immediately looses any reification relation between statements.

***Proposition* 5.1** (Termination of LIST-STATEMENTS)
The procedure LIST-STATEMENTS terminates iff the auxiliary procedure LIST-STATEMENTS-RECURSIVELY terminates.
The procedure LIST-STATEMENTS-RECURSIVELY terminates iff the RDF store does not contain cyclic reification of statements, i.e. iff there are no statements $t_0 = (s_0, p_0, o_0)$, …, $t_n = (s_n, p_n, o_n)$, $n \in \mathbb{N}_>$, such that $t_i = s_{i+1}$ (or $t_i = o_{i+1}$), for all $0 \leqslant i < n$ and $t_n = s_0$ (or $t_n = o_0$).

Proposition
Termination of LIST-STATEMENTS

An evaluation of RDF querying languages has shown that SPARQL and SeRQL [BK03] are the most suitable languages for our purposes [SF09]. While both are supported by the Sesame framework, the W3C recommendation SPARQL is more widely applied. Therefore, SPARQL was selected as the query language of choice for RDF retrieval in the Verdikt project. There are, however, still two problems with the SPARQL language. The first problem is SPARQL's lack of support for general recursive queries of arbitrary depth and even for querying the transitive closure of properties. The second problem is caused by the fact that blank nodes have no features that uniquely identify them across multiple query results. As a consequence, it is impossible to combine multiple results.

Recursive queries Currently, SPARQL does not support the recursive traversal of property paths (cf. [SPA09], Section 2.6). For example, there is no mechanism to express queries like "Find all fragments that are transitively part of the main document". Only fragments that are direct parts of the document can be found, but it is impossible to describe a transitive part-of relation. The only way to retrieve all parts of parts of …is to determine the document depth beforehand and to unfold the recursive query down to that depth.

Blank nodes The lack of recursive queries is aggravated by the second problem: it is almost impossible to combine the results of multiple queries. Identifiers for so-called blank nodes, i.e. nodes without textual content or a URI reference ("content objects" in the Verdikt context), are assigned on a temporary basis, and they are only unique in the context of the query result the blank nodes occur in. Identifiers for blank nodes in different query results have no relation to one another. Therefore, the result of a first query cannot be used as the starting point for a second query. As case in point, this makes it impossible to implement recursive queries by issuing repeated queries, each one building on the results of the last.

The first problem can be bypassed by ignoring the query language mechanism completely and working solely with the class model provided by the Java interface. Here, blank nodes can be uniquely identified, which makes it possible to use the approach of repeated queries mentioned above.
There is no generic solution to the second problem known to the authors – apart from adding custom, externally generated identifiers to all blank nodes on a permanent basis, which is akin to abstaining from using blank nodes entirely and replacing them with URI resources ("vocabulary objects" in the Verdikt context). This, however, defeats the purpose of blank nodes, which is to provide nodes that can be used locally, but which cannot be referenced from an external context. The solution adopted for the Verdikt system is that whenever blank nodes are used, the system disregards the RDF query engine and falls back on the Java interface.

***Note* 5.2**
Ignoring the query language mechanism is almost the opposite approach to the one required for using reification, precluding any attempts to work with either the Java interface or the query language exclusively.

6 Further Metadata Processing

After the metadata information has been extracted from a document and stored in a database, it needs to be processed further and used to generate a verification model that can be used for model checking (cf. Figures 1 and 2). The verification model is an abstraction of the original document consisting of

- a set of states, each representing some fragment of the document
- a set of concepts and roles attached to each state, each representing a number of facts about that state
- a list of connections (transitions) between states.

More details about the verification model (cf. Illustrations 6.2 and 6.4) and on the model checking process itself can be found in [Wei08].
The exact content of the verification model – e.g. fragments of what structural types are represented as states, or which metadata is represented as concepts or roles – depends on external requirements, however. Therefore, the document metadata is stored in a document model, and customised verification models are generated from the document model as required. Since these requirements are not necessarily known beforehand, the metadata model has to be as complete as possible in order to accommodate them. There are four main aspects that can be customised:

Structural Units First, the structural units to be represented as states need to be determined. This is not always straightforward because different abstraction levels may be required depending on the specific application. For example, one application only needs to regard the document structure on the level of chapters, while another requires a more detailed view on the level of paragraphs or something in between. The more detailed view would overload the model for the first application with too many details, hampering both runtime performance and user understanding of the verification results, while the less detailed view would be insufficient for the second application. Thus, the structural type of fragments that are to be mapped to the states of the verification model needs to be adjusted according to the current application.

***Note* 6.1**
Concepts, as well as roles, are defined in the temporal description logic $\mathcal{ALC}\mathsf{CTL}$ [Wei08]. $\mathcal{ALC}\mathsf{CTL}$ is a combination of $\mathcal{ALC}$ [BFH00] and CTL [BHWZ02]: $\mathcal{ALC}\mathsf{CTL}$ concepts and roles inherit the set semantics of $\mathcal{ALC}$, where concepts are interpreted as sets of individuals, and roles are interpreted as relations.

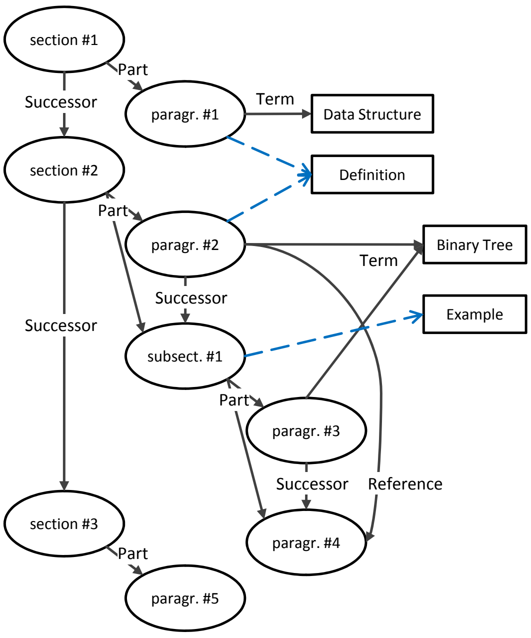
Concepts Second, the metadata to be represented as concepts has to be selected, and the concepts need to be named. A concept, resolved at some state, represents a set of objects that satisfy some semantical stipulation defined by that concept. For example, in Section 2 of this report, a concept named "IllustratedTopic" would yield the set {Document, Document Model}. The concept name "IllustratedTopic", as well as the semantics of "all titles of illustrations" have to be defined for the creation of the verification model.

Roles Next, the metadata to be represented as roles has to be selected, and the roles need to be named as well. Roles are similar to concepts, except that they return sets of two-tuples of objects, instead of flat sets of objects. For example, in Section 3.1 of this report, a role named "topicOf", defining the relation between fragments and topics, would yield the set {(Definition 3.1, Document Context), (Illustration 3.3, Document Context), (Example 3.6, Java/DocBook)}.

Quality Finally, an optional lower bound on the quality of the metadata to be considered can be defined. Some metadata extraction techniques are not perfectly accurate: a degree of uncertainty is introduced into the data model. Possible measures of uncertainty include actual correctness or probability of correctness. A minimum quality of metadata for different measurements can be specified to exclude data of low quality.

***Illustration* 6.2** (Model Generation)
The illustration shows how a verification model is generated from a document model:

Illustration
Model Generation



Let us define the parameters for model generation as follows:

- States: `//*[structural type = 'Section']`
  Select all fragments that are sections

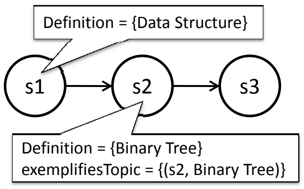
- Concepts:

  – DefinedTopic: `//*[function type = 'Definition']/Part*/Term/*`
  Select all term data that is a descendant (part of a part of a...) of a fragment that is a definition

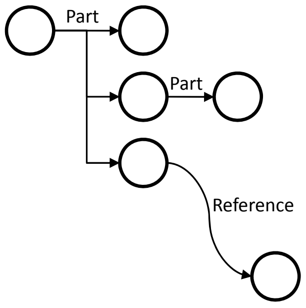
- Roles:

  – exemplifiesTopic: `//*[FunctionType='Example']` $\rightarrow$
  `(./>Part*[StructuralType='Section'], ./Part*/Term/*)`
  Start with all fragments that are examples. Then select all parent fragments (reverse of a part of a part of a...) that are sections and relate them to all descendent (part of a part of a...) term data.

The following verification model will be generated:


Definition = {Data Structure}
s1
s2
s3
Definition = {Binary Tree}
exemplifiesTopic = {(s2, Binary Tree)}

***Note* 6.3**
Neither of the concepts of Illustration 6.2 could have been defined using conventional means (XPath or SPARQL) due to the recursive `/Part*/` (or `/>Part*/`, respectively) path specification:



In XPath, recursion cannot be limited to certain paths. An XPath recursion (e.g. `descendent-or-self::Term`) would follow all descending paths through the document that lead to a "Term" node. There is no way to restrict the recursive descent to "Part" paths. Thus, XPath would not only return all terms that occur in parts of the current node, but also any terms that can be reached on other paths, e.g. by following the "Reference" path indicated above. This, however, would violate the semantics (and obviously the intent) of the rules specifying the concepts.
In contrast to XPath, SPARQL does not allow recursion at all (cf. Section 5).

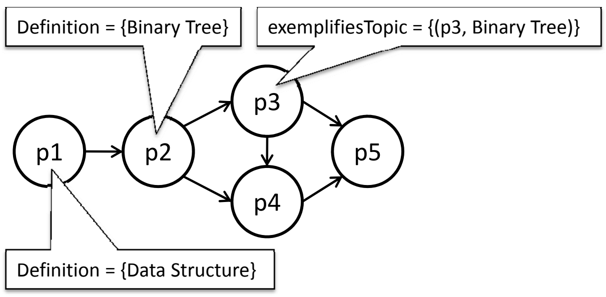
When generating a verification model from a document model, different parameters for the model generation process can lead to very different verification models.

Illustration
Model Generation II

***Illustration* 6.4** (Model Generation II)
Altering the parameter defining the states leads to a verification model different from that of Illustration 6.2:

- States: `//*[structural type = 'Paragraph']`
  Select all fragments that are paragraphs

The following verification model will be generated:


Definition = {Binary Tree}
exemplifiesTopic = {(p3, Binary Tree)}
p1
p2
p3
p4
p5
Definition = {Data Structure}

***Note* 6.5**
The structure of the verification model of Illustration 6.4 differs significantly from that of Illustration 6.2. In particular, the verification model of Illustration 6.4 shows that it is possible to skip the example for binary trees (state p3) in the original document, while the verification model of Illustration 6.2 suggested otherwise (state s2). Thus, different parameters can lead to different models and even to different verification results for the same verification criteria. A criterium specifying that for every definition an example must follow would be satisfied for the verification model of Illustration 6.2, but would be violated for the verification model of Illustration 6.4, even though both verification models were generated from the same document model.

Currently, the Verdikt framework provides two approaches to generating verification models from document models. One is based on using XQuery on a specialised XML serialisation of the RDF graph. The other approach is based on a Java implementation of a set of specification rules.

XML serialisation Because XQuery programs can only be applied to XML data, the RDF graph has to be converted into an XML DOM tree. While a standard exists for representing RDF as XML code [RDF04], this standard allows for many (often substantially) different XML representations of the same RDF data. This makes it all but impossible to create an XQuery program that runs on this data, because it would have to take every possible XML representation into account. The solution is to create an RDF XML serialisation (or an XML *view*, in database terminology) that is unambiguous, i.e. that is always the same for the same RDF data. Details on this approach can be found in [SF09]. As its main drawback it does not scale well for large RDF graphs, because the serialisation instantiates every RDF reference, leading to a replication of data: if two RDF nodes a and b both have edges to a third RDF node c, then the XML serialisation needs to duplicate c and create an edge from a to c_1 and another edge from b to c_2. This scales poorly for graphs with a high number of edges in proportion to nodes.

Java rules The second approach can be applied directly to the RDF graph. It is based on a number of navigation and filter rules such as “all paths with ‘part’-statements” and “all statements with object values of ‘Chapter’ ”, that can be used to select subgraphs, separate some nodes and include others (cf. Illustration 6.2). They are utilised to define states, concepts, roles, and quality constraints for the verification model as described above. A more detailed description of this approach will be available in a forthcoming report.
The Java approach scales better than the XML serialisation, yet suffers from a different drawback: it requires Java programming skills. However, a graphical interface to resolve this issue is currently being developed.

7 Conclusion

In the course of this report, we have presented an overview of the three main components of the model generation aspect of the Verdikt project: the *metadata extraction* from various document formats, the *metadata storage* in an RDF database framework, and the *generation* of a customised *verification model* from the available metadata.

Limitations of the components have been pointed out. Possibilities for improvements have been discussed as well.
In future work, some of the limitations shall be mitigated, in particular the lack of support for recursive queries of non-predetermined depth in metadata querying. Also, enhancements will be devised, among others the utilisation of known structural characteristics of document metadata for database storage and retrieval.

References

[AMMH07] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *VLDB'07: Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 411–422. ACM, 2007.

[Ber01] Daniel Charles Berrios. *Methods for semi-automated index generation for high precision information retrieval.* PhD thesis, Stanford University, 2001. Adviser – Lawrence M. Fagan.

[BFH00] Alexander Borgida, Enrico Franconi, and Ian Horrocks. Explaining alc subsumption. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI)*, pages 209–213, Berlin, Germany, 2000. IOS Press.

[BHWZ02] S. Bauer, I. Hodkinson, F. Wolter, and M. Zakharyaschev. On Non-Local Propositional and Local One-Variable Quantified CTL*. In *Proceedings of TIME'02*, pages 2–9, Manchester, UK, 2002. IEEE Computer Science Press.

[BK03] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. In *SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003.

[BKH02] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *ISWC 2002: Proceedings of the First International Semantic Web Conference*, pages 54–68. Springer, 2002.

[CDD$^+$04] J. J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson. Jena: implementing the semantic web recommendations. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 74–83, New York, NY, USA, 2004. ACM.

[CM04] Valter Crescenzi and Giansalvatore Mecca. Automatic information extraction from large websites. *Journal of the ACM*, 51(5):731–779, 2004.

[DIT07] OASIS Darwin Information Typing Architecture. Available online at http://www.oasis-open.org/committees/dita, 2007. last visited Aug. 2009.

[Doc09] OASIS DocBook. Available online at http://www.oasis-open.org/docbook/, 2009. last visited Aug. 2009.

[DRO09] Drools Business Logic integration Platform. Available online at http://www.jboss.org/drools/, 2009. last visited Aug. 2009.

[FBY92] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures & Algorithms.* Prentice-Hall, Englewood Cliffs, NJ, 1992. Chapter 8.

[Ger03] Gerd Wagner and Said Tabet and Harold Boley. MOF-RuleML: The Abstract Syntax of RuleML as a MOF Model. Integrate 2003, OMG Meeting, available online at http://www.omg.org/docs/br/03-10-02.pdf, 2003. last visited Aug. 2009.

[Ham99] Hamish Cunningham. Information Extraction, a User Guide. Research Memo CS-97-02, University of Sheffield, Sheffield, 1999. Update of 1997 version.

[HLQR07] Ralf Heese, Ulf Leser, Bastian Quilitz, and Christian Rothe. Index Support for SPARQL. In *ESWC 07: Proceedings of the European Semantic Web Conference*, 2007.

[JF08] Mirjana Jakšić and Burkhard Freitag. Temporal Patterns for Document Verification. Technical Report MIP-0805, University of Passau, Germany, 2008.

[KSS04] Reiner Kuhlen, Thomas Seeger, and Dietmar Strauch, editors. *Grundlagen der praktischen Information und Dokumentation*. K. G. Saur Verlag, 2004.

[Mic00] Michael E. Tipping. The Relevance Vector Machine. In *Advances in Neural Information Processing Systems 12*, page 652658. MIT Press, 2000.

[Mil02] Robert C. Miller. *Lightweight Structure in Text*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 2002.

[Ope09] OpenOffice. Available online at http://www.openoffice.org/, 2009. last visited Aug. 2009.

[Qex07] Qexo – The GNU Kawa implementation of XQuery. Available online at http://www.gnu.org/software/qexo/, 2007. last visited Aug. 2009.

[RDF04] RDF/XML Syntax Specification (Revised). Available online at http://www.w3.org/TR/rdf-syntax-grammar/, 2004. last vis. Aug. 2009.

[Rob05] Robert Baumgartner and Oliver Frölich and Georg Gottlob and Patrick Harz and Marcus Herzog and Peter Lehmann. Web Data Extraction for Business Intelligence: the Lixto Approach. BTW 2005, 2005.

[SF09] Christian Schönberg and Burkhard Freitag. Evaluating RDF Querying Frameworks for Document Metadata. Technical Report MIP-0903, University of Passau, Germany, 2009.

[SGK$^+$08] L. Sidirourgos, R. Goncalves, M. L. Kersten, N. Nes, and S. Manegold. Column-Store Support for RDF Data Management: not all swans are white. In *VLDB'08: Proceedings of the International Conference on Very Large Data Bases*, Auckland, New Zealand, September 2008. ACM.

[SPA08] SPARQL Query Language for RDF. Available online at http://www.w3.org-/TR/rdf-sparql-query/, 2008. last visited Aug. 2009.

[SPA09] SPARQL New Features and Rationale. Available online at http://www.w3.org/TR/2009/WD-sparql-features-20090702/, 2009. last visited Aug. 2009.

[Ste03] Stefan Kuhlins and Ross Tredwell. Toolkits for generating wrappers a survey of software toolkits for automated data extraction from Web sites. In *LNCS*, pages 184–198. Springer, 2003.

[STS00] Eleni Stroulia, Judi Thomson, and Gina Situ. Constructing xml-speaking wrappers for web applications: Towards an interoperating web. *Reverse Engineering, Working Conference on*, 0:59, 2000.

[Süß05] Christian Süß. *Eine Architektur für die Wiederverwendung und Adaptation von eLearning-Inhalten*. PhD thesis, University of Passau, 2005.

[Wei08] Franz Weitl. *Document Verification with Temporal Description Logics.* PhD thesis, University of Passau, 2008.

[WJF09] Franz Weitl, Mirjana Jakšić, and Burkhard Freitag. Towards the Automated Verification of Semi-structured Documents. *Data & Knowledge Engineering*, 68:292–317, 2009.

[Wor06] WordNet – a lexical database for the English language. Available online at http://wordnet.princeton.edu/, 2006. last visited Aug. 2009.

[XML04] XML Information Set (Second Edition). Available online at http://www.w3.org/TR/xml-infoset/, 2004. last visited Aug. 2009.

[XML08] XML/HTML Tidy. Available online at http://www.w3.org/People/Raggett/tidy/, 2008. last visited Aug. 2009.

[XPA99] XML Path Language (XPath). Available online at http://www.w3.org/TR/xpath/, 1999. last visited Aug. 2009.

[XPa07] XML Path Language (XPath) 2.0. Available online at http://www.w3.org/TR/xpath20/, 2007. last visited Aug. 2009.

A List of relevant Metadata

This list describes the metadata that is relevant in the scope of the Verdikt project.

A.1 Structural Types of Fragments

Name	Description
Book	Describes a complete book as a structural element. Can have multiple parts or chapters.
Bookpart	Describes a part of a book as a structural element. Can have multiple chapters.
Chapter	Describes a chapter as a structural element.
Subchapter	Describes a subchapter that is part of a chapter as a structural element.
Section	Describes a section as a generic structural element that can be used at different levels of a document.
Screenunit	Describes a screenunit as a structural element, for example in e-learning documents.
Paragraph	Describes a paragraph as a generic structural element at the lowest level of a document.
Level	Describes the level of a structural element, e.g. for layered sections. Depending on the document structure, a level may not be well-defined.
Initial	Describes a possible initial point of a document, that is the start of a reading path. There can be more than one initial point, or none.
Final	Describes a possible final point of a document, that is the end of a reading path. There can be more than one final point, or none.

A.2 Function Types of Fragments

Name	Description
Paragraph	Describes the most generic type of fragment, an untyped paragraph.
Definition	Describes a fragment containing a definition.
Example	Describes a fragment containing an example.
Theorem	Describes a fragment containing a theorem.
Corollary	Describes a fragment containing a corollary.
Lemma	Describes a fragment containing a lemma.
Proposition	Describes a fragment containing a proposition.
Proof	Describes a fragment containing a proof.
Note	Describes a fragment containing a note.
Hint	Describes a fragment containing a hint.
Algorithm	Describes a fragment containing an algorithm.
Sourcecode	Describes a fragment containing source code.
Task	Describes a fragment containing a task.
Solution	Describes a fragment containing a solution.
Table	Describes a fragment containing a table.
List	Describes a fragment containing a list.
Numbered List	Describes a fragment containing a numbered list.
Image	Describes a fragment containing an image.
Animation	Describes a fragment containing an animation, e.g. Adobe Flash.
Audio	Describes a fragment containing audio data.
Video	Describes a fragment containing video data.

A.3 Content

Name	Description
Topic	Describes a topic in a document, e.g. a headline.
Subtopic	Describes a subtopic in a document, e.g. a subheading.
Language	Describes the language of a document or document fragment, in a standard encoding.
Summary	Describes a short summary of a document or document fragment.
Term	Describes a term in basic grammatical form (nominativ singular) that is relevant for a document or for a document fragment.
Programming Language	Describes the Name of a programming language in which a document fragment is written.

A.4 References

Name	Description
Reference	Describes an explicit and direct reference, e.g. “href”.
Successor	Describes a structural successor, e.g. between two chapters.
Part	Describes a structural part-of relation, e.g. between a chapter and a subchapter.
Quote	Describes a literature reference.
Indirect Reference	Describes an implicit and not neccessarily distinct reference, e.g. using a key word.
Disambiguation	Describes a reference to different meaning of a term, e.g. for homonyms.
Related	Describes related terms, e.g. as part of taxonomies.
Unrelated	Describes similar or equal terms that are nonetheless unrelated. See “Disambiguation”.
Same	Describes different terms that point to the same abstract object, e.g. synonyms.
Prerequisite	A term or structural unit should be placed before another on a reading path to facilitate better understanding for the user.

A.5 Didactical Information

Name	Description
Target Group	Describes a target group for a document or a document fragment.
Duration	Describes the probable time required to read or work through a document or document fragment.
Importance	Describes the relecancy of a document or document fragment in relation to other documents or document fragments.
Intensity/ Complexity	Describes the intensity or difficulty of a document or document fragment.
Medium	Describes the target medium of a document or document fragment, e.g. web page, screen document or printed version.
Objective	Describes the learning objective of a document or document fragment. The vocabulary for the objectives should be identical to that of the terms (see above).

A.6 Source

Name	Description
Filename/URI	Describes a local or remove storage location for a source file.
Format	Describes the document format of the source file.
Import Method	Describes the method used to import the source file.
Linenumber	Describes the line number of the source file from which certain data was read.
Internal ID	Describes an internal ID of the source file from which certain data was read.

A.7 Versioning

Name	Description
Version of	Describes the relation to other models of the same document or document fragment in other versions.
Version Number	Describes the version number of a document or document fragment.
Last Change	Describes the time and date of the last change to a source file.
Checksum	Describes the checksum of a source file.

A.8 Errors

Name	Description
Access Error	Describes an access error to a source file, e.g. missing access rights.
Format Error	Describes a format error in a source file, e.g. invalid XML code.
Other Error	Describes other errors, e.g. a memory overflow.

A.9 Evaluation

Name	Description
Quality	Describes the quality of a metadatum in quantitative terms, e.g. "75% true".
Probability	Describes the probability of a metadatum, e.g. "with 75% probability true".
Correctness	Describes the correctness of a metadatum using fuzzy logics, e.g. "The truth is 75% similar to this datum".
Relevancy	Describes the relevancy of a reference, e.g. for indirect references.
Source	Describes the way a metadatum was derived, e.g. "fact", "guess", "conclusion", "inference".
Truth	Describes the final truth value of a metadatum, after consideration of all factors.

A.10 Dublin Core Metadata (extract)

Name	Description
Creator	Describes the creator of a document or document fragment.
Date	Describes the creation date and time of a document or document fragment.
Description	Describes a summary of the content of a document or document fragment.
Format	Describes the document format of a a document or document fragment, in a standard encoding.
Language	Describes the language of a document or document fragment, in a standard encoding.
Relation	Describes the relation of a document or document fragment to other documents or document fragments.
Subject	Describes the topic of a document or document fragment.
Title	Describes the title of a document or document fragment.

B List of available Metadata

This list describes the metadata that is available in various document formats.

B.1 LMML

Metadata	Availability
Structural Types	Complete
Function Types	Complete
Content	Complete, language only implicitly known
References	No content relations (same, related, etc.)
Didactical Information	No duration
Source	No line number (depending on the XML parser)
Versioning	Only partially supported by the format, has to be managed externally
Errors	Complete
Evaluation	No inference or uncertain data, only relevancy
Dublin Core Metadata	No relations, language only implicitly known

B.2 $< \mathrm{ML}^3 >$

Metadata	Availability
Structural Types	Complete
Function Types	Complete
Content	No language
References	No content relations (same, etc.)
Didactical Information	No importance
Source	No line number (depending on the XML parser)
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	No inference or uncertain data
Dublin Core Metadata	No relations or language

B.3 SCORM

Metadata	Availability
Structural Types	Undefined
Function Types	Undefined
Content	Complete, but only on the top document level
References	Only semantical references
Didactical Information	No importance and objectives, medium only implicitly known
Source	No line number (depending on the XML parser)
Versioning	Partially supported through the lifeCycle element
Errors	Complete
Evaluation	No inference or uncertain data
Dublin Core Metadata	Complete

B.4 DocBook

Metadata	Availability
Structural Types	Complete
Function Types	Only sparsely supported, very generic
Content	Terms and summary not supported directly, language not supported
References	No content relations and no indirect references
Didactical Information	Undefined
Source	No line number (depending on the XML parser)
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	No inference or uncertain data
Dublin Core Metadata	No relations or language

B.5 DITA

Metadata	Availability
Structural Types	Only with topic maps
Function Types	Only with othermeta elements or other DITA extensions
Content	Complete, terms only with DITA extensions
References	No content relations and no indirect references
Didactical Information	Only target group and importance
Source	No line number (depending on the XML parser)
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	No inference or uncertain data
Dublin Core Metadata	No relations

B.6 HTML

Metadata	Availability
Structural Types	Limited
Function Types	Undefined
Content	Only language, or with uncertainty
References	Only direct references, semantical references partially supported through link elements
Didactical Information	Undefined
Source	No line number (depending on the XML parser)
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	Inferences have to be made externally
Dublin Core Metadata	Completely available in therory, but rarely used

B.7 HTML with CSS and Information Extraction

Metadata	Availability
Structural Types	Complete
Function Types	Depending on CSS definitions and IE
Content	Complete
References	Only direct references
Didactical Information	Undefined
Source	No line number (depending on the XML parser)
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	Inference can be done by the IE component
Dublin Core Metadata	Completely available in therory, but rarely used

B.8 LaTeX

Metadata	Availability
Structural Types	Complete
Function Types	Available through definition and usage of high-level commands
Content	Only topics
References	No content relations and no indirect references
Didactical Information	Undefined
Source	Complete
Versioning	Not supported, has to be managed externally
Errors	Complete
Evaluation	Inferences have to be made externally
Dublin Core Metadata	Partially available through definition and usage of high-level commands

C List of Metadata Vocabulary

This list describes the predefined vocabulary used to describe document models, grouped by namespace.

C.1 Vocabulary in the `verdikt/Document/` namespace

document:structuralType The structural type of a fragment, e.g. “Chapter” (see Section 2).

***Example* C.1** (document:structuralType)

$$\left\langle (\text{chapter3})\,,\ \xrightarrow{\text{structuralType}},\ [\text{“Chapter”}] \right\rangle$$

$$\left\langle (\text{section3.1})\,,\ \xrightarrow{\text{structuralType}},\ [\text{“Section”}] \right\rangle$$

Example
document:structuralType

document:functionType The function type of a fragment, e.g. “Example” (see Section 2).

***Example* C.2** (document:functionType)

$$\left\langle (\text{section1.1})\,,\ \xrightarrow{\text{functionType}},\ [\text{“Introduction”}] \right\rangle$$

$$\left\langle (\text{paragraph4.3.2})\,,\ \xrightarrow{\text{functionType}},\ [\text{“Definition”}] \right\rangle$$

Example
document:functionType

document:topic The topic of a fragment, usually the title.

document:term A precise term in normal form using a standardised vocabulary, e.g. “Binary Tree”.

document:initial Indicates whether or not a fragment is an initial node, e.g. is the first step on a path through the document. Linear documents only have a single initial fragment, but non-linear documents with a complex structure and multiple starting points may have more than one.

***Example* C.3** (document:initial)

$$\left\langle (\text{section1.1})\,,\ \xrightarrow{\text{initial}},\ [\text{“true”}] \right\rangle$$

$$\left\langle (\text{paragraph4.3.2})\,,\ \xrightarrow{\text{initial}},\ [\text{“false”}] \right\rangle$$

Example
document:initial

document:language The natural language of a fragment in standardised form, e.g. "en_us".

***Example* C.4** (document:language)

$$\left\langle (\text{section3.1}), \xrightarrow{\text{language}}, [\text{“en\_uk”}] \right\rangle$$

Example document:language

C.2 Vocabulary in the `verdikt/Reference/` namespace

reference:part Indicates that a fragment is part of another fragment, regarding the document structure (see Section 2). Fragments that are part of other fragments can inherit properties such as topics or didactic information.

***Example* C.5** (reference:part)

$$\left\langle (\text{chapter2}), \xrightarrow{\text{part}}, (\text{section2.1}) \right\rangle$$

If (chapter2) deals with a certain topic, then it can be inferred that (section2.1) also deals with that topic, or with a more specific subset of it.

Example reference:part

reference:successor Indicates that a fragment follows directly after another (see Section 2).

***Example* C.6** (reference:successor)

$$\left\langle (\text{chapter2}), \xrightarrow{\text{successor}}, (\text{chapter3}) \right\rangle$$

Example reference:successor

***Note* C.7**
One fragment can only have one successor, according to the document order.

reference:reference A general purpose reference, used for cross references and for references with unclear semantics.

reference:citation A citation. Used for literature references.

reference:external Reference to an external entity, usually a web page.

C.3 Vocabulary in the `verdikt/Source/` namespace

source:filename The source filename (relative or absolute) of a fragment. This information is used to keep track of the connection between the original document and its metadata model.

source:URI The source URI of a fragment (for web-based documents). This information is used to keep track of the connection between the original document and its metadata model.

source:lastChange The time stamp of the last change of the source file. Used to detect new versions of documents in the corpus that already have a model representation.

source:checksum The checksum of the source file (at the time of metadata extraction). Used to detect new versions of documents in the corpus that already have a model representation.

source:format The document format of the source file, e.g. "Microsoft Word" or "HTML".

C.4 Vocabulary in the `verdikt/Didactic/` namespace

didactic:duration Time that the associated content should take to work through for a reader.

didactic:audience The targeted audience for the content of a fragment, e.g. "Students".

didactic:intensity The difficulty level of the content of a fragment, e.g. "Advanced".

didactic:medium The intended target medium for the content of a fragment, e.g. "Screen" or "Paper".

C.5 Vocabulary in the `verdikt/Error/` namespace

error:parse Number of parse errors encountered in the source document. This number can indicate possible data loss due to corrupted source files.

error:message An error message recorded for the source document. This message can indicate possible data loss due to corrupted source files.

C.6 Vocabulary in the `verdikt/Evaluation/` namespace

evaluation:quality The quality of a given statement (quantitative measurement). A higher number indicates a higher quality. This is similar to marks ranging from A to F, or from 1 to 5.

***Example* C.8** (evaluation:quality)

$$\left\langle \langle \mathsf{S} \rangle , \xrightarrow{\mathsf{quality}}, [\text{“0.75”}] \right\rangle$$

Example
evaluation:quality

evaluation:probability The probability of a given statement (probabilistic measurement). The primary source for probabilistic data is information obtained using techniques of machine learning.

evaluation:correctness The correctness of a given statement (fuzzy measurement). This type of measurement is currently only used for experimentation purposes.

evaluation:source Information about how the data was derived, e.g. “Fact”, “Guess”, “Reasoning”, “Inference”...

***Example* C.9** (evaluation:source)

$$\left\langle \langle \mathsf{S} \rangle , \xrightarrow{\mathsf{source}} , [\text{“inference”}] \right\rangle$$

Example
evaluation:source

evaluation:truth The truth value of a given statement, either set by the user from external facts or background knowledge, or inferred from the other measurements. This value is an abstraction of the other quality measurements. A truth value of 0.0 indicates a false statement, a truth value below 0.5 indicates a statement that should not be trusted, a truth value above 0.5 indicates a statement that might be trusted, and a truth value of 1.0 indicates a statement that is true.

C.7 Vocabulary in other namespaces

The complete Dublin Core vocabulary can also be used.

D List of DocumentAdapter Methods

This list describes the auxiliary methods provided by the abstract class `DocumentAdapter`.

`String getXSDType(String javatype)` Converts a Java type into an XSD type.

`void openWordnet()` Initialises the Wordnet database for use.

`void closeWordnet()` Closes the Wordnet database after use.

`List<String> getRootWords(String word, String language)` Returns all roots (stems) of a word, using Wordnet.

`edu.mit.jwi.item.POS getPartOfSpeech(String word, String language)` Trys to determine the PartOfSpeech of a word. Returns NULL if unsuccessful. Returns only one POS, even if multiple options exist, in the order Noun, Verb, Adjective, Adverb. Uses Wordnet.

`List<String> getRelatedWords(String word, [boolean samePOS], String language)` Returns a list of semantically related words, using Wordnet.

`List<String> formatWordnetWords(List<String> words)` Formats a list of words that were returned by Wordnet. I.e. replaces “_” with “ ”.

`List<String> getLanguageList()` Returns a list of all language names known to the system.

`List<String> getLanguageCodes(String language)` Returns the codes associated with a language, e.g. "en_us", "en_uk" for "English".

`String getLanguageFromCode(String code)` Returns the language associated with a code, e.g. "German" for "de_de". Returns the empty string in case of failure.

`String getDictionary(String language)` Finds a dictionary for a language. Returns the empty string in case none is found.

`String getDocumentLanguage(Node node, [ResourceObject document], [VerdiktMetadataModel model], [String defvalue])` Returns the language of the current document. If automatic determination fails, the user is asked.

`List<String> extendTerm(String term, String language)` Extends a term by different grammatical word forms. The list returned contains a copy of the original term at position 0.

`Frame getParent()` Returns the parent frame used for swing dialog display.

`void setParent(Frame parent)` Sets the parent frame used for swing dialog display.

`Document loadXML(String filename/InputStream input/InputSource input) throws LoadException` Loads an XML file into an XML DOM tree.

`Document loadWord(String filename) throws LoadException` Loads a Microsoft Word file into an XML DOM tree. Converts the Word file to XML and loads it.

`String loadString(String filename) throws LoadException` Loads a text file into a string.

`void saveString(String filename, String value) throws SaveException` Saves a string to a text file.

`void saveXML(String filename, Node node) throws SaveException` Saves an XML document to a file.

`NodeList xpath(String exp, Node node) throws Exception` Evaluates an XPath expression and returns the result as a node set.

`String xpathValue(String exp, Node node) throws Exception` Evaluates an XPath expression and returns the result value as a string.

`String getTempFilename([String extension])` Generates a filename for a temporary file. Note that this file will be automatically deleted when the virtual machine exits.

`void setXMLImplementation(String value)` Sets the XML DOM implementation that is used.

`void setSAXImplementation(String value)` Sets the SAX implementation that is used.

`Node runXSL(Node tree, String xsl, [String paramname], [Object paramvalue]) throws LoadException` Run an XSLT stylesheet on an XML tree.

`String getPathFromFilename(String filename)` Extracts the path name from a complete filename and converts all "\" to "/". The resulting path includes the final / (e.g. /tmp/example/test.xml results in /tmp/example/).

`String getFileFromFilename(String filename)` Extracts the file name portion from a complete filename (e.g. /tmp/example/test.xml results in text.xml).

`String getLastPathFromFilename(String filename)` Extracts the last path part name from a complete filename (e.g. /tmp/example/test.xml results in example).

`String getNextToLastPathFromFilename(String filename)` Extracts the next to last path part name from a complete filename (e.g. /tmp/example/test.xml results in tmp).

`String getChecksum(String filename/Node node) throws LoadException` Returns the checksum hash of a file/XML node. Note that it is not guaranteed which hash implementation is used.

`String getTextContent(Node node)` Extracts the entire text content from a node. Replaces node.getWholeText() for implementations that do not support this method.

`String xmlEscape(String s)` Escapes XML special characters in a string.

`String xmlEscapeText(String s)` Escapes XML tag characters in a string (does not escape quotation marks).

`String nodeToString(Node node, [int indent])` Returns a string representation of an XML fragment. This String can be used to write to an XML file.

`String nodeInnerToString(Element node)` Returns a string representation of all children of an XML fragment.

`String getStringFromUser(String title, String message, [String[] list], [String defaultvalue])` Asks the user for a simple string value from a list.

`void showProgress(String message, [String note], [int min], [int max])` Shows a progress dialog with progress between min and max and with an initial note.

`void updateProgress(int progress/String note)` Updates the progress to a specific value/note. A value change has no effect for indeterminate progress.

`void finishProgress()` Closes the progress dialog.

`VerdiktObject getDocumentObject(String filename, ResourceObject corpus, MetadataModel model, [String defvalue])` Returns the name of the current document (the document that the file indicated by filename is a part of).

`VerdiktObject getCorpusObject(String filename, MetadataModel model, [String defvalue])` Returns the name of the current corpus (the corpus that the file indicated by filename is a part of).

`boolean canLoad()` Indicates whether or not the document adapter supports the load operation.

`boolean canSave()` Indicates whether or not the document adapter supports the save operation.

`VerdiktMetadataModel load([String filename], [VerdiktMetadataModel model]) throws LoadException` The import method. Call this method to get the metadata content from a file or other data storage facility. The method updates the metadata model passed as a parameter.

`void save([String filename], VerdiktMetadataModel model) throws SaveException` The export method. Call this method to save the metadata content to a file or other data storage facility.

`String getDefaultFilename()` Provides a default filename or default access information for user dialogs.

`List<String> getDefaultFileExtensions()` Returns the default file extensions for documents made available by the adapter.

`boolean isFileAdapter()` Indicates whether or not the document adapter accesses files or something else (e.g. a database).

`String getName()` Returns the name of the adapter.

`String getDescription()` Returns a description of the adapter.

`void setOption(String name, String value)` Sets an option with a specified name and value. Usage of these options depends on the particular adapter.

`String getOption(String name)` Returns the value of an option with a specified name. Usage of these options depends on the particular adapter.

`List<String> listOptions()` Returns all names of options that are currently set.

`HashMap<String,String> getOptions()` Returns the complete set of options.

`void setOptions(HashMap<String,String> opts)` Replaces the complete set of options.

E List of XQuery Extensions and Functions

E.1 List of XQuery Extensions

This list describes the XQuery extensions of the Verdikt framework. For each command, a short description and its return value is given, as well as its return type and when the return value is computed. Possible computing times are *load-time*, i.e. when the XQuery program is read by the processor, but before it is executed, and *run-time*, i.e. during the execution of the XQuery program.

`^print(message)^` Prints the message on the screen and returns it. *(String, run-time)*

`^requestStop()^` Returns true, if the run-time environment requests a stop of operations, e.g. if the maximum number of files have been processed, false otherwise. *(boolean, run-time)*

`^corpus^` Returns the name of the current Corpus. *(String, load-time)*

`^document^` Returns the name of the current Document. *(String, load-time)*

`^generate-id()^` Generates a unique ID. *(String, run-time)*

`^verdikt^` Generates the Verdikt XQuery functions declarations, see Appendix E.2. *(String, load-time)*

`^visited(filename)^` Returns true if the file has been processed before, false otherwise. *(String, run-time)*

`^filename^` Returns the default filename, either from the parameters or from querying the user. *(String, load-time)*

`^filepath(basepath, filename)^` Calculates the current absolute path out of a base path and the filename. *(String, run-time)*

`^load(file)^` Loads an XML file, corrects XML errors (e.g. HTML errors), saves the corrected XML code to a temporary file and returns its filename. *(String, run-time)*

`^loadWord^` Loads a Microsoft Word file, converts it to XML, saves the XML code to a temporary file and returns its filename. *(String, run-time)*

`^save(xmlvar, filename)^` Saves XML code to a file. Returns true on success, false on failure. *(boolean, run-time)*

`^errors()^` Returns the number of errors encountered during loading. *(int, run-time)*

`^checksum(file)^` Calculates the checksum of a file. *(String, run-time)*

`^type(var)^` Returns the Java type of a variable. *(String, run-time)*

`^context.set(name, value, [hierarchical])^` Adds a name/value pair to the current context and returns the value. The optional hierarchical parameter indicates if the element should be ignored by the backup/restore methods (hierarchical=0), or if it should be included (default, hierarchical=1). *(type of value, run-time)*

`^context.add(name, value, [hierarchical])^` Adds a list with the given name to the current context (if it does not already exist), appends the value to it and returns the value. The optional hierarchical parameter indicates if the element should be ignored by the backup/restore methods (hierarchical=0), or if it should be included (default, hierarchical=1). *(type of value, run-time)*

`^context.get(name, [index])^` Retrieves a value from the context, using the specified name. If the value is a list, either the last element is returned, or that indicated by index. If the value is not a list, the optional index parameter is ignored. If the name does not exist or the list is empty, a null value is returned. *(type of value, run-time)*

`^context.remove(name, [index|value])^` Removes the last entry or the entry indicated by index or the entry with the indicated value from a list indicated by name and returns it. If the name does not exist or the list is empty, a null value is returned. Note: Stack behaviour can be achieved by removing elements with no index parameter, while queue behaviour can be achieved by removing elements with an index of 0. *(type of value, run-time)*

`^context.empty(name)^` Checks if the list indicated by name is empty. If a list of this name does not exist, true is returned as well. *(boolean, run-time)*

`^context.clear(name)^` Removes all elements from the list indicated by name. Returns true. *(boolean, run-time)*

`^context.size(name)^` Returns the size of the list indicated by name. *(int, run-time)*

`^context.contains(name, value)^` Checks if the list indicated by name contains the indicated value. *(boolean, run-time)*

`^context.removeDuplicates(name)^` Removes any duplicate entries from the list indicated by name. Returns true. *(boolean, run-time)*

`^context.backup()^` Saves the current context information to a stack and returns null. This method is usually called before a recursion, so that any changes made to the context during the recursion can be restored afterwards. *(null, run-time)*

`^context.restore()^` Restores the context to a former state from a stack and returns null. This method is usually called after returning from a recursion, so that any changes made to the context during the recursion can be restored afterwards. *(null, run-time)*

`^rdf(file)^` Loads the contents of an RDF XML file and generates the appropriate definitions. *(String, load-time)*

`^askUser(title, message)^` Displays a message box with a text input field to the user and returns the user's input. *(String, run-time)*

`^true^` Returns true. *(boolean, run-time)*

`^false^` Returns false. *(boolean, run-time)*

`^^` Inserts the character '^' into the text, used for escaping single instances of ^, since those would result in a load error. *(String, load-time)*

E.2 List of XQuery Functions

This list describes the XQuery functions defined in the Verdikt namespace.

`vdk:model($content)` Generates the RDF declaration for a Verdikt Metadata Model, including the content.

`vdk:statement_subject_content($id, $predicate)` Generates the RDF declaration for a statement, with a content object as the subject. The predicate has to be generated using one of the `vdk:predicate_object_...()` functions.

Example
Statement

***Example* E.1** (Statement)
The XQuery code

```
vdk:statement_subject_content($document,
      vdk:predicate_object_data('doc:', 'StructuralType', 'Document'))
```

produces the RDF statement $\left\langle (\mathsf{Document}\ \alpha)\,, \xrightarrow{\mathsf{StructuralType}}, [\text{“}\mathsf{Document}\text{”}] \right\rangle$.

`vdk:statement_subject_vocable($ns, $name, $predicate)` Generates the RDF declaration for a statement, with a vocabulary object as the subject. The predicate has to be generated using one of the `vdk:predicate_object_...()` functions.

`vdk:statement_subject_statement($id, $predicate)` Generates the RDF declaration for a statement, with a statement object as the subject. The predicate has to be generated using one of the `vdk:predicate_object_...()` functions.

`vdk:predicate_object_content($ns, $name, [$statementid], $objectid)` Generates the RDF declaration for the predicate and the object of a statement, with a content object as the object.

`vdk:predicate_object_data($ns, $name, [$statementid], $data, [$type])` Generates the RDF declaration for the predicate and the object of a statement, with a data object as the object.

`vdk:predicate_object_vocable($ns, $name, [$statementid], $vns, $vname)` Generates the RDF declaration for the predicate and the object of a statement, with a vocabulary object as the object.

`vdk:predicate_object_statement($ns, $name, [$statementid], $objectid)` Generates the RDF declaration for the predicate and the object of a statement, with a statement object as the object.

`vdk:ignore($data)` Simply ignores the data and returns (), used for hiding unwanted return values.